

# **Applied Compositional Thinking for Engineers**



You are reading a work-in-progress book created for the class Applied Compositional Thinking for Engineers (ACT4E):

[applied-compositional-thinking.engineering](http://applied-compositional-thinking.engineering)

Please visit the website to get the last updated edition.

Compiled on 2023-11-03.

**Instructor copies:** If you are an instructor, please contact us to obtain a copy of the book with the solutions to the graded exercises.



# Contents

## Contents

### A. INTRODUCTION

#### 1. The view from above

1.1. Sapiens: a retrospective . . . . .	4
1.2. Compositionality for engineers . .	5
1.3. Departures from tradition . . . . .	6
1.4. Acknowledgments . . . . .	8

#### 2. Putting things together

2.1. Stacking blocks . . . . .	11
2.2. Mixing colors . . . . .	14
2.3. Recipes as ingredients . . . . .	16
2.4. Commutativity and associativity .	17
2.5. Composing recipes . . . . .	20
2.6. Isomorphisms . . . . .	21

#### 3. Exercises

3.1. Introduction . . . . .	24
3.2. Code exercises - Overview . . . . .	25
3.3. Exercise tutorial . . . . .	28
3.4. Turning in exercises . . . . .	29

### B. SETS, FUNCTIONS, RELATIONS

#### 4. Sets and functions

4.1. Logical preliminaries . . . . .	36
4.2. Sets . . . . .	39
4.3. New sets from old . . . . .	44
4.4. Code exercises - Type theory . . . .	49
4.5. Code exercises - Representing sets	51
4.6. Code exercises - FiniteSetProperties	57
4.7. Code exercises - Power set . . . . .	58
4.8. Code exercises - Set products . . .	59
4.9. Code exercises - Disjoint union . .	61
4.10. Functions . . . . .	62
4.11. Code exercises - Mappings . . . . .	70
4.12. Code exercises - FiniteMapOperations . . . . .	71
4.13. Further constructions . . . . .	72

#### 5. Relations

5.1. Distribution networks . . . . .	76
5.2. Relations . . . . .	79
5.3. Code exercises - Relation representation . . . . .	80
5.4. Composing relations . . . . .	81
5.5. Code exercises - Composition . . .	83
5.6. Relations and functions . . . . .	84
5.7. Properties of relations . . . . .	87
5.8. Code exercises - Relation properties	88
5.9. Transpose of a relation . . . . .	89
5.10. Code exercises - Transpose . . . . .	90
5.11. Endorelations . . . . .	91
5.12. Code exercises - Endorelation properties . . . . .	93
5.13. Transitivity . . . . .	94
5.14. Code exercises - Transitivity . . . .	95
5.15. Equivalence relations . . . . .	96

### C. ALGEBRA

#### 6. Sets with operations

6.1. Magmas . . . . .	104
6.2. Semigroups . . . . .	105
6.3. Code exercises - Semigroups . . . .	109
6.4. Code exercises - Free semigroup . .	111
6.5. Monoids . . . . .	112
6.6. Dynamical systems . . . . .	115
6.7. Code exercises - Monoids . . . . .	116
6.8. Groups . . . . .	117
6.9. Code exercises - Groups . . . . .	122
6.10. Rings, fields . . . . .	123
6.11. Vector spaces . . . . .	124

#### 7. Morphisms

7.1. Semigroup morphisms . . . . .	126
7.2. Encoding as morphism . . . . .	128
7.3. Morse coding . . . . .	129
7.4. Monoid morphisms . . . . .	131
7.5. Group morphisms . . . . .	133
7.6. Generators and relations . . . . .	135
7.7. Code exercises - Morphisms . . . .	138

#### 8. Actions

8.1. A simplest dynamical system . . .	142
8.2. Actions . . . . .	143
8.3. Contravariant actions . . . . .	146
8.4. Actions of matrix groups . . . . .	147

<b>D. ORDER</b>	<b>151</b>	13.4. Diagrams . . . . .	219
<b>9. Posets</b>	<b>153</b>	13.5. Categories vs graphs . . . . .	221
9.1. Trade-offs . . . . .	154	13.6. Categories from graphs . . . . .	222
9.2. Ordered sets . . . . .	158	13.7. Code exercises - Semicategories . .	224
9.3. Counting orders . . . . .	163	<b>14. Categories and structures</b>	<b>231</b>
9.4. Code exercises - Posets representation	164	14.1. Categories of sets and functions . .	232
9.5. Power poset . . . . .	166	14.2. Categories of relations . . . . .	233
9.6. Code exercises - Power poset . . . .	167	14.3. Categories of semigroups, monoids, groups . . . . .	234
9.7. Chains and Antichains . . . . .	168	14.4. Categories from linear algebra . . .	235
9.8. Code exercises - Chains and an- tichains . . . . .	170	14.5. Categories of posets . . . . .	236
9.9. Measuring posets . . . . .	171	14.6. Sets with data . . . . .	237
9.10. Code exercises - Measuring posets	172	14.7. Categories of graphs . . . . .	238
9.11. More general preferences . . . . .	173	14.8. Preorders as categories . . . . .	239
<b>10. Constructing posets</b>	<b>175</b>	14.9. Monoids as categories . . . . .	240
10.1. Product of posets . . . . .	176	<b>15. Modeling with categories</b>	<b>241</b>
10.2. Code exercises - Poset product . . .	178	15.1. Mobility . . . . .	242
10.3. Disjoint union of posets . . . . .	179	15.2. Trekking in the Swiss Mountains .	244
10.4. Code exercises - Disjoint union of posets . . . . .	180	15.3. Currency categories . . . . .	246
10.5. Opposite of a poset . . . . .	181	15.4. Resources dependencies . . . . .	249
10.6. Code exercises - Opposite poset . .	182	15.5. Procedures . . . . .	253
10.7. “Twisted” poset of intervals . . . .	183	15.6. Software dependencies . . . . .	256
10.8. Code exercises - Twisted poset con- struction . . . . .	185	15.7. Code exercises - Currency categories	257
10.9. Arrow poset of intervals . . . . .	186	<b>16. Semicategory actions</b>	<b>259</b>
10.10. Code exercises - Arrow poset con- struction . . . . .	187	16.1. Moore machines, first version . . .	260
<b>11. Monotonicity</b>	<b>189</b>	16.2. The category $\langle \mathbf{Set} \rangle$ . . . . .	264
11.1. Monotone maps . . . . .	190	16.3. Moore machines, $\langle \mathbf{Set} \rangle$ version . .	266
11.2. Antitone maps . . . . .	194	16.4. Standard action of Moore machines	269
11.3. Code exercises - Monotone maps .	195	16.5. Semicategory actions . . . . .	272
11.4. Compositionality of monotonicity .	196	16.6. Different actions of Moore machines	274
11.5. Order on monotone maps . . . . .	197	16.7. More machines . . . . .	277
<b>E. CATEGORIES</b>	<b>201</b>	16.8. LTI systems . . . . .	279
<b>12. Graphs</b>	<b>203</b>	<b>17. Constructing categories</b>	<b>285</b>
12.1. Graphs . . . . .	204	17.1. Product of Categories . . . . .	286
12.2. Graph homomorphisms . . . . .	206	17.2. Disjoint Union of Categories . . . .	287
<b>13. (Semi)categories</b>	<b>209</b>	17.3. Opposite Category . . . . .	288
13.1. Interfaces . . . . .	210	17.4. Arrow construction . . . . .	289
13.2. Semicategories . . . . .	212	17.5. Twisted arrow construction . . . .	290
13.3. Categories . . . . .	216	17.6. (Co)slice construction . . . . .	292
		<b>18. Culture</b>	<b>293</b>
		18.1. Definition vs computation . . . . .	294
		18.2. Things that don’t matter . . . . .	295
		18.3. Choice of symbols . . . . .	296
		18.4. Typographical conventions . . . . .	297

<b>F. FUNCTORS</b>	<b>303</b>	25.6. Sum and intersection of DPIs . . .	375
<b>19. Translation</b>	<b>305</b>	<b>26. Feasibility</b>	<b>377</b>
19.1. Layers of abstraction . . . . .	306	26.1. DPs as monotone maps . . . . .	378
19.2. Semifunctors . . . . .	307	26.2. DPs as upper sets . . . . .	380
19.3. Functors . . . . .	308	26.3. Querying design problems . . . . .	382
19.4. More examples of functors . . . . .	311	26.4. Series composition . . . . .	384
19.5. Categorical Databases . . . . .	315	26.5. Identity for DP . . . . .	387
<b>20. Specialization</b>	<b>317</b>	26.6. The category of design problems <b>DP</b>	391
20.1. Subcategories . . . . .	318	<b>27. Lattices</b>	<b>395</b>
20.2. Subcategories of endomorphisms .	319	27.1. Monoidal posets . . . . .	396
20.3. Other examples . . . . .	320	27.2. Monoidal-time procedures . . . . .	398
20.4. Subcategories of Berg . . . . .	321	27.3. Lattices . . . . .	400
<b>21. Syntax and semantics</b>	<b>323</b>	27.4. Lattice homomorphisms . . . . .	403
21.1. Specification versus behavior . . .	324	27.5. Categories <b>Lat</b> and <b>BoundedLat</b> .	404
<b>22. Up the ladder of abstraction</b>	<b>327</b>	<b>28. Lattice structure of DPs</b>	<b>405</b>
22.1. Functor composition . . . . .	328	28.1. Ordering DPs . . . . .	406
22.2. A category of categories . . . . .	329	28.2. Interaction with series composition	407
22.3. Products and sums of functors . . .	330	28.3. Union of Design Problems . . . . .	408
<b>G. CO-DESIGN</b>	<b>335</b>	28.4. Intersection of Design Problems . .	409
<b>23. Design</b>	<b>337</b>	28.5. Lattice structure of <b>DP</b> hom-sets .	410
23.1. What is “design”? . . . . .	338	28.6. Interaction with composition . . .	413
23.2. What is “co-design”? . . . . .	339	<b>29. Constructing design problems</b>	<b>415</b>
23.3. Formal engineering design . . . . .	341	29.1. Companion and conjoint use . . .	416
23.4. Queries in design . . . . .	343	29.2. Companions and conjoint . . . . .	418
<b>24. Poset bounds</b>	<b>345</b>	29.3. Monoidal DPs . . . . .	420
24.1. Minimal and maximal elements . .	346	<b>H. INTERCONNECTIONS</b>	<b>423</b>
24.2. Code exercises - Minimal and maxi- mal elements . . . . .	347	<b>30. Parallel composition</b>	<b>425</b>
24.3. Upper/lower bounds . . . . .	348	30.1. Modeling parallelism . . . . .	426
24.4. Code exercises - Upper/lower bounds	350	30.2. Stacking semicategories . . . . .	429
24.5. Upper and lower sets . . . . .	351	30.3. Associative stacking (semi)categories	431
24.6. Code exercises - Upper and lower sets	352	30.4. Examples of associative stacking semicategories . . . . .	435
24.7. Antichains . . . . .	353	30.5. Functorial stacking semicategories	439
24.8. Code exercises - Upper and lower closure . . . . .	356	30.6. Functorial stacking categories . . .	444
<b>25. Monotone Co-Design Theory</b>	<b>357</b>	30.7. Monoidal stacking (semi)categories	445
25.1. DPIs . . . . .	358	30.8. Strict monoidal (semi)categories .	448
25.2. Examples . . . . .	361	<b>31. Crossing wires</b>	<b>449</b>
25.3. Queries . . . . .	367	31.1. Morphism-representable wire cross- ings . . . . .	450
25.4. Co-design problems . . . . .	369	31.2. A strict monoidal category of permu- tations . . . . .	452
25.5. The semicategory <b>DPI</b> . . . . .	373		

31.3. Symmetric stacking semicategories . . . . .	454	35.2. Isomorphism is not identity . . . . .	521
<b>I. NATURALITY</b>	<b>461</b>	<b>K. COMPOSITIONAL COMPUTATION</b>	<b>525</b>
<b>32. Naturality</b>	<b>463</b>	<b>36. DP queries as functors</b>	<b>527</b>
32.1. Visual Example and Warmup . . . . .	464	36.1. Queries are functors from problem statements to solutions . . . . .	528
32.2. Natural transformations . . . . .	465	36.2. The $\mathbf{Pos}_U$ and $\mathbf{Pos}_L$ categories . . . . .	530
32.3. Morphisms in a Category of Functors . . . . .	469	36.3. Queries as functors . . . . .	542
32.4. Data migration . . . . .	473	<b>37. Solving finite co-design problems</b>	<b>547</b>
32.5. More examples . . . . .	475	37.1. Domain theory and fixed points . . . . .	548
32.6. Yoneda Embedding Theorem & Lemma . . . . .	478	37.2. Finite co-design problems . . . . .	552
<b>33. Adjunctions</b>	<b>479</b>	37.3. Handling loops . . . . .	553
33.1. Formal concept analysis . . . . .	480	37.4. Example: Optimizing over the natural numbers . . . . .	556
33.2. Galois connections . . . . .	486	37.5. Extended Numerical Examples . . . . .	559
33.3. Adjunctions: hom-set definition . . . . .	488	37.6. Complexity of the solution . . . . .	568
33.4. Adjunctions: (co)unit definition . . . . .	489	37.7. Decomposition of CDPs . . . . .	570
33.5. Product-Hom adjunction . . . . .	491	<b>38. Monads</b>	<b>573</b>
33.6. Free-forgetful adjunction . . . . .	492	38.1. Generalized objects and operations . . . . .	574
33.7. Relating the two definitions . . . . .	494	38.2. Monads . . . . .	579
<b>34. Traced Monoidal categories</b>	<b>495</b>	38.3. The Kleisli construction . . . . .	583
34.1. Monoidal categories . . . . .	496	38.4. Algebras of a monad . . . . .	585
34.2. Examples of monoidal categories . . . . .	499	38.5. Monads from Adjunctions . . . . .	589
34.3. Symmetric monoidal categories . . . . .	501	<b>BACK MATTER</b>	<b>593</b>
34.4. Monoidal functors . . . . .	503	<b>Example exams</b>	<b>603</b>
34.5. Dual objects and morphisms . . . . .	505	1. Exam 1 . . . . .	604
34.6. Trace for monoidal categories . . . . .	507	<b>REFERENCES</b>	<b>605</b>
<b>J. UNIVERSAL CONSTRUCTIONS</b>	<b>515</b>		
<b>35. Sameness</b>	<b>517</b>		
35.1. Sameness in category theory . . . . .	518		

# INTRODUCTION | PART A.



---

1. The view from above	3
2. Putting things together	9
3. Exercises	23

---







# 1. The view from above

In this first chapter, we give some motivation to look at the material in this book.

What we aim to do is to accompany you to the top of the Peak of Abstraction and show you *the view from above*, from which you will be able to see that many things that look different have the same structure.

- 1.1 Sapiens: a retrospective . . . . . 4
- 1.2 Compositionality for engineers . 5
- 1.3 Departures from tradition . . . . 6
- 1.4 Acknowledgments . . . . . 8

## 1.1. *Sapiens*: a compositional retrospective

The word *intelligence*, from Latin *intellego*, comes from Proto-Indo-European *\*h<sub>1</sub>entér* (“between”) and *\*leg* (“to gather”), and we can translate it as *the ability to gather things together* to obtain some goal; this, for us, is the essence of intelligence.

The *things* to gather could be abstract, such as pieces of evidence to achieve a conclusion, or physical, like ingredients to prepare a tasty meal, or the parts to create machines that will prepare tasty meals.

Intelligence is not unique to *Homo sapiens*; other animals can reason, build, and use language. Some animals excel at things that *sapiens* cannot do. The sapienses will never experience the richness of the smells of things and processes in the world like *canis* does. The sapienses will never have the same spatial awareness of an octopus with eight arm-legs and neurons distributed all along its body.

But sapienses developed a trick no other animal did. They mastered **abstraction and compositionality**. Composing and decomposing is what gave sapiens an edge.

Sapiens decomposed the process of survival and created **societies**. Once, a single sapiens had the ability to survive by themselves, or in a very small pack of sapiens. Eventually they figured out that it was much more efficient to divide up the work, so that some could specialize in hunting, some in gathering, some in fighting, some in rearing children.

About 10,000 years ago, sapiens invented **agriculture**; it was a momentous change, as it was the first time that they could change the world around them and bend nature to their will. Up to that moment, it was the other way around: as they moved beyond Africa, sapienses adapted *to* the environment; bodies optimized to run after prey in savannas became optimized to fish in tropical seas or to herd cattle on the Alps.

More specialization. Today only 1% work at food production. In fact, you could take most sapiens and put them in the most fruit-rich plains, the most prey-rich savanna, and, alone, they would die in days.

With a sedentary society, while the grains and the rice grew by themselves in the field, they found the time to invent **writing**. They managed to decompose thought into a sequence of symbols, which could be written on clay, and re-composed back by the receiver to reconstruct the original thought. Writing is a teleportation device and a time machine.

**Money** is abstraction of resources.

Artisans, creating the product from start to finish, are inefficient. In the **industrial age**, workflows are decomposed in steps, and a system is put in place for the product to be assembled from each step.

Lately, Sapiens has cultivated an inclination to create **machines that could help them think**. The most important conceptual shift is that Sapiens needs now to express knowledge *formally* and *computationally*. By “formally”, we mean the choice of a formal system shared by man and machine. By “computationally”, we mean that such knowledge needs to be able to produce actual results to use.



## 1.2. Compositional thinking for engineers

The thesis of this book is that most engineering fields would benefit from knowing and using the language of applied category theory to address the design and analysis of *complex systems*.

What is a “system”?

Here is a great quote\*:

*A system is composed of components;  
a component is something you understand.*

Howard Aiken

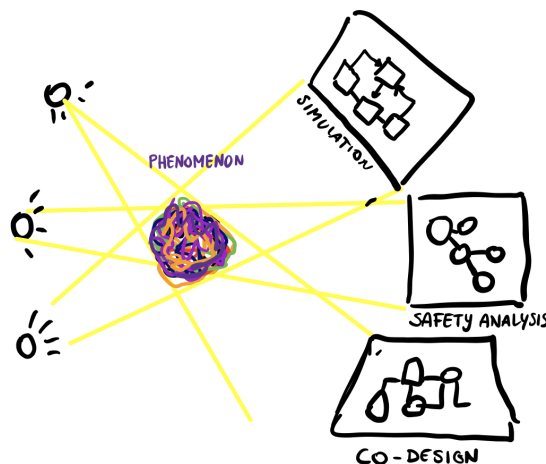
The first part of the quote, “A system is *composed of components*”, is plain as day as much as it is tautological. We could equally say: “A system is *partitioned in parts*”.

The second part, “a component is something you understand”, is where the insight lies: we call “system” what is too complex to be understood naturally by a human.

Aiken referred to computer engineering, but we find exactly the same sentiment expressed in other fields. In systems engineering, Leveson puts it as “complexity can be defined as intellectual unmanageability” [16].

We will be content of this anthropocentric and slightly circular definition of systems and complexity: “systems” are “complex” and “components” are “simple”.

Whether something is a complex system also depends on the task that we need to do with it. One way to visualize this is to imagine a “phenomenon” as a high-dimensional object that we can see from different angles (Fig. 1). For each task, we have a different projection. The decomposition of the system in components can be different according to the task. For example, a system that might be easy to simulate could be very difficult to control.



**Figure 1.:** Engineers live in a Plato’s cave with multiple light sources. A certain phenomenon can be illuminated from different angles and look very different.

The tools presented in this book will make it easier to think about different representations of the same system.

\* This quote is by Howard Aiken (1900-1973), creator of the MARK I computer, as quoted by Kenneth E. Iverson (1920-2004), creator of programming language APL, as quoted in [21], but ultimately source-less and probably apocryphal.

### 1.3. Departures from traditional exposition [experts only]

**This section is for experts only. Skip at a first reading.**

This section describes the “departures from tradition” in our text. We made several choices to streamline the traditional exposition of category theory, to make it more understandable and relevant to the engineering field.

#### Induction vs deduction

The greatest difference between this text and a mathematical text is the use of an **inductive exposition** rather than a deductive explanation. In a typical mathematical exposition of category theory, one defines a general mathematical structure, and then give several specific examples [26].

Instead, here we first build up the examples as something that is interesting per se, and then we show how they can all be instances of the same general concept. In this way, the general concept is well motivated. The path laid by the book is one of *spiral learning*.

For example, we look at various constructions from specific to general:

$$\text{set product} \rightarrow \text{poset product} \rightarrow \text{categorical product.} \quad (1)$$

Similarly, we discuss

$$\text{monoid morphisms} \rightarrow \text{category actions} \rightarrow \text{functors.} \quad (2)$$

#### Materials covered

- ▷ Certain topics (limits, Yoneda’s lemma, *etc.*) that would be traditionally discussed relatively early, are not discussed in this volume. We ordered topics by usefulness in engineering.
- ▷ The main text uses traditional set theory. To ground the exercises, we use slightly more formal **type theory** foundations (setoids, *etc.*). It is in our plan to transition completely to type theory also in the main text. Please contact us if you can help!

**Table 1.1.:** Use of colors

sets	<b>A, B</b>
posets	<b>P, Q</b>
categories	<b>C, D</b>
objects	<i>X, Y</i>
morphisms	<i>f</i> : <i>X</i> → <i>Y</i>
functors	<i>F</i> : <b>C</b> → <b>D</b>
natural transformations	<i>α</i> : <i>F</i> ⇒ <i>G</i>

#### Use of colors

- ▷ We **use colors** to aid in the parsing of formulas and diagrams (Table 1.1). We also color the composition operations. In this way it is easy to see the types at first glance: *f* ; *g*, *F* ; *G*, *etc.*
- ▷ Color is *not* necessary to infer meaning. The choice of colors is **colorblind-friendly** for red-green color blindness. (One of the authors is colorblind.) Please let us know if this is not the case.

#### Notation and conventions

- ▷ In general, we use diagrammatic notation *f* ; *g* (pronounced “*f* then *g*”) rather than *g* ◦ *f* (pronounced “*g* after *f*”) for function and morphism composition.
- ▷ In the discussion of semigroups, we use “;” rather than “◦” as the semigroup composition operation. This is because for us a semigroup/monoid is a special (semi)category with only one object.

- ▷ In Chapter 8 we discuss *covariant* and *contravariant* actions. We do not use the terms *left* and *right* actions because they are notation-dependent.
- ▷ We abundantly use *semicategories* (semifunctors, *etc.*). For us, semicategory is the primitive definition. A category is a semicategory with a particular property: having identities at each object.

### Extensive use of tuples

We extensively use tuples and tuples concatenation to work directly with strict monoidal categories.

This is a list of standard categories together with their “tupled” definition.

**Table 1.2.:** Tuple subcategories of well-known categories

original	tuples subcategory
<b>Set</b> (Def. 13.11)	<b>⟨Set⟩</b> (Def. 16.3)
<b>Pos</b> (Def. 14.8)	<b>⟨Pos⟩</b> (Def. 30.12)
<b>Rel</b> (Def. 14.1)	<b>⟨Rel⟩</b> (Def. 30.14)

### Treatment of monoidal categories

We noticed that there is a step increase in difficulty associated to natural transformations, without much immediate justification. In our trajectory, natural transformations appear first associated to monoidal categories. The role they play there is associated to very technical checks. Nothing exciting! We made the decision to provide a version of monoidal categories that are strict, so that there is no need for natural transformations.

We also noticed sever examples of interest (*e.g.* proper LTI system) that are only semicategories, but they still have a notion of trace.

Furthermore, we noticed that in applications that there are several interesting examples that have a notion of vertical composition but the monoidal structure is not functorial on the nose: for example, systems with states.

Because there is a large part of concrete code exercises, it was not convenient for us to just wave our hands and say things like “let’s just consider the strict / modulo isomorphism version”.

In conclusion, in Chapter 30 we introduce several notions of “stacking” categories, which are defined for semicategories, are strict in the vertical composition operation, and for which the functoriality of the monoidal structure is not a given.

We provide the traditional exposition of monoidal categories in Chapter 34 after the introduction of natural transformations in Chapter 32.

## 1.4. Acknowledgments

There is a multitude of causes that made it possible for this book to come into creation. Here is a short, incomplete list of the people that contributed to make this happen.

We thank David Spivak (Topos Institute) for starting the epidemic of applied category theory that infected us.

We thank Joshua Tan (University of Oxford) to help define some concepts of monotone co-design theory.

We thank all the students at ETH Zürich who followed the class for being guinea pigs and giving constructive feedback.

We thank all the students of the online courses we did on the material for giving us the enthusiasm and the energy to get this done.

### Sponsors

Financial support obtained from ETH Zürich and the Swiss National Science Foundation through the NCCR Automation.

So, in a sense, this book is sponsored by Switzerland.

*Switzerland: come for the chocolate,  
stay for the direct democracy.*



[myswitzerland.com](https://myswitzerland.com)

**ETH** zürich



Swiss National  
Science Foundation



NCCR  
Automation



## 2. Putting things together

In this chapter we discuss the various types of “compositions” we find in applications. The basic idea is thinking about *recipes* that produce something given a list of ingredients. We can ask many things about these recipes: does the order of ingredients matter? Can we go from results back to ingredients? Recipes can also be chained. And, we can think about meta-recipes that have other recipes as ingredients or results.

2.1 Stacking blocks . . . . .	11
2.2 Mixing colors . . . . .	14
2.3 Recipes as ingredients . . . . .	16
2.4 Commutativity and associativity . . . . .	17
2.5 Composing recipes . . . . .	20
2.6 Isomorphisms . . . . .	21

Oct. 24, 1961

G. K. CHRISTIANSEN  
TOY BUILDING BRICK

3,005,282

Filed July 28, 1958

2 Sheets-Sheet 1

FIG. 1.

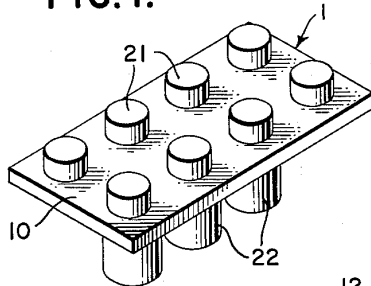


FIG. 2.

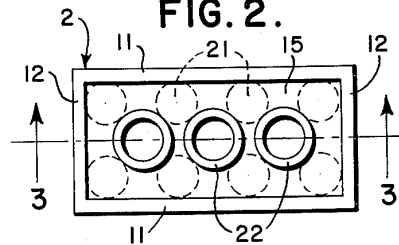


FIG. 3.

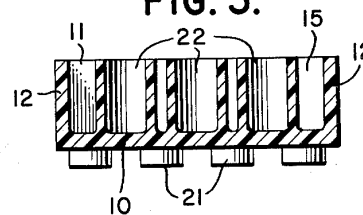


FIG. 4.

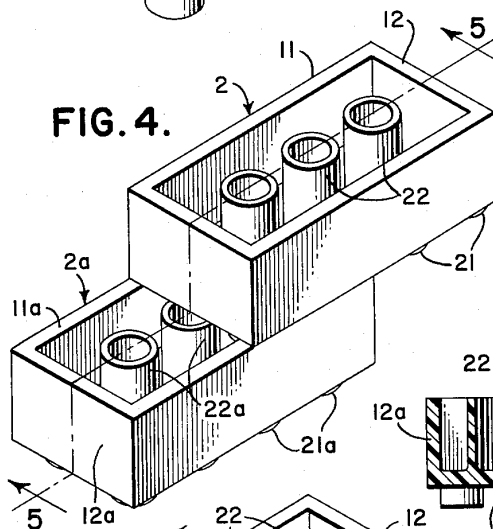


FIG. 5.

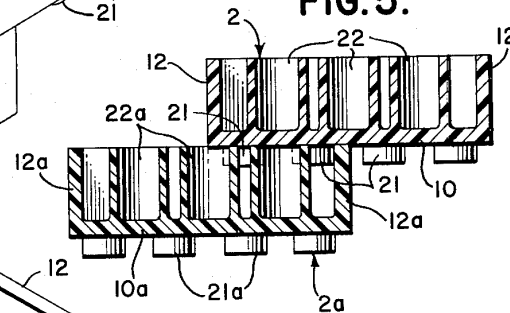
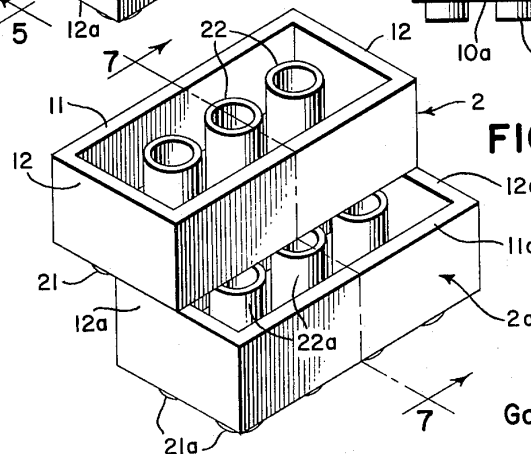


FIG. 6.



INVENTOR

Godtfred Kirk Christiansen

BY  
*Stevens, David, Muller & Mosher*  
ATTORNEYS

Figure 1.: The 1961 Lego patent.



## 2.1. Stacking blocks

The first encounter children have with composition is with toy blocks like Lego. It is a coincidence that there is a *lego* in *intellego* (as explained in Section 1.1) ; the *lego* in Lego is a contraction from Danish *leg godt*, which means *to play well*.

Legos are compositional in this sense: when you put together two blocks, you can treat the ensemble as one block for the purpose of composing it with other blocks.

We are going to use the following graphical notation to talk about composition. We draw a black bar, and we write the *ingredients* at the top, and the *results* at the bottom.

$$\begin{array}{c} \text{ingredient} \quad \text{ingredient} \quad \text{ingredient} \\ \hline \text{result} \quad \text{result} \end{array} . \quad (1)$$

Note that the order of the ingredients matters. For instance, we can have the following recipes for the composition of red and white bricks. We scan the list of ingredients from left to right and then place the bricks on top of what is already on the table.

Composing red and white produces a red-white brick:

$$\begin{array}{c} \text{red brick} \quad \text{white brick} \\ \hline \text{red-white brick} \end{array} . \quad (2)$$

Composing white and red produces a white-red brick:

$$\begin{array}{c} \text{white brick} \quad \text{red brick} \\ \hline \text{white-red brick} \end{array} . \quad (3)$$

We can compose more than one brick. For example, red, white, blue, make a red-white-blue brick:

$$\begin{array}{c} \text{red brick} \quad \text{white brick} \quad \text{blue brick} \\ \hline \text{red-white-blue brick} \end{array} . \quad (4)$$

In Lego, we can also decompose. If we have a red-white-blue brick, we can also recover the single bricks:

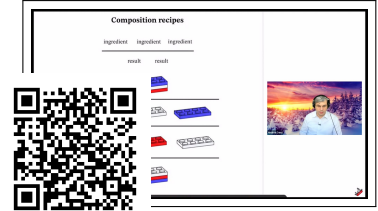
$$\begin{array}{c} \text{red-white-blue brick} \\ \hline \text{red brick} \quad \text{white brick} \quad \text{blue brick} \end{array} . \quad (5)$$

If you have 3 bricks on a table, you can also permute them:

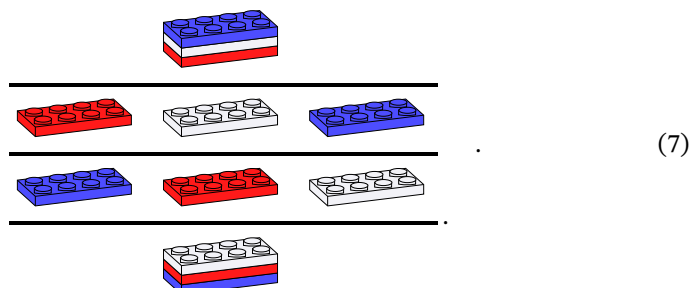
$$\begin{array}{c} \text{red brick} \quad \text{white brick} \quad \text{blue brick} \\ \hline \text{blue brick} \quad \text{red brick} \quad \text{white brick} \end{array} . \quad (6)$$

Consequently, if you have a red-white-blue brick, you can disassemble, permute,

Watch: *Composition* (17 minutes).



and reassemble to obtain a blue-white-red:

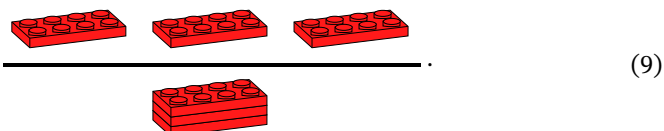


The aforementioned recipe contains several concrete steps to go from the initial ingredient to the final result. If we do not care about the detailed steps, we can summarize the recipe as follows, by eliding the intermediate steps and only remember the ingredient and the results:

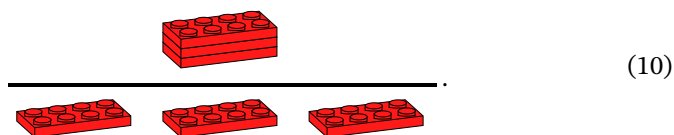


Alternatively, you can think of (8) as the statement of a theorem, and of (7) as the proof of the theorem.

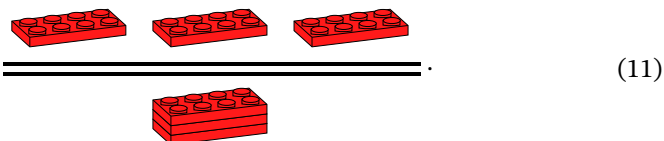
Sometimes we want to think about the transformations that are reversible. For example, we can assemble 3 red bricks into a red-red-red brick:



We can also do the opposite:



To describe the bi-directionality, we use a double line:



The flat pieces of Lego we have looked are actually one third shorter than a “regular” piece:

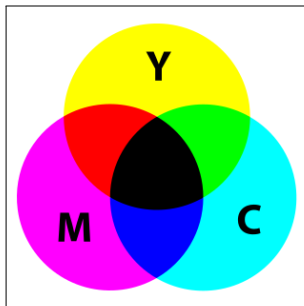


What is the relation between a red-red-red assembly and a full red brick? One point of view that will be very useful is thinking in terms of “substitution”: if I have one of those, can I use it as if I had the other? Lego bricks are very strong when assembled: a red-red-red assembly can certainly substitute a regular brick

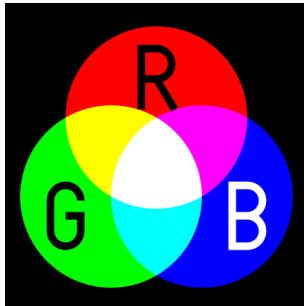


in terms of structural functionality. Therefore, given a red-red-red we can treat it as a full block, but not vice versa:


$$\frac{\text{red-red-red}}{\text{red-red-red}} = \text{red-red-red} \quad (13)$$



(a) Subtractive composition

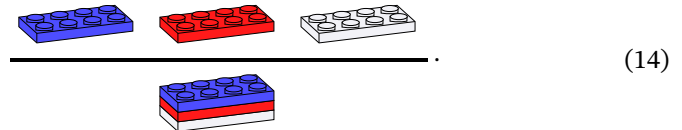


(b) Additive composition

Figure 2.: Additive vs subtractive composition

## 2.2. Mixing colors

We now look at how we can compose colors. In Denmark there is a small group of **Lego purists**: they are only able to conceive of Lego assemblies where all bricks have the same color. For them, a blue, red, white brick, make a block of a color they call *horrible*:



If you ask a color purist, they will tell you that red and red make red:



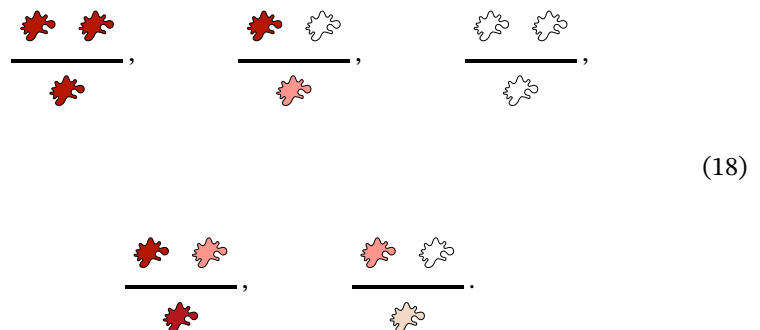
Furthermore, white and white make white:



However, white and red make *horrible*:



We can think of many other ways to compose colors. For example, we can think of formalizing what happens when you **mix paint**. Red and white in equal measure give pink. By mixing and mixing we can obtain all the shades that go from red to white:



Colors on a monitor mix in an **additive** way. Two dark reds give a brighter red.

Red and white remains white:

$$\begin{array}{cc}
 \begin{array}{c} \text{Red} \quad \text{Red} \\ \hline \text{Red} \end{array}, & \begin{array}{c} \text{Red} \quad \text{White} \\ \hline \text{White} \end{array}, \\
 \begin{array}{c} \text{White} \quad \text{White} \\ \hline \text{White} \end{array}, & \begin{array}{c} \text{Red} \quad \text{Red} \\ \hline \text{Red} \end{array}.
 \end{array} \quad (19)$$

Green, red, blue additively make white:

$$\begin{array}{c}
 \text{Red} \quad \text{Green} \quad \text{Blue} \\
 \hline
 \text{White}
 \end{array}. \quad (20)$$

A different way to compose colors is by using the **subtractive** rules in the CMY (cyan, magenta, yellow) color space. These rules formalize the physical process of offset printing: we produce colors by putting pigments that block the other colors:

$$\begin{array}{ccc}
 \begin{array}{c} \text{Red} \quad \text{Red} \\ \hline \text{Red} \end{array}, & \begin{array}{c} \text{Red} \quad \text{White} \\ \hline \text{Red} \end{array}, & \begin{array}{c} \text{White} \quad \text{White} \\ \hline \text{White} \end{array}.
 \end{array} \quad (21)$$

This is how you produce red, blue, green from CMY:

$$\begin{array}{ccc}
 \begin{array}{c} \text{Cyan} \quad \text{Magenta} \\ \hline \text{Blue} \end{array}, & \begin{array}{c} \text{Cyan} \quad \text{Yellow} \\ \hline \text{Green} \end{array}, & \begin{array}{c} \text{Magenta} \quad \text{Yellow} \\ \hline \text{Red} \end{array}.
 \end{array} \quad (22)$$

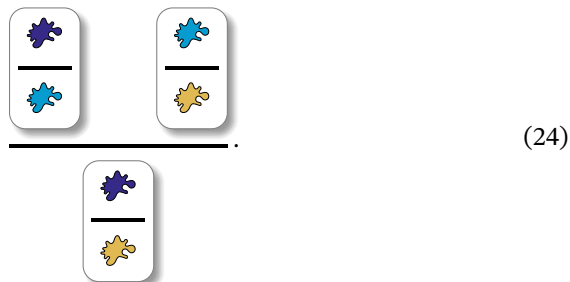
Finally, we can think of a **paint-over-it** composition rule: the first color is replaced by the second:

$$\begin{array}{ccc}
 \begin{array}{c} \text{Red} \quad \text{Red} \\ \hline \text{Red} \end{array}, & \begin{array}{c} \text{Red} \quad \text{Pink} \\ \hline \text{Pink} \end{array}, & \begin{array}{c} \text{Red} \quad \text{White} \\ \hline \text{White} \end{array}.
 \end{array} \quad (23)$$

## 2.3. Recipes as ingredients

We can think at a higher level, by having recipes as ingredients.

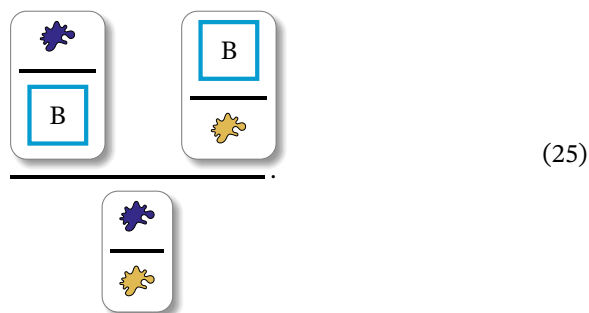
For example, the following shows that if a dark blue stain gives you a light blue stain, and a light blue stain gives you an orange stain, you can produce an orange stain from a dark blue stain:



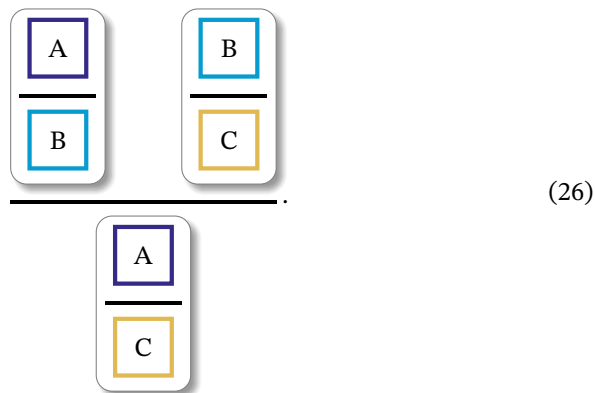
Note that to activate the meta-recipe above, no windsor stain was needed. In fact, for the above to be valid, it is not even necessary to postulate that windsor stains exist.

We can do **abstraction** by replacing some ingredients with *placeholders*. When we write a recipe with placeholders, we mean that the recipe is valid whatever is put in the placeholders, with the constraint that if two placeholders are of a similar color should hold the same thing.

For example, if a windsor stain gives me a B, and B gives me a rob roy stain, then a windsor stain gives me a rob roy stain, no matter what B is:



We can abstract further by saying that: if A gives me B, and B gives me C, then A gives me C:



## 2.4. Commutativity and associativity

With the power of abstraction we can talk about properties of the rules themselves.

For example, we can define *commutativity* as follows. A composition operation is commutative if getting a C from A and B holds if and only if B and A also give a C:

$$\begin{array}{c}
 \boxed{A} \quad \boxed{B} \\
 \hline
 \boxed{C}
 \end{array}
 \quad \equiv \quad
 \begin{array}{c}
 \boxed{B} \quad \boxed{A} \\
 \hline
 \boxed{C}
 \end{array}
 \quad . \tag{27}$$

For associativity, we want to say that, given three things A, B, C, composing A with B and then the result with C is the same thing as composing A with the result of B and C.

$$\begin{array}{c}
 \boxed{A} \quad \boxed{B} \quad \boxed{C} \\
 \hline
 \dots \quad \boxed{C} \\
 \hline
 \boxed{\phantom{C}}
 \end{array}
 \quad \equiv \quad
 \begin{array}{c}
 \boxed{A} \quad \boxed{B} \quad \boxed{C} \\
 \hline
 \boxed{A} \quad \dots \\
 \hline
 \boxed{\phantom{C}}
 \end{array}
 \quad . \tag{28}$$

It is easy to see that this is valid for Lego composition.

$$\begin{array}{c}
 \text{blue} \quad \text{red} \quad \text{white} \\
 \hline
 \text{red+blue} \quad \text{white} \\
 \hline
 \text{red+blue+white}
 \end{array}
 \quad \equiv \quad
 \begin{array}{c}
 \text{red+blue} \quad \text{white} \\
 \hline
 \text{red+blue+white}
 \end{array}
 \quad . \tag{29}$$

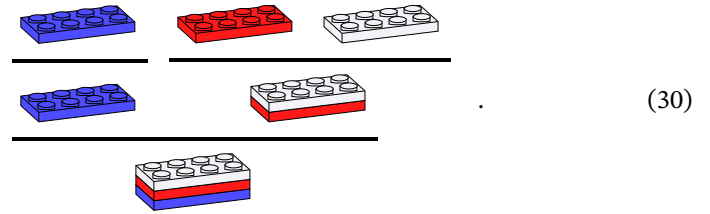



































Table 2.1 shows the properties of the 4 composition rules for composing colors that we described earlier.

The table also notes the presence of a *neutral element* and an *annihilating element*. A neutral element, is an element which, when composed with another element, does not change the original color. Using the additive composition rule, for instance, this element is *black*. On the other hand, when considering the lego purists composition rule no neutral element can be found (indeed, composing with any element will result in a color change).

Table 2.1.: Properties of color composition rules

	Lego purist	Mixing paint	Apply paint	Additive	Subtractive
	<div><div></div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div></div>
Commutative?	✓	✓	✗	✓	✓
Associative?	✓	✗	✓	✓	✓
Annihilative element?	<div><div></div><div></div></div>	✗	<div><div></div><div></div></div>	<div><div></div><div></div></div>	<div><div></div><div></div></div>
Neutral element?	✗	✗	✗	<div><div></div><div></div></div>	<div><div></div><div></div></div>

## 2.5. Composing recipes

We can also compose recipes themselves.

For example, imagine that in our analysis of Lego composition we decompose its color from the shape. Each element is now described by a color and a shape:

$$\begin{array}{c} \text{red brick} \\ \hline \text{red shape, grey brick} \end{array} \quad (31)$$

We can now define composition of color-shape pairs by composing the Lego-purist rule for colors with a color-neutral shape composition rule.

For example, given two pairs

$$\text{red shape, grey brick} \quad \text{and} \quad \text{blue shape, grey brick}, \quad (32)$$

we can find what their composition is by looking at what happens when we compose the components:

$$\begin{array}{c} \begin{array}{|c|} \hline \text{red shape} \quad \text{blue shape} \\ \hline \text{grey brick} \end{array} \quad \begin{array}{|c|} \hline \text{grey brick} \quad \text{grey brick} \\ \hline \text{grey brick} \end{array} \\ \hline \begin{array}{|c|} \hline \text{red shape, grey brick} \quad \text{blue shape, grey brick} \\ \hline \text{grey brick, grey brick} \end{array} \end{array} \quad (33)$$

We can generalize this as follows:

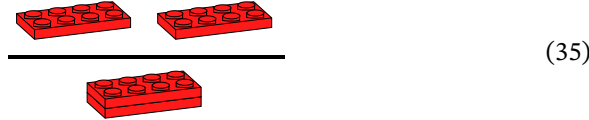
$$\begin{array}{c} \begin{array}{|c|} \hline \text{red square} \quad \text{blue square} \\ \hline \text{black square} \end{array} \quad \begin{array}{|c|} \hline \text{yellow dashed circle} \quad \text{green dashed circle} \\ \hline \text{pink dashed circle} \end{array} \\ \hline \begin{array}{|c|} \hline \text{red square, yellow dashed circle} \quad \text{blue square, green dashed circle} \\ \hline \text{black square, pink dashed circle} \end{array} \end{array} \quad (34)$$



## 2.6. Isomorphisms

Do you know the game “spot the 5 differences”? In this book we are going to play the opposite game, which is “spot how several different things are the same at some level of abstraction”.

Consider two Lego worlds in which all colors are red or all colors are white:



and

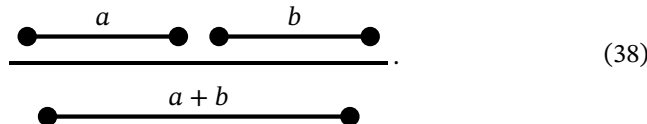


We could create Lego theories for each of the worlds. Although they would describe different worlds, the theories would be *isomorphic*.

If we confine ourselves with composing Lego blocks with the same section, then all it counts is the height of the stacks. The equations above are saying  $1 + 1 = 2$ :

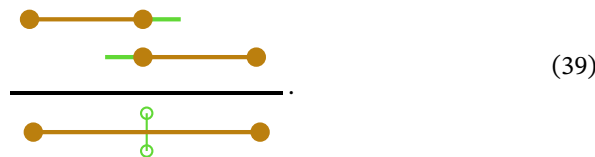
$$\frac{1 \quad 1}{2} . \quad (37)$$

If we are dealing with addition, then there are many other things that follow the same rules. For example, we might look at composing two pieces of rope. If we have a piece of rope of length  $a$  and one of length  $b$ , you can tie them together to get a rope of length  $a + b$ :

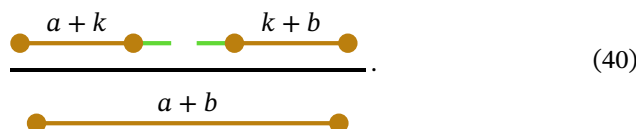


The algebra of ropes captures the algebra of bricks: the bricks are a special case because they have integer height, while ropes can be of any length.

We want to show you a rope trick. Suppose that we want to be more precise than (38) to describe the process of composing ropes, by keeping track of the extra rope that is needed to make a knot:



One first attempt would be to call  $k$  the extra rope for the knot, and have rules like (40): from  $a + k$  and  $k + b$  we obtain a piece of rope of  $a + b$ :



This is fine but not elegant. If you want to compose further, you need to introduce

a notion of subtraction:

$$\begin{array}{c}
 \text{---} a + b \text{---} \quad \text{---} k + c \text{---} \\
 \hline
 \text{---} a + b + c - k \text{---}
 \end{array}
 \quad (41)$$

A more elegant way is the following: consider only ropes of the form  $k + a + k$ , so that we can account for the rope needed for the knots at either ends:

$$\begin{array}{c}
 \text{---} k + a + k \text{---} \quad \text{---} k + b + k \text{---} \\
 \hline
 \text{---} k + a + b + k \text{---}
 \end{array}
 \quad (42)$$

Now when we compose, the 2  $k$ s on the inside elide, and we are left with 2  $k$ s at either end, ready to be knotted with other pieces of rope. Notice that all ropes so created have the 2 extra  $k$ s. We can just remove them from the notation. We obtain new rules for ropes that take into account the knots:

$$\begin{array}{c}
 \text{---} A \text{---} \quad \text{---} B \text{---} \\
 \hline
 \text{---} A + B \text{---}
 \end{array}
 \quad (43)$$

And here's the magic trick: if we don't take into account the knot materials we have the simple rule (38); if we do take into account the knot materials, *for any arbitrary length  $k$* , we obtain (43) which is exactly the same as (38).

**Exercise 1.** Explain the trick: Where did the extra material go?

*See solution on page 31.*



## 3. Exercises

This chapter describes the exercises part of the text.

There are three types of exercises:

- ▷ Solved exercises - with solution at the end of the chapter.
- ▷ Graded Exercises - with solutions omitted from the text in the public version. The solutions are present in a special “instructors edition” available on request.
- ▷ Code exercises, autograded.

3.1 Introduction . . . . .	24
3.2 Code exercises - Overview . . . . .	25
3.3 Exercise tutorial . . . . .	28
3.4 Turning in exercises . . . . .	29

### 3.1. Introduction

This book contains three types of exercises:

- ▷ Solved theoretical exercises, such as Exercise 2 below. These are solved at the end of each part.
- ▷ Graded theoretical exercises, such as Graded Exercise A.1 below. You will not find the solutions in the book.
- ▷ Graded code exercises, such as Code Exercise A.1 below. These are autograded and the solution is not publicly available.

**Remark 3.1. Instructors:** if you want to teach from this book, please get in touch with us. We can provide you with an instructor’s version that contains all solutions to the theoretical exercises.

**Exercise 2.** Can you solve this riddle?

*Thirty white horses on a red hill,  
First they champ,  
Then they stamp,  
Then they stand still.*

What are they?

*See solution on page 31.*

#### Graded exercise A.1

This is a graded exercise.

#### Code exercise A.1

This is a code exercise.

You will have the chance to hand in the graded and code exercises, following the procedure described in Section 3.4.

## 3.2. Overview

### Prerequisites

You will need the following skills:

- ▷ Python programming. If you want to learn Python, we suggest you look at EdX's courses.
- ▷ Git / GitHub usage. If you want to learn Git and GitHub, go to [try.github.io](https://try.github.io).

Additionally, it might be useful to have some idea of what Docker is. In theory, you don't need to know anything about Docker, if everything goes well in our autograding scripts.

### Option 1: Using GitHub Classroom

*If you are following a class where the instructors are using GitHub Classroom, this is the section for you. Otherwise, jump to the section below about “independent use”.*

#### Sign up on GitHub

Sign up on GitHub if you haven't already.

#### Get invite link and accept assignment

GitHub Classroom is a product that helps coordinate assignments for large classes.

If you are enrolled in a class, your instructors will send you an invitation link to join the classroom.

If you are not enrolled in an official class, then you can use this invitation link:

<https://classroom.github.com/a/IMtOmeOH>

In any case, do not pass invite links around: there are different “classrooms”. Do not use old links because the classroom is refreshed after each season.

#### Clone the repository

Once you go through the procedure, you will have created a repository

<https://github.com/INSTRUCTORS-term-exercises-YOU>

where `INSTRUCTORS` is the GitHub organization for the instructors, `term` is the current term (e.g., `spring2022`), and `YOU` is your username.

Clone the repository just created and jump into the directory.

You should see the content as in Fig. 1.

In the following we list two options for you to be able to use the ACT4E interfaces when you code.

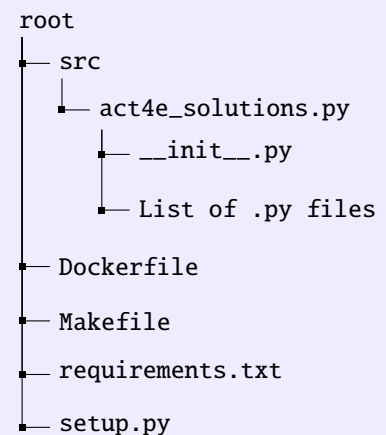
### Option 2: Independent use without GitHub Classroom

*If you are not following a class where the instructors are using GitHub Classroom, this is the section for you. Otherwise, jump to Section 3.2.*

In this section we assume that you are not part of a class.

In that case, you can fork directly the repository

Something incorrect or unclear or missing? Report issues on GitHub by clicking [here](#).



**Figure 1.:** Original content of the exercise template repository.

<https://github.com/ACT4E/ACT4E-exercises-template>

Note that there might be different branches to choose from. The instructions in this book correspond to the branch `alphubel-prod`.

You can do the exercises on your own by using this repository.

## Using a Visual Studio Code development container

The exercises are meant to be solved in a Visual Studio Code “development container”. This allows the experience to work on Windows, Mac, and Linux.

The way this works is that Visual Studio Code will create a Docker environment you can work in. You will be able to see and edit files as usual, but the code will run in the container.

Please follow the instructions in the `README.md` file in the repository.

They will guide you through the installation of Docker, Visual Studio Code, and the development container.

Note that:

- ▷ You can still *edit* the files with an external editor. You do not have to use VS Code as an editor. However, you need to run the code from inside the development container.
- ▷ Even if you are an expert, do not try to run the code from outside the container—there are some Python dependencies that are only available for Linux (arm/amd) and Python 3.10.

## Updating the exercises packages

The exercises themselves and the interfaces are stored in the repository

<https://github.com/ACT4E/ACT4E-exercises.git>,

and it is published as the package `ACT4E-exercises`.

Sometimes there will be a need to update this package, as we improve it or fix bugs.

There are two ways to do this.

1. You can update the package alone by running the following command **inside the container**:

```
python -m pip install -U ACT4E-exercises
```

2. You can use the Visual Studio Code command *Rebuild development container without cache*.

## Without Visual Studio Code, from command line

If you cannot use Visual Studio Code, you can still edit the files from your favorite editor and use Docker alone to check the results.

Whenever you will be instructed to run

```
act4e-check EXERCISENAME
```

You should run the following command instead:

```
make docker-check-EXERCISENAME
```

## Using PyCharm + Docker environment

You can configure PyCharm to obtain the same result as Visual Studio Code, albeit with a bit more configuration.

The idea is that we are going to configure a “Python interpreter” in PyCharm that corresponds to the interpreter *inside* a Docker image.

First, go to the root folder and type

```
make build
```

This will create an image called `act4e-image`. Make sure you can run it using:

```
docker run -it act4e-image python --version
```

Which should give a result like:

```
Python 3.10.2
```

Create a new PyCharm project in the root folder.

In Preferences - Python Interpreter - find the gear and choose “add interpreter”.

Choose the “Docker” type. Put `act4e-image` as the image.

Enable this interpreter for the project.

Now you should be able to open any of the template files and all symbols should be resolvable. For example, you should be able to click `act4e_interfaces` and explore the contents of the package.



### 3.3. Exercise tutorial

During the exercises, you are going to build a Python library to build and manipulate most of the concepts in the book.

The exercises will give you the *interface* of an *abstract class* to be generated.

The interfaces are defined in the package `act4e_interfaces` which is in the repository `ACT4E/ACT4E-interfaces`.

For example, Listing 1 shows an interface `SimpleIntro` to be implemented.

**Listing 1:** The `SimpleIntro` interface.

```
class SimpleIntro(ABC):
    @abstractmethod
    def sum(self, a: int, b: int) -> int:
        """Returns the sum of two integers."""
```

We are going to solve this exercise together.

We assume that you have checked out the repository as explained in the setup section.

We can start by checking if this is a valid solution.

In the development environment, type this:

```
act4e-check TestSimpleIntro
```

Here `TestSimpleIntro` is the code name for the exercise. You can check other exercises by changing the name. Each exercise name is of the form `TestINTERFACE` where `INTERFACE` is the name of the Python class providing the interface.

The command above will fail; it will complain saying that something is not implemented.

Go to the file called `intro.py` in the `src/act4e_solutions` directory. This file contains the skeleton for the class implementation, as in Listing 2.

**Listing 2**

```
import act4e_interfaces as I

class SolSimpleIntro(I.SimpleIntro):
    def sum(self, a: int, b: int) -> int:
        raise NotImplementedError()
```

Now modify the file to implement the right computation, as in Listing 3.

**Listing 3**

```
import act4e_interfaces as I

class SolSimpleIntro(I.SimpleIntro):
    def sum(self, a: int, b: int) -> int:
        return a + b
```

Trying again this:

```
act4e-check TestSimpleIntro
```

You will see a success result.



## 3.4. Turning in exercises

Via GitHub Classroom you can easily turn in your exercises.

Each graded exercise is characterized by a name (for instance `TestFiniteSetRepresentation`).

To hand in an exercise, simply commit it with the name as the commit message, and push it to your solutions branch.

In the case of code exercises, the push will automatically result in an evaluation, which will assign you points (running the same tests as the ones you can run before pushing).



## Solutions to selected exercises

**Solution of Exercise 1.** Note that (38) describes composition for ropes of any arbitrary size, while (43) describes compositions for ropes whose length can be written as  $k + a + k$ , hence with a minimum size of  $2k$ . Therefore, the rope

$$\text{---} \overset{1}{\text{---}} \text{---} \quad (1)$$

in the first theory describes a physical rope of length 1, while the rope

$$\text{---} \overset{1}{\text{---}} \text{---} \quad (2)$$

in the second theory describes a rope of length  $1 + 2k$ . They are different theories that happen to have isomorphic rules. Note that the rope  $k + k$  acts just like the identity 0. If you connect  $k + a + k$  to  $k + k$ , you obtain  $k + a + k$ , for all values of  $a$ .

**Solution of Exercise 2.** *Teeth*, Gollum answered correctly.



# PART B.SETS, FUNCTIONS, RELATIONS



---

<b>4. Sets and functions</b>	<b>35</b>
<b>5. Relations</b>	<b>75</b>

---

Augusta Raurica is a Roman archaeological site in Switzerland, located on the south bank of the Rhine river (near Basel). It was founded by Lucius Munatius Plancus around 44 B.C.







## 4. Sets and functions

Sets and functions are fundamental notions in mathematics. In this chapter we give an informal treatment of those set-theoretic notions which are important for the purposes of this book, while avoiding more complicated aspects. The material in this section should be mostly familiar to anyone with some training in engineering, computer science, a natural science, mathematics, *etc.* We suggest nonetheless reading through: we set some conventions, and some contents may be new or rendered from a new perspective.

<b>4.1 Logical preliminaries . . . . .</b>	<b>36</b>
<b>4.2 Sets . . . . .</b>	<b>39</b>
<b>4.3 New sets from old . . . . .</b>	<b>44</b>
<b>4.4 Code exercises - Type theory . . .</b>	<b>49</b>
<b>4.5 Code exercises - Representing sets</b>	<b>51</b>
<b>4.6 Code exercises - FiniteSetProperties . . . . .</b>	<b>57</b>
<b>4.7 Code exercises - Power set . . . .</b>	<b>58</b>
<b>4.8 Code exercises - Set products . .</b>	<b>59</b>
<b>4.9 Code exercises - Disjoint union .</b>	<b>61</b>
<b>4.10 Functions . . . . .</b>	<b>62</b>
<b>4.11 Code exercises - Mappings . . . .</b>	<b>70</b>
<b>4.12 Code exercises - FiniteMapOperations . . . . .</b>	<b>71</b>
<b>4.13 Further constructions . . . . .</b>	<b>72</b>

## 4.1. Logical preliminaries

We assume the reader to have some familiarity with elementary logical concepts and notation, in the way that they are typically used for reasoning and writing proofs in undergraduate mathematics. We recall some basic notions, giving a “naive” treatment intended as a rudimentary foundation and to fix our notation. A more formal treatment is outside the scope of this text.

### Deduction

The building blocks for reasoning mathematically are logical statements (sometimes called propositions, assertions, logical formulas, *etc.*) which, in principle, may be evaluated to be either true or false, depending on the situation and the assumptions made. In particular, a statement might depend on variables, and the truth or falsity of the statement might vary depending on how these variables are evaluated.

To make mathematical proofs, we start with some assumptions (statements which we take, for the sake of our argument, to be true), and then we apply rules of reasoning, often called *inference rules*, which allow us to deduce new statements from these. These new statements are the “conclusions”. This process is often iterated many times in order to arrive at a statement that we seek to prove.

If we can infer a statement  $Q$  when given a statement  $P$ , we write

$$\frac{P}{Q}. \quad (1)$$

Other common ways of phrasing this are ‘ $P$  implies  $Q$ ’ or ‘ $Q$  follows from  $P$ ’.

If  $Q$  can be inferred from statements  $P_1, \dots, P_n$ , then we write

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{Q}. \quad (2)$$

This notation also allows for combining multiple steps of inference, leading to a “proof tree” such as

$$\frac{\frac{P_1}{Q_1} \quad \frac{P_2 \quad P_3}{Q_2}}{R}. \quad (3)$$

When we write

$$\frac{P}{\equiv Q}, \quad (4)$$

this means that  $Q$  may be inferred from  $P$  and vice versa.

If we want to say that a statement  $Q$  is simply true – that it can be deduced from zero assumptions – then we write

$$\frac{}{Q}. \quad (5)$$



## Connectives

Logical connectives are operations that allow us to construct new logical statements from given ones.

Two familiar logical connectives are “and” and “or”, usually denoted in infix notation by the symbols  $\wedge$  and  $\vee$ , respectively. We think of each of these as a function that takes two logical statements as its arguments, and returns a new logical statement. If  $P$  and  $Q$  are logical statements, then

$$P \wedge Q \quad (6)$$

is the new logical statement which is true precisely when both  $P$  and  $Q$  are true, and otherwise is false. And

$$P \vee Q \quad (7)$$

is the logical statement that is true precisely when either  $P$  or  $Q$ , or both, are true.

A logical operation that only takes on argument is *negation*: if  $P$  is a statement, its negation

$$\neg P \quad (8)$$

is the statement that is true if and only if  $P$  is false.

Furthermore, it is useful to include in our logical language the symbols

$$\top \quad \text{and} \quad \perp \quad (9)$$

for “true” and “false”, respectively, which we think of as connectives taking zero arguments.

## Logical calculus

Various rules, for example expressed using equations, relate the different logical connectives to each other and dictate how to calculate with them. For instance, rules such as

$$(P \wedge Q) \vee R = (P \vee R) \wedge (Q \vee R) \quad (10)$$

or

$$\neg(P \wedge Q) = (\neg P \vee \neg Q) \quad (11)$$

or

$$\neg(\neg P) = P. \quad (12)$$

Note that we use parentheses above to make clear in which order to evaluate a compound logical formula. Conventions about how strongly different logical operations “bind” allow us to use less parentheses. For example, logical negation is taken to bind at the strongest level, so we can write  $\neg P$  instead of  $\neg(P)$  in logical formulas, without introducing ambiguity.

## Variables and quantifiers

Beyond connectives, our logical language also includes *variables*, as well as the so-called *quantifier* symbols  $\exists$  and  $\forall$ , read “there exists” and “for all”, respectively.

The quantifiers may be viewed as operations that have two arguments: the first argument is a variable, say  $x$ , and the second argument is a logical statement that might depend on  $x$ . The result is again a logical statement. For example, given a

statement  $P(x)$  possibly depending on  $x$ ,

$$\exists x P(x) \quad (13)$$

denotes the statement “there exists an  $x$  such that the statement  $P(x)$  is true”. Similarly,

$$\forall x P(x) \quad (14)$$

is the statement “for all  $x$ , the statement  $P(x)$  is true”.

We will use the notation  $\exists!x P(x)$  to say “there exists precisely one  $x$  such that  $P(x)$  is true”.

### Implication as a connective

In addition to expressing the fact that ‘ $P$  implies  $Q$ ’ using the notation

$$\frac{P}{Q}, \quad (15)$$

we can also express the statement ‘ $P$  implies  $Q$ ’ using implication as a logical connective. We use the notation ‘ $\Rightarrow$ ’ for this connective, and we think of it as a function of two variables: given statements  $P$  and  $Q$ , it spits out the new statement  $P \Rightarrow Q$ . It may be expressed (or defined) as

$$(P \Rightarrow Q) = (\neg P \vee Q), \quad (16)$$

depending on whether one wishes to take  $\Rightarrow$  as a primitive connective or define it as a compound connective via (16). This is a matter of convention.

The relationship between (15) and “ $P \Rightarrow Q$ ” is that (15) says “the statement  $P \Rightarrow Q$  is true”; another way to say this would be to write

$$\overline{P \Rightarrow Q}. \quad (17)$$

Another connective that is useful (and commonly used) is called *equivalence*. We use the symbol ‘ $\Leftrightarrow$ ’ for it, and define it by

$$P \Leftrightarrow Q := (P \Rightarrow Q) \wedge (Q \Rightarrow P). \quad (18)$$

When the statement  $P \Leftrightarrow Q$  is true we say ‘ $P$  and  $Q$  are *equivalent*’ or that ‘ $P$  is true *if and only if*  $Q$  is true’. Also, this is the same as saying

$$\frac{P}{Q}, \quad (19)$$



however one basic (and sometimes confusing) point is that sets can be elements of other sets.

For example, we might consider the set

$$\{\text{🍏}, \{\text{🍇}\}, \{\text{🍏}, \text{🥕}, \text{🍇}, \text{🍺}\}\}, \quad (21)$$

which has three elements: namely  $\text{🍏}$ ,  $\{\text{🍇}\}$ , and  $\{\text{🍏}, \text{🥕}, \text{🍇}, \text{🍺}\}$ . Each of  $\text{🍏}$ ,  $\{\text{🍇}\}$ , and  $\{\text{🍏}, \text{🥕}, \text{🍇}, \text{🍺}\}$  is a “thing”, and these three things happen to be assembled together in the set (21). Based on our conventions (curly brackets specify sets), both  $\{\text{🍇}\}$  and  $\{\text{🍏}, \text{🥕}, \text{🍇}, \text{🍺}\}$  are sets (and for our purposes, we do not need to say rigorously what kind of thing  $\text{🍏}$  might be).

From the above discussion it is hopefully now clear that, given sets **A**, **B**, and **C**, say, we can build for example a new set

$$\{\mathbf{A}, \mathbf{B}, \mathbf{C}\} \quad (22)$$

whose elements are **A**, **B**, and **C**. It is hopefully also clear for example that  $\emptyset$  and  $\{\emptyset\}$  are different sets (this is sometimes a confusing case!). The former is the empty set (it has no elements), while the latter is a set which has precisely one element, and that element happens to be the set  $\emptyset$ .

## Equality

Given two sets, we say they are equal if and only if “they have the same elements”. For example, we have seen that  $\{\text{🍏}, \text{🥕}, \text{🍇}, \text{🍺}\}$  and  $\{\text{🍺}, \text{🍇}, \text{🍏}, \text{🥕}\}$  and  $\{\text{🍏}, \text{🥕}, \text{🍺}, \text{🍇}\}$  are all equal as sets, because their elements are the same. In particular, this example shows that a given set might have many names or symbolic representations.

From one perspective, the criterion for knowing when sets are equal reduces to knowing when elements (or “things”) are equal. If **A** and **B** are sets, then to check if  $\mathbf{A} = \mathbf{B}$ , we need to check if respective elements of **A** and **B** are equal.

From another perspective, the question of equality of sets can be expressed in terms of membership: **A** and **B** are equal if and only if the statements  $x \in \mathbf{A}$  and  $x \in \mathbf{B}$  are logically equivalent ( $x$  is a variable that can be instantiated with the elements of **A** or **B**):

$$\begin{array}{c} \mathbf{A} = \mathbf{B} \\ \hline \hline x \in \mathbf{A} \\ \hline \hline x \in \mathbf{B} \end{array} . \quad (23)$$

Note that there are two levels of double lines. Because the first set of double lines is wider, we read the statement as stating an equivalence between  $\mathbf{A} = \mathbf{B}$  and  $x \in \mathbf{A}$  and  $x \in \mathbf{B}$ .

## Subsets

Consider the set  $\{\text{🍏}, \text{🥕}, \text{🍇}\}$ , and compare it with  $\{\text{🍏}, \text{🥕}, \text{🍇}, \text{🍺}\}$ . Each of the elements of the first set is also an element of the second set; in such a case we say that the first set is a *subset* of or is *included* in the second set. In symbols,

$$\{\text{🍏}, \text{🥕}, \text{🍇}\} \subseteq \{\text{🍏}, \text{🥕}, \text{🍇}, \text{🍺}\}. \quad (24)$$

Generally, given sets  $A$  and  $B$ , the statement  $A \subseteq B$  (that  $A$  is a subset of  $B$ ) is logically equivalent to the statement

$$\frac{x \in A}{x \in B} \quad (25)$$

in sequent form and equivalent to the statement

$$\forall x \in A : x \in B \quad (26)$$

in terms of the “for all” universal quantifier.

Returning to (24), the second set is, on the other hand, *not* included in the first set, since  $\text{🍌}$  is an element of the second set, but not the first. If we say a set  $A$  is “strictly included” in another set  $B$ , then we mean “included in and not equal”; the notation for this is  $A \subset B$ .

The notation for inclusion and strict inclusion of sets is analogous to the notation in the context of numbers for “less than or equals”,  $x \leq y$ , and “strictly less than”,  $x < y$ , respectively.

In general, if we do not use the adjective “strictly”, then “inclusion” means that equality is also possible. In particular, in our terminology it is true that any set  $A$  is included in itself:  $A \subseteq A$ .

Inclusion and equality are related as follows: given sets  $A$  and  $B$ ,

$$\frac{A \subseteq B \quad B \subseteq A}{A = B} \quad (27)$$

Many times, in order to prove a statement of the form  $A = B$  it is a useful strategy to prove the two statements  $A \subseteq B$  and  $B \subseteq A$  each.

With respect to inclusion of sets, the empty set  $\emptyset$  has (once again) some slightly tricky behavior:  $\emptyset$  is a subset of any other set. To see why this makes sense, consider the formulation (26): when  $A$  is the empty set, this statement is always true, since quantifying “for all” over the empty set poses no condition at all.

## Specifying a set via a logical statement

In addition to the “naming the elements” way of specifying a set, many times sets are specified with the help of a logical “statement” or “sentence” which serves as a condition which characterizes its elements.

The idea is this: we start out with some given set  $B$ , and then we consider a statement  $S(x)$  which depends on a variable  $x$ , which we think of as running over the elements of  $B$ . We can then ask: for which elements  $x$  of  $B$  is the statement  $S(x)$  true? These elements form a subset of  $B$ , often denoted

$$\{x \in B \mid S(x)\}. \quad (28)$$

For example, let  $B = \{\text{🍌}, \text{🍇}, \text{🥕}\}$  and consider the statement

$$S(x) = “x \text{ is a fruit}”. \quad (29)$$

Then we can form the set

$$\{x \in B \mid x \text{ is a fruit}\} = \{\text{🍌}, \text{🍇}\}. \quad (30)$$

There is an interesting special case of this way of constructing subsets of a set  $B$ :

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

what if, for a given statement  $S(x)$ , *none* of the elements  $x \in \mathbf{B}$  are such that  $S(x)$  is true? Then the result is the empty set.

As an example, consider the statement

$$S(x) = \text{"}x \text{ is the name of a planet"}. \quad (31)$$

Then, for the set  $\mathbf{B} = \{\text{🍎, 🍇, 🥕}\}$ , the set defined as

$$\{x \in \mathbf{B} \mid x \text{ is the name of a planet}\} \quad (32)$$

is equal to the empty set.

## Logical statements quantified over a set

The above describes how to define a set using a logical statement. Often times we also conversely use a set to formulate a logical statement. For example a statement of the kind “there exists  $x \in \mathbf{A}$ , such that the statement  $P(x)$  is true”. Our notation for this will be

$$\exists x \in \mathbf{A} : P(x). \quad (33)$$

Similarly,

$$\forall x \in \mathbf{A} : P(x). \quad (34)$$

denotes the statement “for all  $x \in \mathbf{A}$ ,  $P(x)$  is true”.

**Remark 4.1** (Do you want to be more formal?). The statements (33) and (34) can be formulated in the formats “ $\exists x Q(x)$ ” and “ $\forall x Q(x)$ ”, respectively, that were introduced in Section 4.1 for the logical symbols  $\exists$  and  $\forall$ :

$$\exists x ((x \in \mathbf{A}) \wedge P(x)), \quad (35)$$

and

$$\forall x ((x \in \mathbf{A}) \Rightarrow P(x)). \quad (36)$$

However, (33) and (34) are easier to read and are common usage.

## Sets we know and love

There are some sets from basic mathematics that we are very familiar with, such as the set  $\mathbb{N}$  of natural numbers<sup>†</sup>

$$\mathbb{N} = \{0, 1, 2, 3, 4, \dots\}, \quad (37)$$

the set  $\mathbb{Z}$  of integers (or “whole numbers”)

$$\mathbb{Z} = \{0, +1, -1, +2, -2, +3, -3, \dots\}, \quad (38)$$

or the rational numbers  $\mathbb{Q}$ , real numbers  $\mathbb{R}$ , and complex numbers  $\mathbb{C}$ . In formal set theory, these sets can actually be quite a nuisance to define rigorously; for our purposes this is unnecessary, and we will just work with these sets as usual.

The *set of booleans* is defined as:

$$\mathbf{Bool} = \{\perp, \top\}, \quad (39)$$

where  $\perp$  is “false” and  $\top$  is “true”.

<sup>†</sup> Whether zero should be included in the definition of the natural numbers is a question of convention, and there is no clear universal agreement on this. We choose to follow the ISO 80000-2 standard [13] that includes zero as part of the natural numbers.

It is useful to define a *singleton set*  $\mathbf{1}$ , which is a set containing only one element:

$$\mathbf{1} := \{\bullet\}. \quad (40)$$

It does not matter what the singleton element  $\bullet$  exactly is or what symbol we use to define it. When we use the singleton set  $\mathbf{1}$ , all we will use about it is that it contains exactly one element.

### 4.3. New sets from old

In this section we recall some elementary ways of constructing new sets from old. The idea of *constructing* new things from old things is one of the main themes of the book.

#### Union and intersection

The union of two sets is the set containing elements from either of the two.

**Definition 4.2** (Union of sets)

Given sets  $A$  and  $B$ , their *union* is a new set, denoted  $A \cup B$ , characterized by

$$\frac{x \in (A \cup B)}{x \in A \vee x \in B}. \quad (41)$$

For example, if  $A = \{\text{🍏}, \text{🍇}\}$  and  $B = \{\text{🍺}\}$ , then

$$A \cup B = \{\text{🍏}, \text{🍇}, \text{🍺}\}. \quad (42)$$

Similarly, the intersection of two sets is the set of elements common to both.

**Definition 4.3** (Intersection of sets)

The *intersection* of sets  $A$  and  $B$ , denoted  $A \cap B$ , is the set characterized by

$$\frac{x \in (A \cap B)}{x \in A \wedge x \in B}. \quad (43)$$

So, if  $A = \{\text{🍏}, \text{🍇}, \text{🥕}, \text{🍺}\}$  and  $B = \{\text{🍺}, \text{🍇}, \text{🇩🇪}\}$ , then

$$A \cap B = \{\text{🍺}, \text{🍇}\}. \quad (44)$$

**Exercise 3.** Prove that union and intersection of sets are associative operations.

*See solution on page 97.*

**Exercise 4.** Prove that union and intersection of sets are commutative operations.

*See solution on page 97.*

#### Powerset

**Definition 4.4** (Power set)

Given a set  $A$ , we can form a new set whose elements are precisely all the subsets of  $A$ . This new set is called the *powerset* of  $A$ ; we denote it by  $\text{Pow } A$ .

For example, if  $A = \{\text{🍏}, \text{🥕}, \text{🍇}\}$ , then its powerset is

$$\text{Pow } A = \{\emptyset, \{\text{🍏}\}, \{\text{🥕}\}, \{\text{🍇}\}, \{\text{🍏}, \text{🥕}\}, \{\text{🥕}, \text{🍇}\}, \{\text{🍏}, \text{🍇}\}, \{\text{🍏}, \text{🥕}, \text{🍇}\}\}. \quad (45)$$

**Exercise 5.** Can you count how many elements the powerset  $\text{Pow } A$  has in the following cases?

1.  $A = \{\text{🍏}\}$ .
2.  $A = \{\text{🍏}, \text{🥕}\}$ .



3.  $A = \{\text{🍎}, \text{🥕}, \text{🍇}\}.$

4.  $A = \emptyset.$

Can you guess a general formula for the size of the powerset of a finite set?

See solution on page 97.

Now suppose we fix a set  $A$  for a moment. Given  $S \in \text{Pow } A$ , the *complement* of  $S$  with respect to  $A$  is

$$A \setminus S = \{x \in A \mid x \notin S\}, \quad (46)$$

which is again an element of  $\text{Pow } A$ . In situations where it is evident which ambient set  $A$  we are working with, the notation  $S^c$  is sometimes used instead of  $A \setminus S$ .

We also note that the operations of union and intersection, when restricted to  $\text{Pow } A$ , again produce elements of  $\text{Pow } A$ . That is, if  $S, T \in \text{Pow } A$ , then  $S \cup T \in \text{Pow } A$ , and similarly  $S \cap T \in \text{Pow } A$ .

The operations  $\cup$ ,  $\cap$ , and  $(-)^c$  on  $\text{Pow } A$  obey various rules which are useful to be familiar with. For example,

$$(S \cap T)^c = (S^c) \cup (T^c). \quad (47)$$

Can you state more such rules? A useful visual aid for such calculations are so-called Venn diagrams (Fig. 2, Fig. 3, Fig. 4).

#### Graded exercise B.1 (DistributingSubsets)

Let  $A$  be a set, and let  $S, T, U \subseteq A$  be subsets. Prove that

$$S \cap (T \cup U) = (S \cap T) \cup (S \cap U). \quad (48)$$

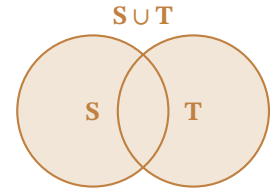


Figure 2.: Venn diagram for union operation.

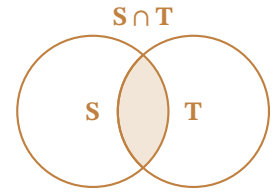


Figure 3.: Venn diagram for intersection operation.

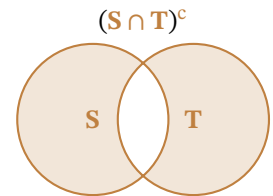


Figure 4.: Venn diagram for complement operation.

## Tuples

A “tuple” is a finite sequence or listing of “things” where their order matters and repetitions are allowed. We use the following notation to denote a tuple of numbers, for example:

$$\langle 3.5, -2, 7, 68 \rangle. \quad (49)$$

Tuples always have a *length*, which can be any natural number. A tuple of length  $n \in \mathbb{N}$  is called an  $n$ -tuple. There is only one possible tuple of length zero; we call it the empty tuple and denote it by  $\langle \rangle$ .

Tuples of length two are often called *ordered pairs* or just *pairs*; tuples of length three are called *triples*; tuples of length four are called *quadruples*, and so on.

We concatenate tuples using the symbol  $\circledast$ :

$$\langle a, b \rangle \circledast \langle c \rangle \circledast \langle \rangle = \langle a, b, c \rangle. \quad (50)$$

The items inside the brackets that indicate a tuple will be called *entries* or *components*. If  $t$  is a tuple of length  $n$ , then  $t[i]$  will refer to the  $i^{\text{th}}$  entry of  $t$ , where  $1 \leq i \leq n$ . So, for example, if

$$t = \langle 2, 9, -1, 3, 6 \rangle, \quad (51)$$

then  $t[1] = 2$ ,  $t[2] = 9$ ,  $t[3] = -1$ , etc.

Typically, we use tuples in situations where we also specify, for each entry of the tuple, a set of which that entry is an element. For example, if  $A = \{\text{🍎}, \text{🥕}, \text{🍇}\}$  and  $B = \{\text{🍷}, \text{🍷}\}$ , then sometimes we will want to specify that  $\langle \text{🍎}, \text{🍷} \rangle$  is a 2-tuple where  $\text{🍎} \in A$  and  $\text{🍷} \in B$ .

## Lists

Our notion of *list* is similar to that of a tuple, except that we require its entries to all be elements of a single specified set or to be all things of a specified type.

We will use the notation

$$[3, 7, 8]_{\mathbb{N}} \quad (52)$$

to denote a list of natural numbers, for example, and

$$[3, 7, 8]_{\mathbb{R}} \quad (53)$$

to denote a list of real numbers. Often it will be clear which type of things we are dealing with, in which case we drop the subscript in our notation and simply write

$$[3, 7, 8]. \quad (54)$$

We concatenate lists using the symbol  $\circ$ :

$$[1, 3] \circ [ ] \circ [5, 7] = [1, 3, 5, 7]. \quad (55)$$

Of course, the type of things we consider need not be numbers. For example, we might work with lists whose entries are from the set  $\mathbf{A} = \{\text{🍏}, \text{🍌}, \text{🥕}, \text{🥞}, \text{🍇}\}$ . Such a list is  $[\text{🍇}, \text{🍏}, \text{🥕}, \text{🍏}]_{\mathbf{A}}$ , for instance.

Similar to tuples, lists can have any length  $n \in \mathbb{N}$ , including zero. The entries of a list  $l$  of length  $n$  will be denoted  $l[i]$ , where  $1 \leq i \leq n$ .

In contrast to tuples, we have a different empty list (zero-length list) for every possible type of list. The empty list of things of a given type  $T$  will be denoted  $[ ]_T$ .

The set of all lists of elements of  $\mathbf{A}$  is written as **List**  $\mathbf{A}$ .

## Cartesian product

### Definition 4.5 (Cartesian product of sets)

Given sets  $\mathbf{A}$  and  $\mathbf{B}$ , their *cartesian product* is a new set – denoted  $\mathbf{A} \times \mathbf{B}$  – whose elements are precisely all possible 2-tuples  $\langle x, y \rangle$  such that the first entry  $x$  is an element of  $\mathbf{A}$  and the second entry  $y$  is an element of  $\mathbf{B}$ .

For example, if  $\mathbf{A} = \{\text{🍏}, \text{🍇}, \text{🥕}\}$  and  $\mathbf{B} = \{\text{🍺}, \text{🇩🇪}\}$ , then

$$\mathbf{A} \times \mathbf{B} = \{\langle \text{🍏}, \text{🍺} \rangle, \langle \text{🍏}, \text{🇩🇪} \rangle, \langle \text{🍇}, \text{🍺} \rangle, \langle \text{🍇}, \text{🇩🇪} \rangle, \langle \text{🥕}, \text{🍺} \rangle, \langle \text{🥕}, \text{🇩🇪} \rangle\}. \quad (56)$$

In the special case where  $\mathbf{A} = \emptyset$  or  $\mathbf{B} = \emptyset$ , then  $\mathbf{A} \times \mathbf{B} = \emptyset$ .

Another way to represent the cartesian product is the following:

$$\begin{array}{c} \text{🍏} \\ \text{🍇} \\ \text{🥕} \\ \mathbf{A} \end{array} \times \begin{array}{c} \text{🍺} \\ \text{🇩🇪} \\ \mathbf{B} \end{array} = \begin{array}{cc} \langle \text{🍏}, \text{🍺} \rangle & \langle \text{🍏}, \text{🇩🇪} \rangle \\ \langle \text{🍇}, \text{🍺} \rangle & \langle \text{🍇}, \text{🇩🇪} \rangle \\ \langle \text{🥕}, \text{🍺} \rangle & \langle \text{🥕}, \text{🇩🇪} \rangle \\ \mathbf{A} \times \mathbf{B} \end{array} \quad (57)$$

**Remark 4.6.** For finite sets  $\mathbf{A}$  and  $\mathbf{B}$ , the size of  $\mathbf{A} \times \mathbf{B}$  is the product (multiplication) of the sizes of  $\mathbf{A}$  and  $\mathbf{B}$ :

$$\text{card}(\mathbf{A} \times \mathbf{B}) = \text{card}(\mathbf{A}) \cdot \text{card}(\mathbf{B}). \quad (58)$$

This is one reason why we think of  $\mathbf{A} \times \mathbf{B}$  as a kind of multiplication of sets.

**Remark 4.7** (Do you want to be more formal?). In formal set theory, 2-tuples (ordered pairs) are often defined by setting

$$\langle x, y \rangle := \{\{x\}, \{x, y\}\}. \quad (59)$$

In this case, the cartesian product is

$$\mathbf{A} \times \mathbf{B} = \{z \in \text{PowPow}(\mathbf{A} \cup \mathbf{B}) \mid \exists x \in \mathbf{A}, \exists y \in \mathbf{B} : z = \langle x, y \rangle\}. \quad (60)$$

Formulas similar to (59) could also be used to define  $n$ -tuples set-theoretically. We will, however, treat tuples as a primitive construction (*i.e.*, without reference to formal set theory), and hence for us also the construction of “cartesian product of sets” is primitive.

## Disjoint union

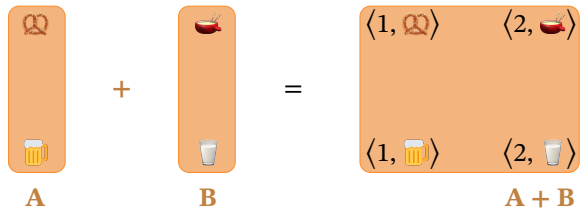
**Definition 4.8** (Disjoint union of sets)

Given sets  $\mathbf{A}$  and  $\mathbf{B}$ , their disjoint union is the set

$$\mathbf{A} + \mathbf{B} := (\{1\} \times \mathbf{A}) \cup (\{2\} \times \mathbf{B}). \quad (61)$$

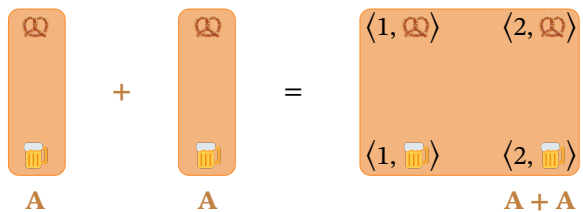
In other words, an element of  $\mathbf{A} + \mathbf{B}$  is either a tuple  $\langle 1, x \rangle$  for some  $x \in \mathbf{A}$  or a tuple  $\langle 2, y \rangle$  for some  $y \in \mathbf{B}$ . The sets  $\{1\}$  and  $\{2\}$  here simply provide labels which “force” the sets  $\{1\} \times \mathbf{A}$  and  $\{2\} \times \mathbf{B}$  to be disjoint (even if  $\mathbf{A}$  and  $\mathbf{B}$  have elements in common).

Consider the sets  $\mathbf{A} = \{\text{🍪}, \text{🍺}\}$  and  $\mathbf{B} = \{\text{🍷}, \text{🍹}\}$ . Their disjoint union can be represented as:



$$\begin{array}{c} \text{🍪} \\ \text{🍺} \\ \mathbf{A} \end{array} + \begin{array}{c} \text{🍷} \\ \text{🍹} \\ \mathbf{B} \end{array} = \begin{array}{cc} \langle 1, \text{🍪} \rangle & \langle 2, \text{🍷} \rangle \\ \langle 1, \text{🍺} \rangle & \langle 2, \text{🍹} \rangle \\ \mathbf{A} + \mathbf{B} \end{array} \quad (62)$$

We can define the disjoint union of a set with itself; this corresponds to having two distinct copies of the set:



$$\begin{array}{c} \text{🍪} \\ \text{🍺} \\ \mathbf{A} \end{array} + \begin{array}{c} \text{🍪} \\ \text{🍺} \\ \mathbf{A} \end{array} = \begin{array}{cc} \langle 1, \text{🍪} \rangle & \langle 2, \text{🍪} \rangle \\ \langle 1, \text{🍺} \rangle & \langle 2, \text{🍺} \rangle \\ \mathbf{A} + \mathbf{A} \end{array} \quad (63)$$

Let us also think about what happens when the empty set is at play. For example, if  $\mathbf{A} = \emptyset$ , then  $\emptyset + \mathbf{B} = \{2\} \times \mathbf{B}$ , and similarly  $\mathbf{A} + \emptyset = \{\langle 1, x \rangle \mid x \in \mathbf{A}\}$ . Also,  $\emptyset + \emptyset = \emptyset$ .

**Remark 4.9.** If  $\mathbf{A}$  and  $\mathbf{B}$  are finite, then the size of  $\mathbf{A} + \mathbf{B}$  is the sum of the sizes of  $\mathbf{A}$  and  $\mathbf{B}$ :

$$\text{card}(\mathbf{A} + \mathbf{B}) = \text{card}(\mathbf{A}) + \text{card}(\mathbf{B}). \quad (64)$$

This is a reason why we think of  $\mathbf{A} + \mathbf{B}$  as a form of addition of sets.

**Remark 4.10.** Later we will see that the cartesian product of sets is a special case of a very general construction in category theory, called the categorical product, and that the disjoint union of sets is a special case of a “dual” construction, called categorical coproduct.

## 4.4. ○ Type theory

**Remark 4.11.** You can skip this section until you want to get started on the code exercises.

At a basic level, the notion of set seems uncontroversial. However, there are some difficulties that led to the development of *type theory*, which can be used as an alternative foundation to mathematics. Type theory is also used as the foundation for computing; the exercises utilize the type theory nomenclature.

If you are an experienced programmer, you likely already have some informal knowledge of type theory which will be sufficient to carry you through these exercises.

If you are more theoretical inclined, a proper comprehensive introduction to type theory is in Pierce’s book [25].

### Sets/elements vs types/terms

We do not give here a complete introduction to type theory. In fact, there are different type systems. For now, what is important to understand is the difference between set theory and type theory. This allows translating between the theory parts of this book, which use set theory, and the programming exercises.

In type theory there are two basic kinds: *terms* and *types*. A term is a formal expression built from the constructions in a language. A term might have a type. We write  $a : T$  to say that the *term*  $a$  has the *type*  $T$ . For example, we could say  $42 : \text{int}$  to mean that the *term*  $42$  has *type*  $\text{int}$  (the type of integers).

In set theory there are elements and sets. It is assumed that elements have “an identity of their own”: each element is equal to itself and different from the other elements. Furthermore, elements might *belong* to sets.

In type theory the concepts of *equality* and *belonging* are not primitive. Rather, we can build them from scratch, enjoying much more flexibility.

For example, we can define a *setoid* as a set-like structure in type theory as follows.

#### Definition 4.12 (Setoid)

A *setoid* is a tuple  $S = \langle T, \text{contains}, \text{equal} \rangle$  where

- ▷  $T$  is a type;
- ▷  $\text{contains} : T \rightarrow \text{Bool}$  is a function that tells use whether an element belongs to the setoid;
- ▷  $\text{equal} : T \times T \rightarrow \text{Bool}$  is the equality.

The function  $\text{equal}$  should have the properties of an *equivalence relation*, which will be discussed in Def. 5.28.

Here is an example that shows the advantages of such construction. Imagine we want to represent rational numbers with two numbers, nominator and denominator, using arbitrary integers. We want to define a setoid for which the element  $\langle 2, -4 \rangle$  is equal to  $\langle -1, 2 \rangle$ . Assume that we already have a setoid for integers defined, so that we have a function  $\text{equal}_{\text{int}}$ .

We can do it as follows:

$$S_1 = \begin{cases} T = \text{Tuple}[\text{int}, \text{int}] \\ \text{contains} = \langle a, b \rangle \mapsto \text{contains}_{\text{int}}(a) \wedge \text{contains}_{\text{int}}(b) \\ \text{equal} = \langle \langle a, b \rangle, \langle c, d \rangle \rangle \mapsto \text{equal}_{\text{int}}(a, c) \wedge \text{equal}_{\text{int}}(b, d) \end{cases} \quad (65)$$

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

One can define a different setoid with the same underlying type `Tuple[int, int]` where a pair of elements is interpreted as an interval:

$$S_2 = \begin{cases} T = \text{Tuple}[\text{int}, \text{int}] \\ \text{contains} = \langle a, b \rangle \mapsto \text{contains}_{\text{int}}(a) \wedge \text{contains}_{\text{int}}(b) \wedge (a \leq b) \\ \text{equal} = \langle \langle a, b \rangle, \langle c, d \rangle \rangle \mapsto \text{equal}_{\text{int}}(a, c) \wedge \text{equal}_{\text{int}}(b, d) \end{cases} \quad (66)$$

The elements  $\langle 1, 4 \rangle$  and  $\langle 2, 8 \rangle$  are in both setoids. However, we have that

$$\text{equal}_{S_1}(\langle 1, 4 \rangle, \langle 2, 8 \rangle) = \top \quad \text{and} \quad \text{equal}_{S_2}(\langle 1, 4 \rangle, \langle 2, 8 \rangle) = \perp. \quad (67)$$

Every time we have an equality we should think about in which setoid the equality is understood. We would rewrite the above as

$$\langle 1, 4 \rangle =_{S_1} \langle 2, 8 \rangle \quad \text{and} \quad \langle 1, 4 \rangle \neq_{S_2} \langle 2, 8 \rangle. \quad (68)$$

## Python and other design choices

We chose to use Python as the language for the exercises because, as of 2021, it is the most used language in the world, according to the TIOBE index.

We should also note that, fixed the language, there is considerable freedom in defining the interfaces and the class hierarchy. We tried to follow these principles:

- ▷ *Elements are opaque*: We never specify any interface for elements. It is the *structure* that “owns” the elements that knows how to operate on them. For example, we define a `SetProduct` which is a set whose elements are a tuple of elements. Rather than defining a class `ElementTuple` that can tell us its `elements()`, there is a function `unpack` in `SetProduct` that tells us the content of the opaque element.

This works very well and gives the implementer (you) total freedom to choose the internal representation.

However, this might be counter-intuitive for a beginner, as it goes against the beginner’s advice in object-oriented languages, where a `Dog` is an `Animal` that `barks()`.

- ▷ *Object-oriented composition over inheritance*: Two basic principles of object-oriented programming are “composition” (create complex objects out of simpler objects) and “class inheritance” (subclass a simple class to obtain a more complex class). An example use of inheritance would be saying: “a poset is a set which also has an order relation”; composition would be saying “a poset is a tuple of a set and an order relation”. We adopt composition, which is also the way we define mathematical objects. The over-use of inheritance is another beginner mistake in object-oriented programming.

## 4.5. Representing sets

As a first step, we are going to build together the support for sets.

We define three interfaces:

- ▷ **Setoid**, for a generic set;
- ▷ **EnumerableSet**, for a set that can enumerate its elements;
- ▷ **FiniteSet**, for a set that has a finite set of elements.

They are defined in the package `act4e_interfaces` which is in the repository `ACT4E/ACT4E-exercises`.

Figure 5 shows their relation in a UML diagram. The arrow means “inherits from”.

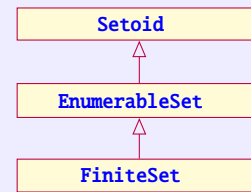


Figure 5.: UML inheritance diagram

### Setoids

Listing 4 shows the interface for **Setoid**.

Listing 4: The **Setoid** interface.

```

class Setoid(Generic[E], ABC):
    """
    A setoid is something to which elements may belong,
    has a way of distinguishing elements,
    and is able to (de)serialize its elements.
    """

    @abstractmethod
    def contains(self, x: E) -> bool:
        """Returns true if the element is in the set."""

    def equal(self, x: E, y: E) -> bool:
        """Returns True if the two elements are to be considered equal."""
        return x == y # default is to use the Python equality

    def apart(self, x: E, y: E) -> bool:
        return not self.equal(x, y)

    @abstractmethod
    def save(self, h: IOHelper, x: E) -> ConcreteRepr: ...

    @abstractmethod
    def load(self, h: IOHelper, o: ConcreteRepr) -> E: ...
  
```

**Setoid** is declared “generic” in **E**; that represents the type of the elements.

The class **Setoid** derives from **ABC**, which means *abstract base class*; it tells the interpreter that some class methods will have to be implemented by the subclasses. This is the equivalent of the “virtual” methods in C++. The methods that need to be implemented are marked by a decorator `@abstractmethod`. For **Setoid**, we ask that the implementer implements the `contains()` function, and they can optionally implement `equal()`.

This is the semantics of the first three methods:

- ▷ The method `contains()` checks if an element is part of this set.
- ▷ The method `equal()` checks if two elements are the same. The default way to do this is to use Python’s equality operator (which calls the `__equal__` method of the class of the first object); later we will see cases in which we might want to choose something different.
- ▷ The method `apart()` checks that two elements are “apart”. You can ignore this for the entire first volume, but it will become more relevant later when we talk

more in detail about constructivism.<sup>‡</sup>

There are other methods that have to do with serializing/deserializing the data:

- ▷ The method `save()` saves an object into a “concrete representation”, indicated by the placeholder type `ConcreteRepr`. We use this type to mean something that can be serialized to YAML.
- ▷ The method `load()` does the inverse.

Note that YAML supports the following types: `int`, `bool`, `float`, `datetime`, `dict`, `list`. Python values composed by these types and compositions are directly representable in YAML. There are two exceptions to note:

- ▷ Class instances;
- ▷ Tuples.

If you use internally those types, you need to come up with a way to represent the data.

For example, imagine that you are using internally a class `Vector`:

```
@dataclass
class Vector:
    x: int
    y: int
    z: int

a = V(1, 2, 3)
```

You might choose either of these possible variations for serialization:

Serialize as array      `[1, 2, 3]`

Serialize as dictionary      `{'x': 1, 'y': 2, 'z': 3}`

The choice is up to you; and the choice is encoded in the load/save functions of the `Setoid` interface.

## Enumerable sets

Listing 5 shows the interface for `EnumerableSet`:

**Listing 5:** The `EnumerableSet` interface.

```
class EnumerableSet(Generic[E], Setoid[E], ABC):
    @abstractmethod
    def elements(self) -> Iterator[E]:
        """Note: possibly non-terminating."""
```

The class inherits from `Setoid` and adds the method `elements()`. The method returns a Python `Iterator`: you can use it to get the elements contained in the `EnumerableSet`. Note that an iterator does not necessarily terminate.

## Finite sets

Listing 6 shows the interface for `FiniteSet`.

**Listing 6:** The `FiniteSet` interface.

<sup>‡</sup> See Apartness relation on Wikipedia.



```
class FiniteSet(Generic[E], EnumerableSet[E], ABC):
    """A finite set has a finite size."""

    @abstractmethod
    def size(self) -> int:
        """Return the size of the finite set."""
```

The additional method `size()` returns the size of the set. For a `FiniteSet`, the iterator return `elements()` terminates.

You will have to implement the `FiniteSet` class as part of the exercise.

## Representation for finite sets

We have defined formats for all the structures to be used in the exercises. The file format for `FiniteSet` is shown in Fig. 6. The file represents a dictionary with only one field called `elements`, which contains a list of elements. YAML can represent most primitive data format of Python, as well as lists and dictionaries. We visualize the data samples in the book using YAML, but you never have to deal with YAML yourself. You can expect to receive a Python data structure as in Fig. 7.

Throughout the exercises we will encounter many times classes that must load or save data. We describe them through the interfaces of `Loader`, `Saver`, and (union of the two) `Serializer`. The two type parameters `X` and `R` correspond to the Python type that they serialize and to the resulting serialization format. For now, you can ignore the argument `h: IOHelper`.

Listing 7: The `Loader` interface.

```
class Loader(Protocol[X, R]):
    def load(self, h: IOHelper, data: R) -> X:
        """Load a finite set from data structure.
        Throw InvalidFormat if the format is incorrect.
        """
    ...
```

Listing 8: The `Saver` interface.

```
class Saver(Protocol[X, R]):
    def save(self, h: IOHelper, ob: X) -> R: ...
```

Listing 9: The `Serializer` interface.

```
class Serializer(Generic[X, R], Loader[X, R], Saver[X, R]):
    ...
```

In particular, `FiniteSetRepresentation` is a serializer of `FiniteSet`. The type `FiniteSet_desc` is a type that describes the serialization format in Fig. 6. We don't show it, but you can click through and see the definition in the development environment.

Listing 10: The `FiniteSetRepresentation` interface.

```
class FiniteSetRepresentation(Serializer[FiniteSet[Any], FiniteSet_desc],
    <- ABC):
    @abstractmethod
    def load(self, h: IOHelper, data: FiniteSet_desc) -> FiniteSet[Any]:
        """Load a finite set from data structure.
        Throw InvalidFormat if the format is incorrect.
        """
```

```
elements: [a, b, c]
```

Figure 6.: Example shown in YAML format.

```
{"elements": ["a", "b", "c"] }
```

Figure 7.: We show data formats in YAML because it is terse, but in the exercises you will receive the Python data structure directly.

```
@abstractmethod
def save(self, h: IOHelper, f: FiniteSet[Any]) -> FiniteSet_desc:
    """Serializes into a data structure"""
```

In the listing above, for clarity, we wrote the signature of the methods `load()` and `save()`; however, these are implied by the fact that the class inherits from `Serializer`.

#### Code exercise B.1 (TestFiniteSetRepresentation)

The first exercise is to implement the methods that read and write from this file format. You have to implement the interface `FiniteSetRepresentation` shown in Listing 10. As part of the exercise, you will also have to implement a concrete subclass of `FiniteSet`.

Check your solution using:

```
act4e-check TestFiniteSetRepresentation
```

## Walk-through for first exercise

We are going to solve this exercise together.

We assume that you have checked out the template repository as explained in the setup section.

There is already a skeleton implemented in the template in the file

`act4e_solutions/sets_representation.py`

which contains the following:

Listing 11

```
from typing import Any

import act4e_interfaces as I

class SolFiniteSetRepresentation(I.FiniteSetRepresentation):
    def load(self, h: I.IOHelper, data: I.FiniteSet_desc) -> I.FiniteSet[Any]:
        raise NotImplementedError()

    def save(self, h: I.IOHelper, f: I.FiniteSet[Any]) -> I.FiniteSet_desc:
        raise NotImplementedError()
```

We can start by checking if this is a valid solution. Type this:

```
act4e-check TestFiniteSetRepresentation
```

Here `TestFiniteSetRepresentation` is the code name for the exercise. You can check other exercises by changing the name.

This will fail; it will complain saying that some parts of the exercise were not implemented.

First, you need to add your implementation of `FiniteSet`. You can do this in `act4e_solutions/sets_representation.py` or create a separate file (called `act4e_solutions/my_finite_sets.py` for instance). For the latter, don't forget to import the newly created class in `act4e_solutions/sets_representation.py` by adding the following line:

```
from .my_finite_sets import MyFiniteSet
```

A possible skeleton is in Listing 12. Note that what matters, is the class you are concretely implementing, and not the name you give to it.

Listing 12

```

from typing import Collection, Iterator, List, TypeVar

import act4e_interfaces as I

E = TypeVar("E")

class MyFiniteSet(I.FiniteSet[E]):
    _elements: List[E]

    def __init__(self, elements: Collection[E]): ...

    def size(self) -> int: ...

    def contains(self, x: E) -> bool: ...

    def elements(self) -> Iterator[E]: ...

    def save(self, h: I.IOHelper, x: E) -> I.ConcreteRepr: ...

    def load(self, h: I.IOHelper, o: I.ConcreteRepr) -> E: ...

```

At this point we need to implement the rest of the code.

For example, Listing 13 is a valid implementation of `FiniteSet`.

Listing 13

```

from typing import cast, Collection, Iterator, List, TypeVar

import act4e_interfaces as I

E = TypeVar("E")

class MyFiniteSet(I.FiniteSet[E]):
    _elements: List[E]

    def __init__(self, elements: Collection[E]):
        self._elements = list(elements)

    def size(self) -> int:
        return len(self._elements)

    def contains(self, x: E) -> bool:
        for y in self._elements:
            if self.equal(x, y):
                return True
        return False

    def elements(self) -> Iterator[E]:
        for _ in self._elements:
            yield _

    def save(self, h: I.IOHelper, x: E) -> I.ConcreteRepr:
        return cast(I.ConcreteRepr, x)

    def load(self, h: I.IOHelper, o: I.ConcreteRepr) -> E:
        return cast(E, o)

```

The loading and saving of the data is very simple. To load just take the `elements` field of the data structure and give it to `MyFiniteSet`. To save, reconstruct the list elements from the `MyFiniteSet` instance by using the method `elements()`, which returns an iterator. Note that the `save()` function you have to implement must work for all `FiniteSet` instances, not just your particular implementation of `FiniteSet`;

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

therefore, you cannot access the `_elements` field directly. An example of this is shown in Listing 14.

Listing 14

```
from typing import TypeVar

import act4e_interfaces as I

E = TypeVar("E")

class MyFiniteSetRepresentation(I.FiniteSetRepresentation):
    def load(self, h: I.IOHelper, data: I.FiniteSet_desc) -> I.FiniteSet[E]:
        if not isinstance(data, dict):
            raise I.InvalidFormat()
        # note: later on the format will be extended
        if not "elements" in data:
            raise I.InvalidFormat()
        if not isinstance(data["elements"], list):
            raise I.InvalidFormat()
        elements = data["elements"]
        return MyFiniteSet(elements)

    def save(self, h: I.IOHelper, f: I.FiniteSet[E]) -> I.FiniteSet_desc:
        all_elements = [f.save(h, _) for _ in f.elements()]
        return {"elements": all_elements}
```

With this code, the tests should pass. Try again to run:

```
act4e-check TestFiniteSetRepresentation
```

Note that the types used by the function `save()` of `SolFiniteSetRepresentation` is `FiniteSet`, not `MyFiniteSet`. The method must be able to work for *any* implementation of `FiniteSet`, not just yours. In fact, the tester will try to call that function with a different implementation.

Note that the implementation above uses this line to get the elements:

```
all_elements = [f.save(h, _) for _ in f.elements()] # ok
```

This is correct because it is using the method `elements()` that all implementations of `FiniteSet` need to have. What could be wrong is using code like the following:

```
all_elements = [f.save(h, _) for _ in f._elements] # !!! not correct
```

This code accesses the attribute `_elements` of the class `MyFiniteSet`. It will not work with other implementations. In fact, the tests will fail.

## 4.6. FiniteSetProperties

### Code exercise B.2 (TestFiniteSetProperties)

Given two finite sets, check if they are a subset of the other.

Implement the interface in Listing 15.

Test using:

```
act4e-check TestFiniteSetProperties
```

**Listing 15:** The `FiniteSetProperties` interface.

```
class FiniteSetProperties(ABC):
    @abstractmethod
    def is_subset(self, a: FiniteSet[E], b: FiniteSet[E]) -> bool:
        """True if `a` is a subset of `b`."""

    def equal(self, a: FiniteSet[E], b: FiniteSet[E]) -> bool:
        return self.is_subset(a, b) and self.is_subset(b, a)

    def is_strict_subset(self, a: FiniteSet[E], b: FiniteSet[E]) -> bool:
        return self.is_subset(a, b) and not self.is_subset(b, a)
```

The skeleton is provided in `act4e_solutions/sets_properties.py` in the class `Sol-FiniteSetProperties`.

As you can see, once you define `is_subset()`, you get for free the implementation of `equal()` and `is_strict_subset()`. Of course, there might be more efficient ways to implement `equal()` than two calls of `is_subset()`.

## 4.7. ○ Power set

### Code exercise B.3 (TestFiniteMakePowerSet)

Compute the power set of a finite set.

Implement the interface in Listing 16.

Test using:

```
act4e-check TestFiniteMakePowerSet
```

The skeleton is provided in `act4e_solutions/sets_power.py` in the class `SolFiniteMakePowerSet`.

Listing 16: The `FiniteMakePowerSet` interface.

```
class FiniteMakePowerSet(ABC):
    @abstractmethod
    def powerset(self, s: FiniteSet[C]) -> FiniteSetOfFiniteSubsets[C, Any]:
        ...
```

Note that `powerset()` must return a special setoid that is a `FiniteSetOfFiniteSubsets` (Listing 18), which derives from `SetOfFiniteSubsets`.

Listing 17: The `SetOfFiniteSubsets` interface.

```
class SetOfFiniteSubsets(Generic[C, E], Setoid[E], ABC):
    """A set of subsets."""

    @abstractmethod
    def contents(self, e: E) -> Iterator[C]:
        """Returns the contents of an element representing a subset."""

    @abstractmethod
    def construct(self, elements: Collection[C]) -> E:
        """Get the element representing the given subset."""
```

Listing 18: The `FiniteSetOfFiniteSubsets` interface.

```
class FiniteSetOfFiniteSubsets(Generic[C, E], SetOfFiniteSubsets[C, E],
    FiniteSet[E], ABC):
    pass
```

These classes are declared generic in `C` and `E`. `C` is the type of the elements inside the subset. `E` is the type of the subset.

The function `contents()` decomposes a subset in components. The function `construct()` constructs a subset from a sequence of elements.

Note how we are not constraining what is the type of `E`: we do not care how the implementation represents a subset, but just that we can access the elements inside it.

As for `FiniteSet`, you will have to implement the class `FiniteSetOfFiniteSubsets` as part of this exercise. Again, you can either do this in `act4e_solutions/sets_power.py` or create a separate file (called `act4d_solutions/my_power_sets.py` for instance). For the latter, don't forget to import the newly created class in `act4e_solutions/sets_power.py` by adding the following line:

```
from .my_power_sets import MyFiniteSetOfFiniteSubsets
```

Moreover, don't forget to define the `__init__()` for `MyFiniteSetOfFiniteSubsets` and to implement all the methods in the classes `FiniteSetOfFiniteSubsets` inherits from. That is you need to implement the methods marked by a decorator `@abstractmethod` in `SetOfFiniteSubsets`, `FiniteSet` and `Setoid`.

## 4.8. Set products

We define a **SetProduct** to be a **Setoid** that also has the structure of the cartesian product of sets. This means that:

- ▷ It is possible to recover the “factors” of the product (if  $A = B \times C \times D$ , we must be able to recover the components **B**, **C**, **D** from **A**);
- ▷ It is possible to access the elements of a tuple (if  $A = B \times C \times D$  and  $x = \langle b, c, d \rangle$ , we should be able to access the components of the tuple).
- ▷ On the other way around: it is possible to construct a tuple from the single components.

Our Python interpretation of these properties is specified in the class **SetProduct**.

**Listing 19:** The **SetProduct** interface.

```
class SetProduct(Generic[C, E], Setoid[E], ABC):
    """A set product is a setoid that can be factorized."""

    @abstractmethod
    def components(self) -> Sequence[Setoid[C]]:
        """Returns the components of the product."""

    @abstractmethod
    def pack(self, args: Sequence[C]) -> E:
        """Packs an element of each setoid into an element of the product."""

    @abstractmethod
    def unpack(self, args: E) -> Sequence[C]:
        """Unpacks an element of the product to the constituents."""
```

The semantics is the following:

- ▷ The generic type variable **C** represents the type of the components; the generic type variable **E** refers to the type of the product. Note that we are not constraining the internal representation of the product. (A possible choice could be  $E = \text{List}[C]$ ).
- ▷ The method **components()** returns the ordered list of the setoid factors; these must be setoids of type **C**.
- ▷ The method **pack()** takes a number of arguments and creates an element of the product.
- ▷ The method **unpack()** allows to recover the components of an element of the product.

There is also a specialization, which corresponds to a product of finite sets; the only change is that the components are finite sets rather than setoids.

**Listing 20:** The **FiniteSetProduct** interface.

```
class FiniteSetProduct(Generic[C, E], FiniteSet[E], SetProduct[C, E], ABC):
    """Specialization of SetProduct where we deal with FiniteSets"""

    @abstractmethod
    def components(self) -> Sequence[FiniteSet[C]]:
        """Returns the components"""
```

### Code exercise B.4 (TestFiniteMakeSetProduct)

Implement the interface in Listing 21.

Test using:

```
act4e-check TestFiniteMakeSetProduct
```

Listing 21: The `FiniteMakeSetProduct` interface.

```
class FiniteMakeSetProduct(ABC):
    @abstractmethod
    def product(self, components: Sequence[FiniteSet[C]]) ->
        FiniteSetProduct[C, Any]: ...
```

As in the previous exercises, you need to implement the class `FiniteSetProduct` and all the methods from the classes it inherits from as well as the `__init__()` method. This procedure will be similar for subsequent exercises and won't be repeated every time.

## Representation

```
product:
- elements: [a, b]
- elements: [1, 2]
```

Figure 8.:  $\{a, b\} \times \{1, 2\}$ 

```
product:
- elements: [1]
- elements: [1]
- elements: [1]
```

Figure 9.:  $\{1\} \times \{1\} \times \{1\}$ 

```
product:
- elements: [1]
- elements: []
```

Figure 10.:  $\{1\} \times \emptyset$ 

### Code exercise B.5 (TestFiniteSetRepresentationProduct)

Extend now the code you wrote for loading and saving sets to allow the format in Figs. 8 to 10. We add another clause to the parsing algorithm:

1. If it has a field `elements`, it is a finite set described with elements directly.
2. If there is a field `product`, it must be a list of sets, and the semantics is the product of those sets.
3. Otherwise, it is an error – for now; we will introduce many more ways to describe sets.

Test your results using

```
act4e-check TestFiniteSetRepresentationProduct
```



## 4.9. Disjoint union

A `SetDisjointUnion` is a particular type of set, which has two extra methods: `pack()` and `unpack()`:

- ▷ The method `pack()` allows constructing an element of the disjoint union in a given component.
- ▷ The method `unpack()` allows deducing which component the element comes from and get its original value.

Listing 22: The `SetDisjointUnion` interface.

```
class SetDisjointUnion(Generic[C, E], Setoid[E], ABC):
    @abstractmethod
    def components(self) -> Sequence[Setoid[C]]:
        """Returns the components of the union"""

    @abstractmethod
    def pack(self, i: int, e: C) -> E:
        """Injection mapping: construct element of the i-th component.
        Raises InvalidValue if i is not a valid index or e is not an element of the i-th component.
        """

    @abstractmethod
    def unpack(self, e: E) -> Tuple[int, C]:
        """Returns the index of the component and the element."""
```

Listing 23: The `FiniteSetDisjointUnion` interface.

```
class FiniteSetDisjointUnion(Generic[C, E], FiniteSet[E], SetDisjointUnion[C, E], ABC):
    """Specialization of SetDisjointUnion where we deal with FiniteSets."""

    @abstractmethod
    def components(self) -> Sequence[FiniteSet[C]]: ...
```

### Code exercise B.6 (TestFiniteMakeSetDisjointUnion)

Write code to create the disjoint union of a list of sets.

Implement the interface in Listing 24.

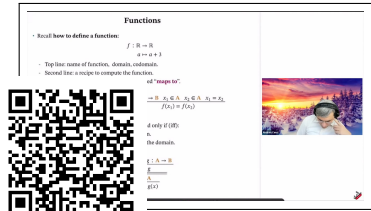
Test using:

```
act4e-check TestFiniteMakeSetDisjointUnion
```

Listing 24: The `FiniteMakeSetDisjointUnion` interface.

```
class FiniteMakeSetDisjointUnion(ABC):
    @abstractmethod
    def disjoint_union(self, components: Sequence[FiniteSet[C]]) -> FiniteSetDisjointUnion[C, Any]:
        """Returns a disjoint union of the given components."""
```

Watch: *Functions nomenclature* (7 minutes).



$x$	$f(x)$
1	1
2	4
3	9
...	...

Figure 11.: A function described via a table.

4.10. Functions

Functions are familiar to all of us – to some degree – from our school days. In this section we present some basic terminology and our way of defining functions formally, and we discuss various ways to *think* about functions. When speaking about functions, we use the words “map” or “mapping” as synonyms.

Specifying functions

Functions can be specified in a variety of ways. Sometimes they are indicated with the help of a formula, such as  $f(x) = x^2$ . Or with the help of a table, as in Fig. 11. Other times a function might be characterized by equations or properties. For example, the exponential function

$$\begin{aligned} \exp : \mathbb{R} &\rightarrow \mathbb{R}, \\ x &\mapsto e^x, \end{aligned}$$

(69)

is known to be characterized by the fact that it satisfies the differential equation  $f'(x) = f(x)$  and the initial condition  $f(0) = 1$ . Still other times one might be able to prove the existence (and perhaps also uniqueness) of some function satisfying some given properties, but one might not have any concrete means to “evaluate” or “calculate” that function. Whatever the route may be by which a function is specified, for us an essential non-negotiable part of specifying a function is to say which set is its *source* (or *domain*), and which set is its *target* (or *co-domain*). That is, a function  $f$  is always something that goes from one set  $\mathbf{A}$  (the source of  $f$ ) to another set  $\mathbf{B}$  (the target of  $f$ ). We write this as  $f : \mathbf{A} \rightarrow \mathbf{B}$ . For example, the formula “ $f(x) = x^2$ ” does not specify a function yet, because we did not yet say what source and target set we are considering. If we are thinking of  $x^2$  as defining a function  $\mathbb{R} \rightarrow \mathbb{R}$ , then this is one function, and if we are thinking of  $x^2$  as defining a function  $\mathbb{R}_{>0} \rightarrow \mathbb{R}$ , then this is another function. And if we are thinking of  $x^2$  as a function  $\mathbb{N} \rightarrow \mathbb{N}$ , this is yet another.

One way to see why this is important: the function

$$\begin{aligned} f : \mathbb{R}_{>0} &\rightarrow \mathbb{R}, \\ x &\mapsto x^2, \end{aligned}$$

(70)

is monotone (increasing), while

$$\begin{aligned} g : \mathbb{R} &\rightarrow \mathbb{R}, \\ x &\mapsto x^2, \end{aligned}$$

(71)

is not. They have different properties. Or, as another example, the function  $g$  can be shown to be a continuous function, while for the function

$$\begin{aligned} h : \mathbb{N} &\rightarrow \mathbb{N}, \\ x &\mapsto x^2, \end{aligned}$$

(72)

it is perhaps not immediately clear what the question of continuity even means.

## Functions as deterministic machines

One typical way of thinking about functions is in terms of “input” and “output”. Given a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  we sometimes speak of “plugging in” an element  $x \in \mathbf{A}$  into the function  $f$ , and then it will “output” an element  $f(x) \in \mathbf{B}$ .

One reason for this kind of thinking is that sometimes functions describe things that are like a computational process or a machine: for instance, we might give a software program an input, it might then perform a series of computations, and then output an answer, and all of this might be described by a function.

Another reason for thinking of functions in terms of input and output is because humans often use functions – *as mathematical entities* – in a deterministic “machine” kind of way. Starting with some element  $x \in \mathbf{A}$ , we can sometimes use the function  $f$  to *calculate* or otherwise *determine* what the “output”  $f(x) \in \mathbf{B}$  is. For example, if we consider the function

$$\begin{aligned} f : \mathbb{Z} &\rightarrow \mathbb{Z}, \\ x &\mapsto x^2, \end{aligned} \tag{73}$$

then, given any input  $x \in \mathbb{Z}$ , we can compute the output  $f(x) \in \mathbb{Z}$  by multiplying the input  $x$  with itself.

Mathematically speaking, functions are *deterministic* in the sense that for any input  $x$ , there is *exactly one* output  $f(x)$ . This is in contrast to the fact that a given output  $f(x)$  might arise from various possible inputs: for example  $4 \in \mathbb{Z}$  could be the output of (73) for the input  $2 \in \mathbb{Z}$  or for the input  $-2 \in \mathbb{Z}$ .

## Functions as relations

Another point of view is that a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  defines a certain kind of *relation* between the elements of  $\mathbf{A}$  and  $\mathbf{B}$ . Given an  $x \in \mathbf{A}$ , the function  $f$  tells us that this  $x$  is related to a certain  $y \in \mathbf{B}$ , which we happen to call  $f(x)$ . This point of view is fully compatible with thinking of functions as “mathematically deterministic”. However, it is more general than interpreting functions as describing processes which are “physically deterministic” in any sense or where “the input precedes the output”.

As an illustration, consider a large phone book (of personal mobile numbers), which is just a table of names and phone numbers. Let  $\mathbf{A}$  be the set of phone numbers in the book, and  $\mathbf{B}$  the set of names. There is a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  which, given any phone number in  $\mathbf{A}$ , will output the name of the person to whom that number is registered. Normally, every number is assigned to a single name, so a name as an “output” of the function is completely determined (mathematically speaking) by the number one “inputs”. However, there is no “physical determinism” here: there is no non-mathematical process by which the name of the person was “computed” or “causally determined” by the phone number. Rather, the function we described arises simply from a table of information.

## A formal definition

The idea is that we can formally define a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  by the ordered pairs  $\langle x, y \rangle \in \mathbf{A} \times \mathbf{B}$  which are the elements of what might be called the “graph” of the function. In other words, those ordered pairs of the form “ $\langle x, f(x) \rangle$ ”.

### Definition 4.13 (Function)

Let  $\mathbf{A}$  and  $\mathbf{B}$  be sets. A function  $f : \mathbf{A} \rightarrow \mathbf{B}$  is a subset

$$f \subseteq \mathbf{A} \times \mathbf{B}, \quad (74)$$

with the following property:

$$\forall x \in \mathbf{A} \quad \exists! y \in \mathbf{B} : \langle x, y \rangle \in f. \quad (75)$$

Here  $\mathbf{A}$  is the source (or domain) and  $\mathbf{B}$  is the target (or co-domain) of  $f$ .

We emphasize, once again, that the source and target of a function are “baked in” as *part of the definition* of the function.

The property (75) describes the “mathematical determinism” that functions are supposed to have: for any  $x \in \mathbf{A}$  there exists *exactly one* element  $y \in \mathbf{B}$  that is “the result” of the function  $f$  applied to  $x$ .

Another important aspect of (75) is that it says that *for every*  $x \in \mathbf{A}$  there exists a  $y \in \mathbf{B}$  that is related to  $x$  by  $f$ . In other words, we do not allow functions to be “partially defined”. For example, the formula “ $f(x) = 1/x$ ” could be used to define a function  $\mathbb{R} \setminus \{0\} \rightarrow \mathbb{R}$ , but it would *not* be valid for defining a function  $\mathbb{R} \rightarrow \mathbb{R}$ .

Although we take Def. 4.13 as our *formal* definition of functions, we will continue to use the standard kinds of notation for functions, for example usually writing  $y = f(x)$  and not  $\langle x, y \rangle \in f$ . The formal definition above is useful to keep in the back of our minds though. For instance, when thinking about situations involving the empty set.

## To and from the empty set

Do there exist functions  $\emptyset \rightarrow \mathbf{B}$  for any set  $\mathbf{B}$ ? What about  $\mathbf{A} \rightarrow \emptyset$ ?

Consulting Def. 4.13, we can figure out that there is *always* a function  $\emptyset \rightarrow \mathbf{B}$  (no matter what set  $\mathbf{B}$  is) because the condition “ $\forall x \in \mathbf{A} \dots$ ” in (75) is trivially satisfied, as we are quantifying over  $\mathbf{A} = \emptyset$ . In this case,  $f \subseteq \emptyset \times \mathbf{B}$  corresponds to  $\emptyset \subseteq \emptyset \times \mathbf{B} = \emptyset$ .

On the other hand, if  $\mathbf{A} \neq \emptyset$ , there are *no* functions of the type  $\mathbf{A} \rightarrow \emptyset$ , because the part “ $\exists! y \in \mathbf{B}$ ” of (75) cannot be satisfied, since here  $\mathbf{B} = \emptyset$ .

## Injective, surjective, bijective functions

Even if we don’t know a lot of the specifics of some functions, there is a lot we can still say about how functions between sets can behave *in general*. In the following we review a number of basic observations and properties.

Let  $f : \mathbf{A} \rightarrow \mathbf{B}$  be a function.

### Definition 4.14 (Injective function)

A  $f$  is said to be *injective* if for all  $x_1, x_2 \in \mathbf{A}$

$$\frac{f(x_1) = f(x_2)}{x_1 = x_2}, \quad (76)$$

### Definition 4.15 (Surjective function)

and  $f$  is called *surjective* if the condition

$$\forall y \in \mathbf{B} \exists x \in \mathbf{A} : f(x) = y \quad (77)$$

holds.

### Definition 4.16 (Bijective function)

A function which is both injective and surjective is called *bijective*.

**Exercise 6.** For the following functions, determine whether they are injective, surjective, or bijective, and explain why:

- $$f: \mathbb{R} \rightarrow \mathbb{R},$$

$$x \mapsto x + 10.$$
- $$g: \mathbb{R} \rightarrow \mathbb{R},$$

$$x \mapsto x^2.$$
- $$h: \mathbb{N} \rightarrow \mathbb{N},$$

$$x \mapsto x^2.$$
- $$k: \mathbb{N} \rightarrow \mathbb{N},$$

$$x \mapsto 3x.$$

See solution on page 97.

### Image, preimage

The *image* of  $f: A \rightarrow B$  is the set

$$f(A) := \{y \in B \mid \exists x \in A: f(x) = y\}. \quad (78)$$

More generally, given a subset  $S \subseteq A$ , its image under  $f$  is

$$f(S) := \{y \in B \mid \exists x \in S: f(x) = y\}. \quad (79)$$

Given a subset  $T \subseteq B$ , its *preimage* under  $f$  is

$$f^{-1}(T) := \{x \in A \mid f(x) \in T\}. \quad (80)$$

An alternative way of phrasing injectivity of  $f$  is to say that for every singleton subset  $\{y\} \subseteq B$ , its preimage under  $f$  is either a singleton set or the empty set. Surjectivity of  $f$  is equivalent to saying that  $f(A) = B$ .

### Function composition

Importantly, functions can be *composed* when the target set of one functions is the same as the source set of another.

### Definition 4.17 (Composition of functions)

Given functions  $f: A \rightarrow B$  and  $g: B \rightarrow C$ , we denote their *composition* by

$$f \circ g: A \rightarrow C,$$

$$x \mapsto g(f(x)). \quad (81)$$

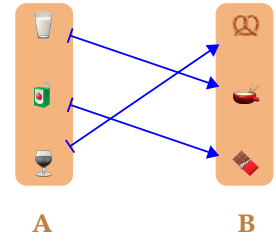


Figure 12.: An injective function.

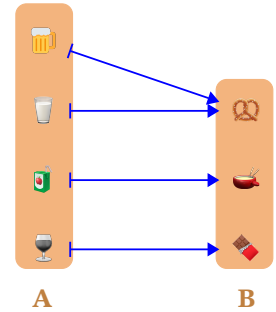


Figure 13.: A surjective function.

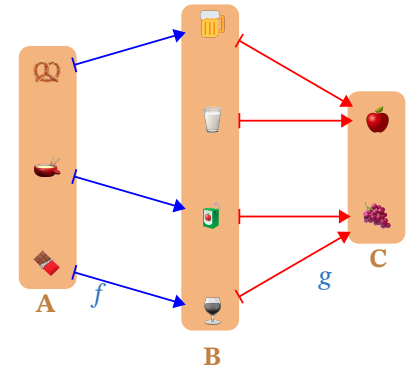


Figure 14.

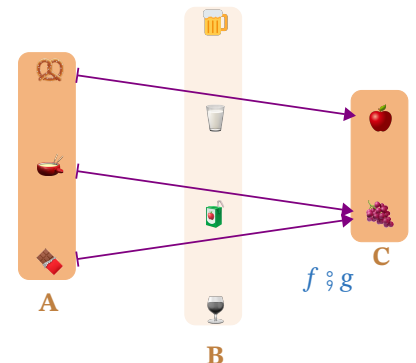


Figure 15.

The notation “ $f \circ g$ ” is different from the more traditional notation “ $g \circ f$ ”. We speak pronounce it as “ $f$  then  $g$ ”, which aligns with the fact that, to evaluate the composition  $f \circ g$  at an element  $x \in A$ , we first apply  $f$  to compute  $f(x)$ , and then we apply  $g$  to compute the result.

## Identity functions

For every set  $B$  there is a special function  $\text{id}_B : B \rightarrow B$  which “does nothing”.

**Definition 4.18** (Identity function on a set)  
The *identity function* on a set  $B$  is given by

$$\begin{aligned} \text{id}_B : B &\rightarrow B, \\ y &\mapsto y. \end{aligned} \tag{82}$$

Because such a function “does nothing”, it behaves neutrally with respect to the composition of functions: given  $f : A \rightarrow B$  and  $g : B \rightarrow C$ , we have  $f \circ \text{id}_B = f$  and  $\text{id}_B \circ g = g$ .

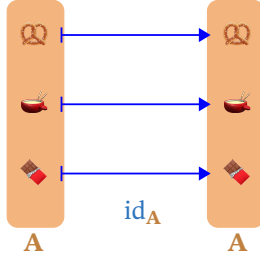


Figure 16.: An identity function

## Isomorphisms

Identity functions are used, for example, to say when a function is *invertible* or, synonymously, that it is an *isomorphism*.

**Definition 4.19** (Isomorphism)

A function  $f : A \rightarrow B$  is an isomorphism if there exists an *inverse* to  $f$ : a function  $g : B \rightarrow A$  such that

$$f \circ g = \text{id}_B \quad \text{and} \quad g \circ f = \text{id}_A. \tag{83}$$

**Exercise 7.** Show that an inverse to  $f$  is necessarily unique (so we can speak of “the” inverse).

*See solution on page 98.*

**Remark 4.20.** Note that if  $g$  is the inverse to  $f$ , then also  $f$  is the inverse of  $g$ .

**Exercise 8.** Show that a function is an isomorphism if and only if it is bijective.

*See solution on page 98.*

**Definition 4.21** (Isomorphisms of sets)

Given sets  $A$  and  $B$ , we say that they are *isomorphic*, and write  $A \simeq B$ , if there exists an isomorphism  $A \rightarrow B$  (or  $B \rightarrow A$ ).

We can use the notion of isomorphism to make a more formal definition of the size, or cardinality, of a set, as follows. First we posit that any set  $A$  has an attribute which we call its cardinality, and denote by  $\text{card}(A)$  (as before). Then we say that for any two sets,  $A$  and  $B$ , it shall hold that

$$\begin{aligned} \text{card}(A) &= \text{card}(B) \\ \hline A &\simeq B \end{aligned} \tag{84}$$

For a finite set  $A$ , we say it has size  $n \in \mathbb{N}$  if there exists an isomorphism between  $A$  and the set  $\{1, 2, \dots, n\}$ .

**Graded exercise B.2** (CountingIsos)

Let  $\mathbf{A} = \{\star, \circ\}$ ,  $\mathbf{B} = \{1, 2, 3\}$ , and  $\mathbf{C} = \{a, b, c\}$ .

1. How many isomorphisms are there  $\mathbf{A} \rightarrow \mathbf{B}$ ?
2. How many isomorphisms are there  $\mathbf{B} \rightarrow \mathbf{C}$ ?

## Sets of functions

So far we have mostly been thinking of functions as a way to relate one set to another. However, in our formal definition, a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  is a certain kind of element in  $\text{Pow}(\mathbf{A} \times \mathbf{B})$ . Consider all those elements of  $\text{Pow}(\mathbf{A} \times \mathbf{B})$  which are indeed functions: they form a set, the set of all functions from  $\mathbf{A}$  to  $\mathbf{B}$ . A notation we will use for this set is  $\mathbf{B}^{\mathbf{A}}$ .

Why this notation?

If  $\mathbf{A}$  and  $\mathbf{B}$  are finite sets, then  $\mathbf{B}^{\mathbf{A}}$  is also a finite set and its size is the size of  $\mathbf{B}$  to the power of the size of  $\mathbf{A}$ :

$$\text{card}(\mathbf{B}^{\mathbf{A}}) = \text{card}(\mathbf{B})^{\text{card}(\mathbf{A})}. \quad (85)$$

## Arithmetic with sets

In our notation for cartesian product, disjoint union, and function sets we have used notation inspired by basic operations in arithmetic, motivated in part by the following formulas for sizes of finite sets:

$$\text{card}(\mathbf{A} \times \mathbf{B}) = \text{card}(\mathbf{A}) \cdot \text{card}(\mathbf{B}), \quad (86)$$

$$\text{card}(\mathbf{A} + \mathbf{B}) = \text{card}(\mathbf{A}) + \text{card}(\mathbf{B}), \quad (87)$$

$$\text{card}(\mathbf{B}^{\mathbf{A}}) = \text{card}(\mathbf{B})^{\text{card}(\mathbf{A})}. \quad (88)$$

The parallels of these operations to operations in arithmetic go further. For example, the following identities hold already on the level of sets, before computing their size; we simply need to replace “=” with the symbol “ $\simeq$ ” for “isomorphic”:

$$\mathbf{A} \times (\mathbf{B} + \mathbf{C}) \simeq (\mathbf{A} \times \mathbf{B}) + (\mathbf{A} \times \mathbf{C}), \quad (89)$$

$$\mathbf{A}^{\mathbf{B}+\mathbf{C}} \simeq \mathbf{A}^{\mathbf{B}} \times \mathbf{A}^{\mathbf{C}}, \quad (90)$$

$$(\mathbf{A}^{\mathbf{B}})^{\mathbf{C}} \simeq \mathbf{A}^{\mathbf{B} \times \mathbf{C}}. \quad (91)$$

These formulas are actually true for sets of any cardinality, not just finite sets.

## Product and sum of functions

In the following, we define the *product* and *sum* of functions. These concepts go hand-in-hand with the notions of product and sum of sets.

**Definition 4.22** (Product of functions)

Given functions  $f : \mathbf{A} \rightarrow \mathbf{B}$  and  $g : \mathbf{C} \rightarrow \mathbf{D}$ , their *product* is the function:

$$\begin{aligned} f \times g : \mathbf{A} \times \mathbf{C} &\rightarrow \mathbf{B} \times \mathbf{D}, \\ \langle a, c \rangle &\mapsto \langle f(a), g(c) \rangle. \end{aligned} \quad (92)$$



**Example 4.23.** Consider  $f, g : \mathbb{R} \rightarrow \mathbb{R}$  with  $f(a) = a^2$  and  $g(b) = b + 1$ . We have  $(f \times g)(2, 5) = \langle 2^2, 5 + 1 \rangle = \langle 4, 6 \rangle$ .

**Definition 4.24** (Sum of functions)

Given functions  $f : \mathbf{A} \rightarrow \mathbf{B}$  and  $g : \mathbf{C} \rightarrow \mathbf{D}$ , their *sum* is the function:

$$\begin{aligned} f + g : \mathbf{A} + \mathbf{C} &\rightarrow \mathbf{B} + \mathbf{D} \\ \langle 1, a \rangle &\mapsto \langle 1, f(a) \rangle, \\ \langle 2, c \rangle &\mapsto \langle 2, g(c) \rangle. \end{aligned} \tag{93}$$

**Example 4.25.** Consider  $f : \mathbb{Z} \rightarrow \mathbb{N}$  with  $f(a) = a^2$  and  $g : \mathbb{R} \rightarrow \mathbb{R}$  with  $g(b) = b^3$ . We have  $(f + g)((1, 2)) = f(2) = 2^2 = 4$ , and  $(f + g)((2, 3)) = g(3) = 3^3 = 27$ .

## The compositional perspective

A central theme in this book is to study how various mathematical structures compose. For instance, what are general patterns of how functions and sets *relate to one another* via composition?

In time, we will see that many features that sets and functions exhibit can be broadly generalized to other kinds of mathematical entities. A guiding philosophy for making or understanding such generalizations is to formulate properties of functions in a way that only uses their “external compositional aspects” and does not rely on the fact that we are dealing with sets, which have elements, and so we can “look inside them”.

This likely sounds very vague at the moment. Let us illustrate with some examples.

**Example 4.26.** Consider the property that a function may (or may not) have of being bijective. According to Exercise 8, a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  is bijective if and only if it is an isomorphism. The latter means, by definition, that there exists  $g : \mathbf{B} \rightarrow \mathbf{A}$  such that  $f \circ g = \text{id}_{\mathbf{A}}$  and  $g \circ f = \text{id}_{\mathbf{B}}$ . The point is that the equations in the definition of “isomorphism” only make use of the operation of function composition, the notion of equality, and the existence of special identity functions. There is no mention of elements of sets, as there is in the definition of “bijective”.

**Example 4.27.** The notion of “subset” is traditionally defined, as we did above, by saying that  $\mathbf{A} \subseteq \mathbf{B}$  if and only if  $\forall x \in \mathbf{A}$ :

$$\frac{x \in \mathbf{A}}{x \in \mathbf{B}}. \tag{94}$$

There are, however, alternatives that do not refer to “elements”. To see one way, consider the set  $\mathbf{2} = \{0, 1\}$ . Any function  $f : \mathbf{B} \rightarrow \mathbf{2}$  defines a subset

$$\mathbf{A}_f = \{y \in \mathbf{B} \mid f(y) = 1\} \subseteq \mathbf{B}. \tag{95}$$

Conversely, any subset  $\mathbf{A} \subseteq \mathbf{B}$  defines a function  $f_{\mathbf{A}} : \mathbf{B} \rightarrow \mathbf{2}$  by setting

$$f_{\mathbf{A}}(y) = \begin{cases} 1 & \text{if } y \in \mathbf{A}, \\ 0 & \text{elsewhere.} \end{cases} \tag{96}$$

It can be checked that this defines a 1-to-1 correspondence between functions  $\mathbf{B} \rightarrow \mathbf{2}$  and subsets of  $\mathbf{B}$ . In other words, there is a bijection between the set  $2^{\mathbf{B}}$  and

the set  $\text{Pow } \mathbf{B}$ . So, instead of using a definition of subset that involves elements of sets, we could work with functions  $\mathbf{B} \rightarrow \mathbf{2}$ .

**Graded exercise B.3** (SubsetsAsFunctions)

Let  $\mathbf{B}$  be any set. Prove that  $2^{\mathbf{B}} \simeq \text{Pow } \mathbf{B}$ .

**Example 4.28.** Even the notions of “element of a set” and “evaluation of a function at an element” can be described purely in terms of functions and their composition, without needing to “look inside” of the sets involved.

To show how this works, let us first define  $\mathbf{1} := \{\bullet\}$ , a singleton set whose only element is the symbol “ $\bullet$ ” (any singleton set would do; for concreteness we are fixing one and calling it  $\mathbf{1}$ ).

Now we are ready to make an interesting observation: functions  $\mathbf{1} \rightarrow \mathbf{A}$  are in 1-to-1 correspondence with the elements of  $\mathbf{A}$ . A function  $f : \mathbf{1} \rightarrow \mathbf{A}$  will have to map “ $\bullet$ ” to some element  $f(\bullet) \in \mathbf{A}$ , and since  $\mathbf{1}$  has no other elements, that is all that  $f$  does. So  $f$  “picks out” an element of  $\mathbf{A}$ . We can work with functions  $\mathbf{1} \rightarrow \mathbf{A}$  in place of elements of  $\mathbf{A}$ .

Next, let’s talk about function evaluation. Consider a function  $g : \mathbf{A} \rightarrow \mathbf{B}$ . Given an element  $x \in \mathbf{A}$ , this element will be mapped by  $g$  to an element  $g(x) \in \mathbf{B}$ . If we use, instead of  $x \in \mathbf{A}$ , the function  $f : \mathbf{1} \rightarrow \mathbf{A}$  to which it corresponds, then the element  $g(x) \in \mathbf{B}$  corresponds to the function  $f \circ g : \mathbf{1} \rightarrow \mathbf{B}$ . In other words, we can talk about evaluation of a function  $g : \mathbf{A} \rightarrow \mathbf{B}$  “at an element of  $\mathbf{A}$ ” without actually using elements, but rather just using functions and function composition.

## 4.11. Mappings

Listing 25 shows our interface for `Mapping`.

Listing 25: The `Mapping` interface.

```
class Mapping(Generic[A, B], ABC):
    @abstractmethod
    def source(self) -> Setoid[A]: ...

    @abstractmethod
    def target(self) -> Setoid[B]: ...

    @abstractmethod
    def __call__(self, a: A) -> B: ...
```

A `Mapping` has a source and a target, which, in general, are setoids.

A `Mapping` is able to take an element from the source setoid and return an element in the second setoid.

Here we use Python's syntax sugar. If an object defines a `__call__()` method, then these are *syntactically* equivalent:

```
assert myobject(x) == myobject.__call__(x)
```

Therefore, once you implement a `Mapping` you can pass it around like if it was a Python function.

We also define a subclass `FiniteMap` which is constrained to have both source and target be a `FiniteSet` rather than a `Setoid` (Listing 26).

Listing 26: The `FiniteMap` interface.

```
class FiniteMap(Generic[A, B], Mapping[A, B], ABC):
    """A finite map is a mapping between two finite sets."""

    @abstractmethod
    def source(self) -> FiniteSet[A]: ...

    @abstractmethod
    def target(self) -> FiniteSet[B]: ...
```

```
source:
  elements: [a, b, c]
target:
  elements: [1, 2]
values:
  - [a, 1]
  - [b, 2]
  - [c, 1]
```

Figure 17.: Format for representing maps.

We define a YAML representation for finite maps as in Fig. 17.

The fields `source` and `target` are two sets. These are expressions that you can pass to the set constructors you developed previously.

The field `values` is an array of pairs: a pair  $\langle x, y \rangle$  means that  $x$  maps to  $y$ .

For the data to be well-formed it is necessary that for each element in the domain, there is exactly one row for that element. If that is not the case, throw an exception `InvalidFormat`.

### Code exercise B.7 (TestFiniteMapRepresentation)

Create a function to load the data by implementing the interface in Listing 27.

Listing 27: The `FiniteMapRepresentation` interface.

```
class FiniteMapRepresentation(Serializer[FiniteMap[Any, Any]],
    ↳ FiniteMap_desc, ABC):
    pass
```

As part of the exercise, you have to implement `FiniteMap` with all its necessary methods.

## 4.12. FiniteMapOperations

Finally, we implement two simple construction: identity and composition.

### Code exercise B.8 (TestFiniteMapOperations)

Implement the interface in Listing 28.

Test using:

```
act4e-check TestFiniteMapOperations
```

**Listing 28:** The `FiniteMapOperations` interface.

```
class FiniteMapOperations(ABC):
    @abstractmethod
    def identity(self, s: FiniteSet[A]) -> FiniteMap[A, A]:
        """Returns the identity on a set."""

    @abstractmethod
    def compose(self, f: FiniteMap[E1, E2], g: FiniteMap[E2, E3]) ->
        FiniteMap[E1, E3]:
        """Compose two functions."""
```

### 4.13. Further constructions

We review here a few more things that are nice to have in our mathematical toolbox.

#### Arbitrary unions and intersections

Previously we defined the notion of union and the intersection for any two given sets  $\mathbf{A}$  and  $\mathbf{B}$ .

Now consider a set of sets – call it  $\xi$  – which might have two elements (say, sets  $\mathbf{A}$  and  $\mathbf{B}$ ), or it might have 108 elements, or it might have infinitely many elements. No matter how many sets are elements of  $\xi$ , we can also take the union of all those sets. This is denoted  $\bigcup \xi$ .

In other words,

$$\frac{x \in \bigcup \xi}{\exists \mathbf{A} \in \xi : x \in \mathbf{A}}. \quad (97)$$

Similarly we can define the intersection  $\bigcap \xi$  of all the sets which are elements of  $\xi$ :

$$\frac{x \in \bigcap \xi}{\forall \mathbf{A} \in \xi : x \in \mathbf{A}}. \quad (98)$$

This notation for arbitrary unions and intersections is related to our previous definitions for two sets  $\mathbf{A}$  and  $\mathbf{B}$  via

$$\mathbf{A} \cup \mathbf{B} = \bigcup \{\mathbf{A}, \mathbf{B}\} \quad \text{and} \quad \mathbf{A} \cap \mathbf{B} = \bigcap \{\mathbf{A}, \mathbf{B}\}. \quad (99)$$

#### Families

Informally, a family of sets is a collection of sets where each set is labeled with an index. The index allows us to refer to any given member of the family.

For example, we might want to work with a family of sets labeled with an index from the natural numbers  $\mathbb{N}$ :

$$\mathbf{A}_0, \mathbf{A}_1, \mathbf{A}_2, \dots \quad (100)$$

The set  $\mathbb{N}$  here is the *index set* – the set in which the indices 1, 2, 3, ... *etc.* live.

Now, what if we need to consider a family of sets where the indices live in the set  $\mathbb{R}$ ? Then we cannot write things as in (100), because  $\mathbb{R}$  is not a countable set. Instead, we write

$$\{\mathbf{A}_i\}_{i \in \mathbb{R}}. \quad (101)$$

Note that this is not the same thing as set of sets: it is possible here that sets in the family with different indices might be equal as sets. In other words, it might be that  $i \neq j$  but  $\mathbf{A}_i = \mathbf{A}_j$ , for some indices  $i$  and  $j$ . (It might even be that all the sets  $\mathbf{A}_i$  are in fact the same set.)

Note that one might also call (100) a sequence of sets, and recall that, formally, sequences are defined as functions whose source is the set  $\mathbb{N}$ . Our formal definition of a family of sets will have a similar flavor: a family will be defined as a function whose source is the index set of the family.

To spell this out, let  $\{\mathbf{A}_i\}_{i \in \mathbf{I}}$  be a collection of sets (with index set  $\mathbf{I}$ ). We can

consider the set of sets

$$\{\mathbf{A}_i \mid i \in \mathbf{I}\},^{\S} \quad (102)$$

and in particular also its union  $\mathbf{A} := \bigcup \{\mathbf{A}_i \mid i \in \mathbf{I}\}$ . Now, our formal definition of a family  $\{\mathbf{A}_i\}_{i \in \mathbf{I}}$  of sets is that it is a function

$$f : \mathbf{I} \rightarrow \text{Pow } \mathbf{A} \quad (103)$$

such that  $f(i) = \mathbf{A}_i$  for all  $i \in \mathbf{I}$ .

**Remark 4.29.** To simplify notation, for unions of families of sets we usually write  $\bigcup_{i \in \mathbf{I}} \mathbf{A}_i$  instead of  $\bigcup \{\mathbf{A}_i \mid i \in \mathbf{I}\}$ , and similarly for intersections.

## Arbitrary cartesian products and disjoint unions

We have seen that, given sets  $\mathbf{A}$  and  $\mathbf{B}$ , we can form their cartesian product  $\mathbf{A} \times \mathbf{B}$  and disjoint union  $\mathbf{A} + \mathbf{B}$ . And similarly for any finite number of sets.

What about if we start with a family  $\{\mathbf{A}_i\}_{i \in \mathbf{I}}$  of sets – can we still take their cartesian product or disjoint union? The answer is yes.

For the cartesian product of families we use the symbol  $\prod$  instead of the infix notation  $\times$ , and for disjoint unions of families we use  $\sum$  instead of the infix notation  $+$ . (The upper case greek letter sigma  $\sum$  stands for “sum”, the upper case greek letter pi  $\prod$  stands for “product”.)

We define the cartesian product of a family  $\{\mathbf{A}_i\}_{i \in \mathbf{I}}$  as

$$\prod_{i \in \mathbf{I}} \mathbf{A}_i := \{f : \mathbf{I} \rightarrow \bigcup_{i \in \mathbf{I}} \mathbf{A}_i \mid f(i) \in \mathbf{A}_i\}. \quad (104)$$

And for the disjoint union of a family  $\{\mathbf{A}_i\}_{i \in \mathbf{I}}$  we set

$$\sum_{i \in \mathbf{I}} \mathbf{A}_i := \bigcup_{i \in \mathbf{I}} \mathbf{A}_i \times \{i\}. \quad (105)$$

<sup>§</sup> For those familiar with ZFC set theory: the axiom of replacement ensures that this is a set.








## 5. Relations

This chapter describes *relations*, which are a generalization of functions.

5.1 Distribution networks . . . . .	76
5.2 Relations . . . . .	79
5.3 Code exercises - Relation representation . . . . .	80
5.4 Composing relations . . . . .	81
5.5 Code exercises - Composition . . . . .	83
5.6 Relations and functions . . . . .	84
5.7 Properties of relations . . . . .	87
5.8 Code exercises - Relation properties . . . . .	88
5.9 Transpose of a relation . . . . .	89
5.10 Code exercises - Transpose . . . . .	90
5.11 Endorelations . . . . .	91
5.12 Code exercises - Endorelation properties . . . . .	93
5.13 Transitivity . . . . .	94
5.14 Code exercises - Transitivity . . . . .	95
5.15 Equivalence relations . . . . .	96




Watch: *Distribution networks* (7 minutes).



**Distribution networks**

Power Plants	High Voltage/Nodes	Low Voltage/Nodes	Consumers
Plant 1	HVN 1	LVN 1	C1
Plant 2	HVN 2	LVN 2	C2
	HVN 3	LVN 3	C3
	HVN 4	LVN 4	C4
	HVN 5	LVN 5	C5
	HVN 6	LVN 6	C6
	LVN 7	LVN 7	C7

**transmission, and distribution**



High voltage nodes

Low voltage nodes

Consumers

### 5.1. Distribution networks

Consider the type of networks that arise for example in the context of electrical power grids. In a simplified model for a certain region or country, we may have the following kinds of components: power plants (places where electrical power is produced), high voltage transmission lines and nodes, transformers stations, low voltage transmission lines and nodes, and consumers, such as homes and businesses. The situation is depicted in Fig. 1.

To model the connectivity between the components of the power grid, we now draw arrows between components that are connected. We set the direction of the arrows to flow from energy production, via transmission components, to energy consumption, as depicted in Fig. 2.

A possible question one asks about such a power distribution network is: which consumers are serviced by which power sources? For example, power sources such as a solar power plant may fluctuate due to weather conditions, while other power sources, such as a nuclear power plant, may shut down every once in a while due to maintenance work. To see which consumers are connected to which power plants, we can follow “connectivity paths” traced by sequences of arrows, as in Fig. 3. There, two possible connectivity paths are depicted (in red and orange, respectively).

We also will want to know the overall connectivity structure of transmission lines. For example, some lines may go down during a storm, and we want to ensure enough redundancy in our system. In addition to the connections modeled in Fig. 2, we can also include, for example, information about the connectivity of high voltage nodes among themselves, as in Fig. 4.

Power Plants	High Voltage Nodes	Low Voltage Nodes	Consumers
Plant 1	HVN 1	LVN 1	C1
	HVN 2	LVN 2	C2
Plant 2	HVN 3	LVN 3	C3
	HVN 4	LVN 4	C4
Plant 3	HVN 5	LVN 5	C5
		LVN 6	C6
		LVN 7	

Figure 1.: Components of electrical power grids.

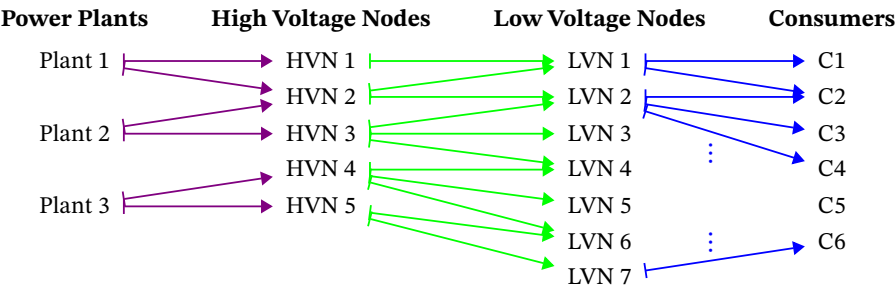


Figure 2.: Connectivity between components in electric power grids.

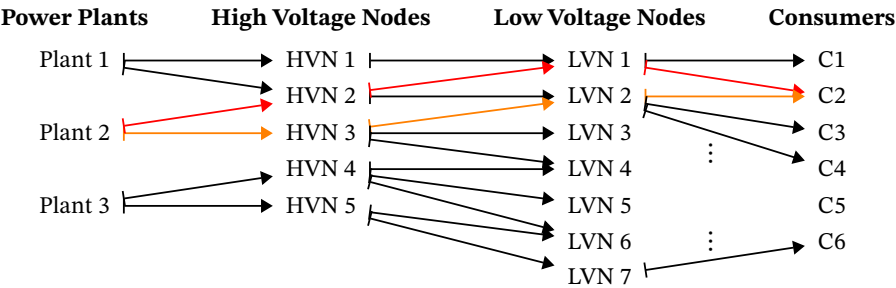


Figure 3.: Connection between consumers and power plants.

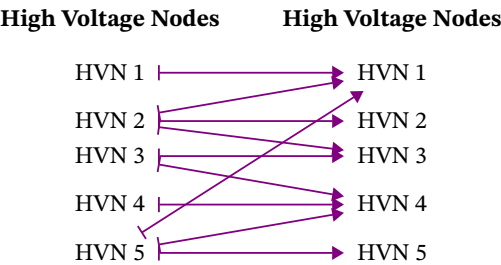
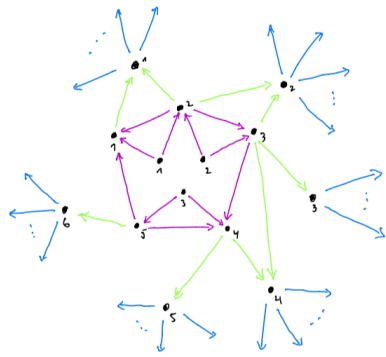
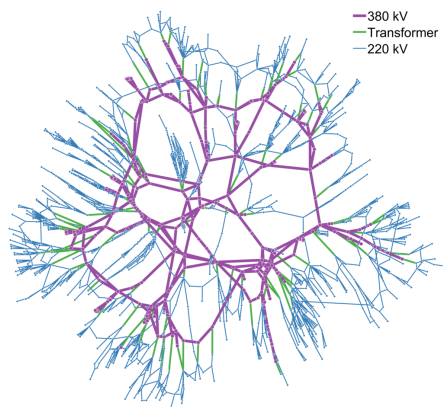


Figure 4.: Connectivity between high voltage nodes.



**Figure 5.:** Alternative visualization for connectivity.



**Figure 6.:** A schematic view of a power grid.

The information encoded in Fig. 4 and Fig. 2 can also be displayed as a single graph, see Figs. 4 and 5.

If we ignore the directionality of the arrows, this is analogous to a depiction of type shown in Fig. 6, which is a schema of a power grid [3]\*.

\* See [https://en.wikipedia.org/wiki/Electrical\\_grid](https://en.wikipedia.org/wiki/Electrical_grid)

## 5.2. Relations

A basic mathematical notion which underlies the above discussion is that of a **binary relation**.

### Definition 5.1 (Binary relation)

A **binary relation**  $R$  from a set  $A$  to a set  $B$  is a subset of the cartesian product  $A \times B$ :

$$R \subseteq A \times B. \quad (1)$$

We will often drop the word “binary” and simply use the name “relation”.

We also write

$$R : A \rightarrow B \quad (2)$$

to indicate a relation from  $A$  to  $B$ . ( $A$  is the source, and  $B$  is the target).

**Example 5.2.** Let  $A = \{\text{🍪}, \text{🍷}, \text{🍷}\}$  and  $B = \{\text{🍺}, \text{🍺}, \text{🍺}, \text{🍺}\}$ . An example of a relation is the subset

$$R = \{\langle \text{🍪}, \text{🍺} \rangle, \langle \text{🍷}, \text{🍺} \rangle, \langle \text{🍷}, \text{🍺} \rangle\} \subseteq A \times B. \quad (3)$$

If  $A$  and  $B$  are finite sets, we can depict a relation  $R : A \rightarrow B$  graphically as in Fig. 7. For each element  $\langle x, y \rangle \in A \times B$ , we draw an arrow from  $x$  to  $y$  if and only if  $\langle x, y \rangle \in R$ .

We can also depict this relation graphically as a subset of  $A \times B$  in a “coordinate system way”, as in Fig. 8.

The shaded area is the subset  $R$  defining the relation.

**Remark 5.3** (Notation for relations). From now on we will also use the following notation, where we write

$$xRy := \langle x, y \rangle \in R \quad (4)$$

instead of writing  $\langle x, y \rangle \in R$ .

**Exercise 9.** Given an arbitrary set  $B$ , does there always exist a relation  $B \rightarrow \emptyset$ ?

*See solution on page 99.*

**Exercise 10.** Given an arbitrary set  $A$ , does there always exist a relation  $A \rightarrow \emptyset$ ?

*See solution on page 99.*

Watch: *Definition of relation* (5 minutes).

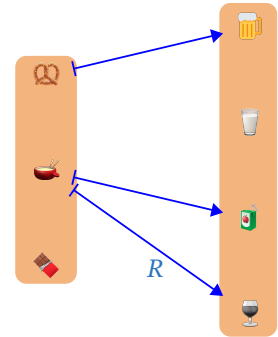
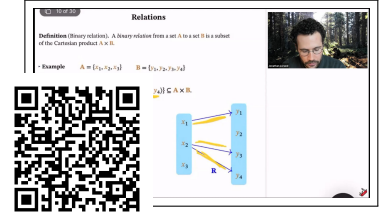


Figure 7.



Figure 8.: Relations visualized in “coordinate systems”.

### Graded exercise B.4 (VisualizeLeqRelation)

Let  $A = B = \{1, 2, 3, 4\}$  and consider the relation  $R : A \rightarrow B$  defined by

$$R = \{\langle x, y \rangle \in A \times B \mid x \leq y\}. \quad (5)$$

Visualize the relation  $R$  via the method in Fig. 7 and Fig. 8 each.

### 5.3. Relation representation

#### Interface

We define a **Relation** as an object that has a source and target **Setoid**, and has a mapping that is a function  $S \times T \rightarrow \mathbf{Bool}$ . The function tells us whether two objects are related.

Listing 29: The **Relation** interface.

```
class Relation(Generic[E1, E2], ABC):
    @abstractmethod
    def source(self) -> Setoid[E1]: ...

    @abstractmethod
    def target(self) -> Setoid[E2]: ...

    @abstractmethod
    def holds(self, e1: E1, e2: E2) -> bool: ...
```

A finite relation **FiniteRelation** is a relation between two finite sets.

Listing 30: The **FiniteRelation** interface.

```
class FiniteRelation(Generic[E1, E2], Relation[E1, E2], ABC):
    @abstractmethod
    def source(self) -> FiniteSet[E1]: ...

    @abstractmethod
    def target(self) -> FiniteSet[E2]: ...
```

The format is shown in Fig. 9.

#### Representation

```
source:
  elements: [a, b, c, d]
target:
  elements: [1, 2, 3, 4]
values:
- [a, 1]
- [a, 2]
- [c, 3]
```

Figure 9.

#### Code exercise B.9 (TestFiniteRelationRepresentation)

Create functions to load and save the data.

Implement the interface in Listing 31.

Test using:

```
act4e-check TestFiniteRelationRepresentation
```

Listing 31: The **FiniteRelationRepresentation** interface.

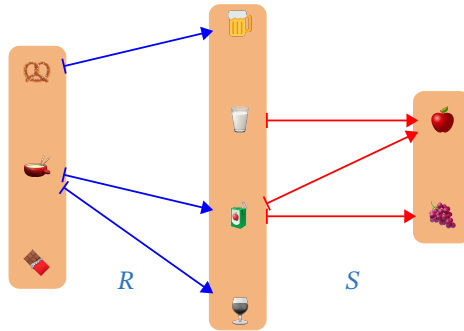
```
class FiniteRelationRepresentation(ABC):
    @abstractmethod
    def load(self, h: IOHelper, data: FiniteRelation_desc) ->
        FiniteRelation[Any, Any]:
        """Load a finite set from given YAML data"""

    @abstractmethod
    def save(self, h: IOHelper, f: FiniteRelation[Any, Any]) ->
        FiniteRelation_desc:
        """Load a finite set from given YAML data"""
```

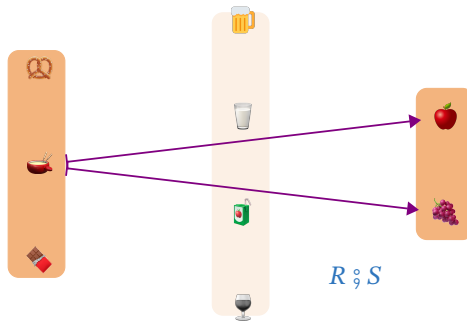
## 5.4. Composing relations

The visualization in Fig. 7 hints at the fact that we can compose relations if the target of the one is the source of the other.

To illustrate composition of relations, consider a simple example, involving sets  $\mathbf{A}$ ,  $\mathbf{B}$ , and  $\mathbf{C}$ , and relations  $R : \mathbf{A} \rightarrow \mathbf{B}$  and  $S : \mathbf{B} \rightarrow \mathbf{C}$ , as depicted graphically below in Fig. 10a.








(a) Relations compatible for composition.



(b) Composition of relations.

The composite relation  $R ; S : \mathbf{A} \rightarrow \mathbf{C}$  is defined to be such that  $x(R ; S)z$  if and only if there exists some  $y \in \mathbf{B}$  such that  $xRy$  and  $ySz$ . Graphically this means that for  $\langle x, z \rangle$  to be an element of the relation  $R ; S$ , the elements  $x$  and  $z$  need to be connected by at least one sequence of two arrows such that the target of the first arrow is the source of the second.

For example, in Fig. 10a, there is an arrow from  to , and from there on to , and therefore, in the composition  $R ; S$  depicted in Fig. 10b, there is an arrow from  to .

### Definition 5.4 (Relation composition)

Given relations  $R : \mathbf{A} \rightarrow \mathbf{B}$ ,  $S : \mathbf{B} \rightarrow \mathbf{C}$ , their composition is the relation

$$R ; S := \{\langle x, z \rangle \in \mathbf{A} \times \mathbf{C} \mid \exists y \in \mathbf{B} : (xRy) \wedge (ySz)\}. \quad (6)$$

### Graded exercise B.5 (Composing Relations)

- Let  $\mathbf{A} = \mathbb{N}$ ,  $\mathbf{B} = \mathbb{Z}$ , and  $\mathbf{C} = \mathbb{R}$ . Consider the relation  $f : \mathbf{A} \rightarrow \mathbf{B}$  with

$$f = \{\langle x, y \rangle \in \mathbb{N} \times \mathbb{Z} \mid x = y^2\}, \quad (7)$$

Watch: *Composing relations* (3 minutes).

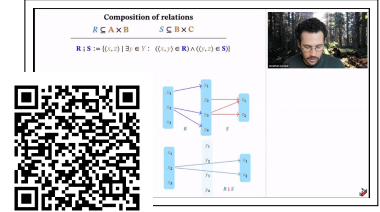


Figure 10.: Illustrations for relations composition.

and consider the relation  $g : \mathbf{B} \rightarrow \mathbf{C}$  with

$$g = \{\langle y, z \rangle \in \mathbb{Z} \times \mathbb{R} \mid y = 2z\}. \quad (8)$$

Calculate the relation  $f \circ g : \mathbf{A} \rightarrow \mathbf{B}$ .

2. Let  $\mathbf{A} = \mathbb{N}$ ,  $\mathbf{B} = \mathbb{Z} \times \mathbb{Z}$ , and  $\mathbf{C} = \mathbb{R}$ . Consider the relation  $f : \mathbf{A} \rightarrow \mathbf{B}$  with

$$f = \{\langle x, \langle y_1, y_2 \rangle \rangle \in \mathbb{N} \times (\mathbb{Z} \times \mathbb{Z}) \mid x = y_1 - y_2\} \quad (9)$$

and consider the relation  $g : \mathbf{B} \rightarrow \mathbf{C}$  with

$$g = \{\langle \langle y_1, y_2 \rangle, z \rangle \in (\mathbb{Z} \times \mathbb{Z}) \times \mathbb{R} \mid y_2 z = y_1\}. \quad (10)$$

Calculate the relation  $f \circ g : \mathbf{A} \rightarrow \mathbf{B}$ .

## 5.5. Composition

### Code exercise B.10 (TestFiniteRelationCompose)

Implement the composition of two relations.

Implement the interface in Listing 32.

Test using:

```
act4e-check TestFiniteRelationCompose
```

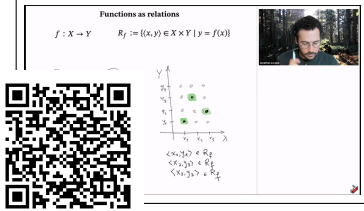
**Listing 32:** The `FiniteRelationCompose` interface.

```
class FiniteRelationCompose(ABC):
    @abstractmethod
    def compose(self, fr: FiniteRelation[E1, E2], fr2: FiniteRelation[E2, E3])
        -> FiniteRelation[E1, E3]:
        """Compose two relations"""
```

Note here how Python typing is helping in expressing the intention. The first relation is from `E1` to `E2`; the second relation is from `E2` to `E3`; consequently, the result is a relation from `E1` to `E3`.



Watch: *Functions and relations* (6 minutes).



## 5.6. Relations and functions

Every function between sets can be thought as a relation: this was the basis of our formal definition of function, Def. 4.13. Let us restate that definition once again here, using notation that we have developed since.

### Definition 5.5 (Function as a special type of relation)

Let  $\mathbf{A}$  and  $\mathbf{B}$  be sets. A relation  $R \subseteq \mathbf{A} \times \mathbf{B}$  is a *function* if it satisfies the following two conditions:

1. for all  $x \in \mathbf{A}$  there exists an element  $y \in \mathbf{B}$  such that  $x R y$ ;
2. for all  $x, y_1, y_2$ , this holds:

$$\frac{x R y_1 \quad x R y_2}{y_1 = y_2}. \quad (11)$$

Although we will mostly continue to think about functions in the “usual” way (as opposed to the perspective of Def. 5.5), it is illuminating – both for understanding relations and functions – to study the relationships between the two points of view.

### From functions to relations

Recall how to go from viewing a function in the “usual” way to viewing it as a relation, as in Def. 5.5.

As an illustration, consider the sets  $\mathbf{A} = \{\text{🍪}, \text{🍷}, \text{🍷}\}$  and  $\mathbf{B} = \{\text{🍺}, \text{🍺}, \text{🍺}, \text{🍺}\}$ , and the function  $f : \mathbf{A} \rightarrow \mathbf{B}$  defined (in the “usual” way) by

$$f(\text{🍪}) = \text{🍺}, \quad f(\text{🍷}) = \text{🍺}, \quad f(\text{🍷}) = \text{🍺}. \quad (12)$$

This way of specifying the function  $f$  may be depicted graphically as in Fig. 11.

The relation that this function defines, in the sense of Def. 5.5, is

$$\{\langle \text{🍪}, \text{🍺} \rangle, \langle \text{🍷}, \text{🍺} \rangle, \langle \text{🍷}, \text{🍺} \rangle\} \subseteq \mathbf{A} \times \mathbf{B}. \quad (13)$$

This relation (13) is what is often called the *graph* of  $f$ . That is, it is the set of tuples in  $\mathbf{A} \times \mathbf{B}$  which are a pairing of an element of the source set  $\mathbf{A}$  with the element which is its image under  $f$ . In Fig. 12, the graph of (13) is visualized by highlighting the elements of the graph among all the elements of  $\mathbf{A} \times \mathbf{B}$ .

In general, any function  $f : \mathbf{A} \rightarrow \mathbf{B}$  corresponds to the relation

$$\{\langle x, y \rangle \in \mathbf{A} \times \mathbf{B} \mid y = f(x)\}. \quad (14)$$

### From relations to functions

Let’s start now with a relation  $R \subseteq \mathbf{A} \times \mathbf{B}$  satisfying the conditions of Def. 5.5 and see how this corresponds to a function  $f_R : \mathbf{A} \rightarrow \mathbf{B}$  in the “usual” sense.

Choose an arbitrary  $x \in \mathbf{A}$ . According to point 1 in Def. 5.5, there exists a  $y \in \mathbf{B}$  such that  $x R y$ . Choose such a  $y$ , and call it  $f_R(x)$ . This gives us recipe to get from any  $x$  to a  $y$ . But given a specific  $x \in \mathbf{A}$ , what if we choose  $y$  differently each time we apply the recipe? Point 2 guarantees that this can’t happen: it says that the element  $f_R(x)$  that we associate to a given  $x \in \mathbf{A}$  is in fact uniquely determined by that  $x$ . Put another way, the condition 2 says: if  $f_R(x_1) \neq f_R(x_2)$ , then  $x_1 \neq x_2$ .

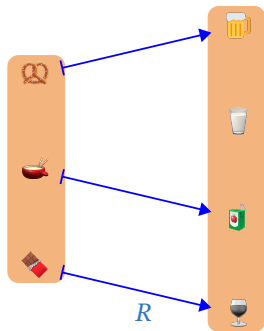


Figure 11.: Visualization of the function (13).



Figure 12.: The graph of the function (13).

**Remark 5.6.** Of course, not every relation corresponds to a function – namely precisely those that do not satisfy the conditions in Def. 5.5. For example, the relation in Fig. 7 is not a function. In fact, it does not satisfy *either* of the conditions in Def. 5.5.

## Identity relations

We have already discussed how, for any set  $A$ , there is always an identity function

$$\begin{aligned} \text{id}_A : A &\rightarrow A, \\ y &\mapsto y. \end{aligned} \tag{15}$$

which “does nothing”. If we turn such functions into relations, we call the result *identity relations*.

### Definition 5.7

Let  $A$  be any set. The *identity relation* on  $A$  is

$$\text{id}_A = \{\langle x, y \rangle \in A \times A \mid x = y\}. \tag{16}$$

## Composing functions

If we define functions as special kinds of relations, how is relation composition related to the “usual” way of composing of functions? The answer is that these two apparently different ways of composing functions actually give the same result.

**Lemma 5.8.** Let  $R \subseteq A \times B$  and  $S \subseteq B \times C$  be relations which are functions. Then their composition  $R \circ S \subseteq A \times C$  is again a function, and it corresponds to the “usual” composition of the functions corresponding to  $R$  and  $S$ .

*Proof.* First let us check that when  $R$  and  $S$  are composed as relations, the result is again a function. For this we check that  $R \circ S$  satisfies the two conditions stated in Def. 5.5.

1. Choose an arbitrary  $x \in A$ . We need to show that there exists  $z \in C$  such that  $x R \circ S z$ . Since  $R$  is a function, there exists  $y \in B$  such that  $x R y$ . Choose such a  $y \in B$ . Then, because  $S$  is a function, there exists  $z \in C$  such that  $y S z$ . By the definition of composition of relations, we see that  $z$  is such that  $x R \circ S z$ .
2. Let  $x_1 R \circ S z_1, x_2 R \circ S z_2$ . We need to show that if  $x_1 = x_2$ , then  $z_1 = z_2$ . So suppose  $x_1 = x_2$ . Since  $x_1 R \circ S z_1, x_2 R \circ S z_2$ , there exist  $y_1, y_2 \in B$  such that, respectively,

$$x_1 R y_1 \wedge y_1 S z_1, \tag{17}$$

$$x_2 R y_2 \wedge y_2 S z_2. \tag{18}$$

Since  $x_1 = x_2$  and  $R$  is a function, we conclude that  $y_1 = y_2$  must hold. Now, since  $S$  is also a function, this implies that  $z_1 = z_2$ , which is what was to be shown.

Second let us check that relation composition of functions gives the same result as the “usual” composition of functions. Let  $f_R$  and  $g_S$  denote the relations  $R$  and  $S$  when we are thinking of them in the “usual” way of thinking about functions. Our goal is to show that  $f_R \circ g_S$  corresponds to  $R \circ S$ ; in other words, that the latter is the graph of former.

Suppose first that  $\langle x, z \rangle$  is in the graph of  $f_R \circ g_S$ , so  $z = (f_R \circ g_S)(x)$ . In particular  $z = g_S(f_R(x))$ , which means there exists  $y = f_R(x) \in \mathbf{B}$  such that  $\langle x, y \rangle \in R$  and  $\langle y, z \rangle \in S$ . This implies that  $\langle x, z \rangle \in R \circ S$ .

Conversely, suppose  $\langle x, z \rangle \in R \circ S$ . By the definition of relation composition there must exist  $y \in \mathbf{B}$  such that  $\langle x, y \rangle \in R$  and  $\langle y, z \rangle \in S$ , which means  $y = f_R(x)$  and  $z = g_S(y)$ . Thus,  $z = g_S(f_R(x))$ .  $\square$

## Relations via functions

Though not every relation is a function, we can however think about relations *in terms of* functions. Here are three ways:

1. We can think of a relation  $R : \mathbf{A} \rightarrow \mathbf{B}$  as a function  $\mathbf{A} \times \mathbf{B} \rightarrow \mathbf{Bool}$ .

Given  $R$  we can define a function  $\phi_R : \mathbf{A} \times \mathbf{B} \rightarrow \{\perp, \top\}$  from it by setting

$$\phi_R(\langle x, y \rangle) = \begin{cases} \top & \text{if } x R y, \\ \perp & \text{otherwise.} \end{cases} \quad (19)$$

Conversely, given a function  $\phi : \mathbf{A} \times \mathbf{B} \rightarrow \{\perp, \top\}$  we can define a relation  $R_\phi \subseteq \mathbf{A} \times \mathbf{B}$  from it by setting

$$R_\phi = \{\langle x, y \rangle \in \mathbf{A} \times \mathbf{B} \mid \phi_R(\langle x, y \rangle) = \top\}. \quad (20)$$

These two constructions are inverse to one-another.

2. We can think of a relation  $R : \mathbf{A} \rightarrow \mathbf{B}$  as a function  $\mathbf{A} \rightarrow \mathbf{Pow}(\mathbf{B})$ .

Given  $R$  we can define a function  $\hat{\phi}_R : \mathbf{A} \rightarrow \mathbf{Pow}(\mathbf{B})$  via

$$\hat{\phi}_R(x) = \{y \in \mathbf{B} \mid x R y\}. \quad (21)$$

Conversely, given a function  $\hat{\phi} : \mathbf{A} \rightarrow \mathbf{Pow}(\mathbf{B})$ , we can define

$$R_{\hat{\phi}} = \{\langle x, y \rangle \in \mathbf{A} \times \mathbf{B} \mid y \in \hat{\phi}_R(x)\}. \quad (22)$$

These two constructions are inverse to one another, too.

3. We can think of a relation  $R \subseteq \mathbf{A} \times \mathbf{B}$  as a function  $\mathbf{B} \rightarrow \mathbf{Pow}(\mathbf{A})$ .

Given  $R$  we can define a function  $\check{\phi}_R : \mathbf{B} \rightarrow \mathbf{Pow}(\mathbf{A})$  via

$$\check{\phi}_R(y) = \{x \in \mathbf{A} \mid x R y\}. \quad (23)$$

Conversely, given a function  $\check{\phi} : \mathbf{B} \rightarrow \mathbf{Pow}(\mathbf{A})$ , we can define

$$R_{\check{\phi}} = \{\langle x, y \rangle \in \mathbf{A} \times \mathbf{B} \mid x \in \check{\phi}_R(y)\}. \quad (24)$$

These two constructions are *also* inverse to one another.

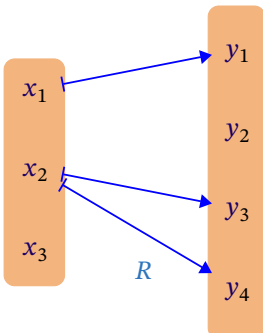


Figure 13.

### Graded exercise B.6 (Rel3Functions)

For the relation  $R$  illustrated in Fig. 13, write out the three functions that describe it, respectively, in the three ways outlined in Section 5.6.

## 5.7. Properties of relations

We have seen that relations generalize functions – every function defines a relation, via its graph, but not every relation comes from a function in this way. Many notions that we are familiar with for functions also generalize to relations. Here are a few.

### Definition 5.9 (Properties of a relation)

We say that a relation  $R : \mathbf{A} \rightarrow \mathbf{B}$  is:

1. *Injective* if

$$\frac{xRy \quad zRy}{x = z}; \quad (25)$$

2. *Single-valued* if

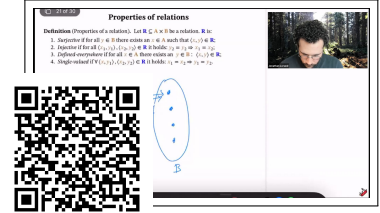
$$\frac{xRy \quad xRu}{y = u}; \quad (26)$$

3. *Surjective* if for all  $y \in \mathbf{B}$  there exists an element  $x \in \mathbf{A}$  such that  $xRy$ ;

4. *Everywhere-defined* if for all  $x \in \mathbf{A}$  there exists an element  $y \in \mathbf{B}$  such that  $xRy$ .

**Example 5.10.** The relation depicted in Fig. 7 is injective but not surjective. It is not single-valued, nor everywhere-defined.

Watch: *Properties of relations* (9 minutes).



## 5.8. ○ Relation properties

### Code exercise B.11 (TestFiniteRelationProperties)

Check that an endorelation has the properties mentioned in Def. 5.9.

Implement the interface in Listing 33.

Test using:

```
act4e-check TestFiniteRelationProperties
```

**Listing 33:** The `FiniteRelationProperties` interface.

```
class FiniteRelationProperties(ABC):
    @abstractmethod
    def is_surjective(self, fr: FiniteRelation[E1, E2]) -> bool:
        """Return True if the relation is surjective."""

    @abstractmethod
    def is_injective(self, fr: FiniteRelation[E1, E2]) -> bool:
        """Return True if the relation is injective."""

    @abstractmethod
    def is_defined_everywhere(self, fr: FiniteRelation[E1, E2]) -> bool:
        """Return True if the relation is defined everywhere."""

    @abstractmethod
    def is_single_valued(self, fr: FiniteRelation[E1, E2]) -> bool:
        """Return True if the relation is single-valued"""
```

## 5.9. Transpose of a relation

One can notice a certain duality in the properties listed in Def. 5.9. This is made more precise through the following definition.

### Definition 5.11 (Transpose of a relation)

Let  $R : \mathbf{A} \rightarrow \mathbf{B}$  be a relation. We define its *transpose* (or *opposite*, or *reverse*)  $R^T : \mathbf{B} \rightarrow \mathbf{A}$  as follows:

$$\frac{x R y}{y R^T x}. \quad (27)$$

**Remark 5.12.** Here are some useful properties of a relation  $R : \mathbf{A} \rightarrow \mathbf{B}$  and its opposite  $R^T : \mathbf{B} \rightarrow \mathbf{A}$ :

1.  $(R^T)^T = R$ ;
2.  $R$  is everywhere-defined if and only if  $R^T$  is surjective;
3.  $R$  is single-valued if and only if  $R^T$  is injective.
4.  $R$  is everywhere-defined if and only if  $\text{id}_{\mathbf{A}} \subseteq R \circ R^T$ ;
5.  $R$  is single-valued if and only if  $R^T \circ R \subseteq \text{id}_{\mathbf{B}}$ .

**Remark 5.13.** The aforementioned duality can be seen by “reading the relations (arrows) backwards” (Figs. 14 and 15).

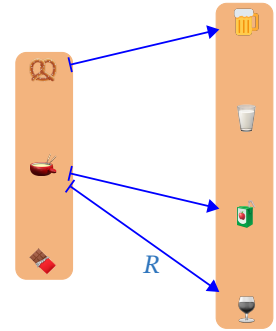


Figure 14.

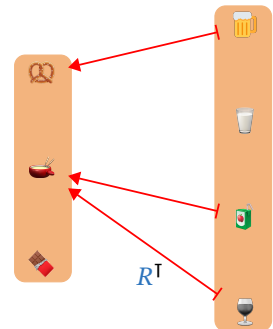
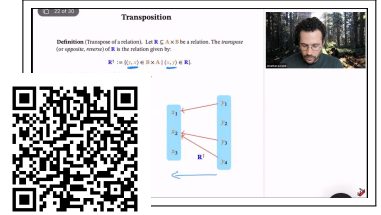


Figure 15.

Watch: *Transposition of relations* (2 minutes).



### Graded exercise B.7 (RelProperties)

Provide a proof of each of the properties listed in Remark 5.12.

## 5.10. ○ Transpose

### Code exercise B.12 (TestFiniteRelationOperations)

Compute the transpose relation.

Implement the interface in Listing 34.

Test using:

```
act4e-check TestFiniteRelationOperations
```

**Listing 34:** The `FiniteRelationOperations` interface.

```
class FiniteRelationOperations(ABC):
    @abstractmethod
    def transpose(self, fr: FiniteRelation[E1, E2]) -> FiniteRelation[E2, E1]:
        """Create the transposed of a relation"""

    @abstractmethod
    def as_relation(self, f: FiniteMap[E1, E2]) -> FiniteRelation[E1, E2]:
        """Re-writes a Finite Map as a relation."""
```

## 5.11. Endorelations

### Definition 5.14 (Endorelation)

An *endorelation* on a set  $A$  is a relation  $R : A \rightarrow A$ .

**Example 5.15.** “Equality” on a set  $A$  is an endorelation  $=_A$  of the form

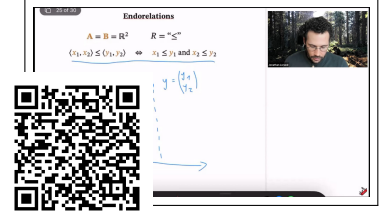
$$=_A := \{\langle x, y \rangle \in A \times A \mid x = y\}. \quad (28)$$

**Example 5.16.** Take  $A = \mathbb{N}$ . The relation “less than or equal” is an endorelation of the form

$$\leq := \{\langle x, y \rangle \in \mathbb{N} \times \mathbb{N} \mid x \leq y\}. \quad (29)$$

**Example 5.17.** The relation depicted in Fig. 4 is an endorelation between the set of high voltage nodes.

Watch: *Endorelations* (12 minutes).



### Definition 5.18 (Symmetry, asymmetry, and antisymmetry)

An endorelation  $R : A \rightarrow A$  is *symmetric* if

$$\frac{xRy}{yRx}, \quad (30)$$

is *asymmetric* if

$$\frac{xRy \quad yRx}{\perp}, \quad (31)$$

and is *antisymmetric* if

$$\frac{xRy \quad yRx}{x = y}. \quad (32)$$

### Definition 5.19 (Reflexivity and irreflexivity of endorelations)

An endorelation  $R : A \rightarrow A$  is *reflexive* if

$$\frac{\top}{xRx}, \quad (33)$$

and is *irreflexive* if

$$\frac{xRx}{\perp}. \quad (34)$$

**Table 5.1.:** Summary of endorelation properties.

Reflexive	Total	Symmetric	Transitive
$\frac{\top}{xRx}$	$\frac{\top}{xRy \vee yRx}$	$\frac{xRy}{yRx}$	$\frac{xRy \quad yRz}{xRz}$
Irreflexive	Asymmetric	Antisymmetric	
$\frac{xRx}{\perp}$	$\frac{xRy \quad yRx}{\perp}$	$\frac{xRy \quad yRx}{x = y}$	



**Definition 5.20** (Totality of endorelations)

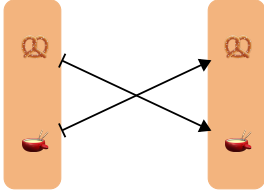
An endorelation  $R : \mathbf{A} \rightarrow \mathbf{A}$  is *total* if

$$\frac{\top}{(xRy) \vee (yRx)} . \quad (35)$$

**Example 5.21.** The relation “less than or equal” on  $\mathbb{N}$  is not symmetric. It is reflexive since  $n \leq n \ \forall n \in \mathbb{N}$ , and it is transitive since  $l \leq m$  and  $m \leq n$  implies  $l \leq n$ .

**Example 5.22.** The relation depicted in Fig. 4 on page 77 is reflexive (each node is connected to itself).

**Example 5.23.** The endorelation depicted in Fig. 16 is a symmetric relation on  $\mathbf{A} = \{\mathfrak{Q}, \text{🍒}\}$ .



**Figure 16.:** Example of symmetric endorelation.

## 5.12. Endorelation properties

### Code exercise B.13 (TestFiniteEndorelationProperties)

Check that an endorelation has the properties mentioned in Section 5.11. Implement the interface in Listing 35.

Test using:

```
act4e-check TestFiniteEndorelationProperties
```

Listing 35: The `FiniteEndorelationProperties` interface.

```
class FiniteEndorelationProperties(ABC):
    @abstractmethod
    def is_reflexive(self, fr: FiniteRelation[E, E]) -> bool:
        """Return True if the relation is reflexive."""

    @abstractmethod
    def is_irreflexive(self, fr: FiniteRelation[E, E]) -> bool:
        """Return True if the relation is irreflexive."""

    @abstractmethod
    def is_transitive(self, fr: FiniteRelation[E, E]) -> bool:
        """Return True if the relation is transitive."""

    @abstractmethod
    def is_symmetric(self, fr: FiniteRelation[E, E]) -> bool:
        """Return True if the relation is symmetric"""

    @abstractmethod
    def is_antisymmetric(self, fr: FiniteRelation[E, E]) -> bool:
        """Return True if the relation is antisymmetric"""

    @abstractmethod
    def is_asymmetric(self, fr: FiniteRelation[E, E]) -> bool:
        """Return True if the relation is asymmetric"""
```

### 5.13. Transitivity

**Definition 5.24** (Transitivity of endorelations)

An endorelation  $R : \mathbf{A} \rightarrow \mathbf{A}$  is *transitive* if

$$\frac{xRy \quad yRz}{xRz}. \quad (36)$$

**Example 5.25.** The relation “has the same birthday as” is transitive because if Anna has the same birthday as Bob, and Bob has the same birthday as Clara, then Anna has the same birthday as Clara.

**Definition 5.26** (Transitive closure)

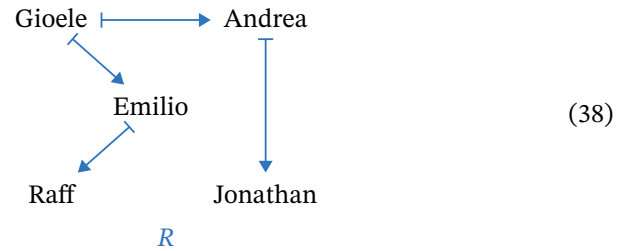
The *transitive closure* of an endorelation  $R$  on a set  $\mathbf{A}$  is the unique relation  $R^+$  on  $\mathbf{A}$  satisfying the following conditions:

1.  $R \subseteq R^+$ ;
2.  $R^+$  is transitive;
3. if  $S$  is a relation on  $\mathbf{A}$  that satisfies the previous two points, then  $R^+ \subseteq S$ .

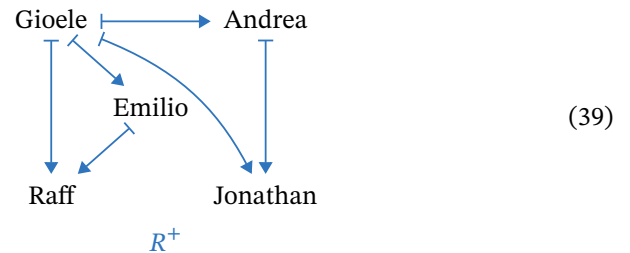
**Example 5.27.** Consider a relation  $R$  on a set of people

$$\mathbf{A} = \{\text{Gioele, Andrea, Jonathan, Emilio, Raff}\}, \quad (37)$$

which describes who invites which friend to a party:



In other words, Gioele invites Andrea and Emilio, Andrea invites Jonathan, and Emilio invites Raff. The transitive closure  $R^+$  of  $R$  describes all invitations resulting from transitivity.



In particular, Gioele invites Jonathan and Raff as well, due to the fact that Andrea invites Jonathan, and Emilio invites Raff.

## 5.14. ○ Transitivity

### Code exercise B.14 (TestFiniteEndorelationOperations)

Compute the transitive closure of an endorelation.

Implement the interface in Listing 36.

Test using:

```
act4e-check TestFiniteEndorelationOperations
```

**Listing 36:** The `FiniteEndorelationOperations` interface.

```
class FiniteEndorelationOperations(ABC):
    @abstractmethod
    def transitive_closure(self, fr: FiniteRelation[E, E]) -> FiniteRelation[E,
        ↵ E]:
        """Returns the transitive closure of a relation"""
```

*Hint:* Finding the transitive closure requires an iterative algorithm.

Suppose that you have an endorelation  $R$ . For each element  $x$ , you need to “follow the arrows” along the relation. For example, if  $x R y$  and  $x R z$ , in the transitive closure of  $x$  there are  $y, z$  as well as all the elements in the transitive closure of  $y$  and  $z$ .

This property suggests a recursive solution; however, note that there might be loops, and if you do not take appropriate measures, the code will be stuck in an infinite loop. If you choose the recursive approach, you must remember the elements that you have already seen in the recursion.

The cleanest approach is not recursive. For each element, you want to find the elements reachable from it. You can do this by iterating over “hops”. First, you add the elements that are reachable in 1 hop, and you mark those for “expansion” later. Then, you expand those elements to find the elements that are reachable in 2 hops. But do not mark for expansion those that have already been seen.

If you choose the second approach, you have created something that could later be expanded into a “search algorithm” (Dijkstra’s algorithm), which finds not only which elements are reachable from a start element, but also the minimum number of hops to get there.

Watch: *Equivalence relations* (2 minutes).



## 5.15. Equivalence relations

### Definition 5.28 (Equivalence relation)

An endorelation  $R : \mathbf{A} \rightarrow \mathbf{A}$  is an *equivalence relation* if it is symmetric, reflexive, and transitive. If  $R$  is an equivalence relation, we sometimes write  $x \sim y$  instead of  $x R y$ .

**Example 5.29.** The relation “equals” on  $\mathbb{N}$  is an equivalence relation. The relation “less than or equal” on  $\mathbb{N}$  is not.

**Example 5.30.** The relation on  $\mathbb{N}$  “differing by a multiple of 3”

$$\frac{x R y}{(x - y) \bmod 3 = 0} \quad (40)$$

is an equivalence relation. Indeed, the relation is reflexive, and symmetric. Furthermore, if  $x$  differs by a multiple of 3 from  $y$  and  $y$  differs by a multiple of 3 from  $z$ , then  $x$  differs by a multiple of 3 from  $z$  (transitivity).

**Example 5.31.** The relation “has the same birthday as” on the set of all people is an equivalence relation. It is symmetric, because if Anna has the same birthday as Bob, then Bob has the same birthday as Anna. It is reflexive because every person has the same birthday as themselves.

**Example 5.32.** Let  $f : \mathbf{A} \rightarrow \mathbf{B}$  be a function between sets. The following defines an equivalence relation  $\sim_f$ :

$$\frac{x \sim_f y}{f(x) = f(y)} \quad (41)$$

### Definition 5.33 (Partition)

A *partition* of a set  $\mathbf{A}$  is a collection  $\{\mathbf{A}_i\}_{i \in I}$  of subsets  $\mathbf{A}_i \subseteq \mathbf{A}$  such that

1.  $\mathbf{A}_i \cap \mathbf{A}_j = \emptyset \quad \forall i \neq j$ ;
2.  $\bigcup_{i \in I} \mathbf{A}_i = \mathbf{A}$ .

**Remark 5.34.** Equivalence relations are a way to group together elements of a set which we think of as “the same” in some respect. There is a one-to-one correspondence between equivalence relations on a set  $\mathbf{A}$  and partitions on  $\mathbf{A}$ .

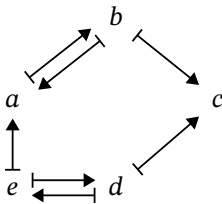


Figure 17.

**Example 5.35.** An example of partitions can be shown through information networks. An exemplary network is depicted in Fig. 17. Here, nodes represent data centers, and the arrows represent information flows. We say that data centers  $a$  and  $b$  are equivalent ( $x \sim y$ ) if and only if there is a path from  $x$  to  $y$  and a path from  $y$  to  $x$ . In Fig. 17, we have that  $a \sim b$ ,  $e \sim d$ , and also every center is equivalent with itself.

### Graded exercise B.8 (CountingEquivalenceRelations)

Let  $\mathbf{A} = \{1, 2, 3, 4\}$ . How many different equivalence relations are there on  $\mathbf{A}$ ? Explain how you found your answer.

## Solutions to selected exercises

**Solution of Exercise 3.** We start by the union operation  $\cup$ . We need to prove

$$\mathbf{A} \cup (\mathbf{B} \cup \mathbf{C}) = (\mathbf{A} \cup \mathbf{B}) \cup \mathbf{C}.$$

We have:

$$\begin{aligned} x &\in \mathbf{A} \cup (\mathbf{B} \cup \mathbf{C}) \\ \Leftrightarrow x &\in \mathbf{A} \vee (x \in \mathbf{B} \vee x \in \mathbf{C}) \\ \Leftrightarrow (x &\in \mathbf{A} \vee x \in \mathbf{B}) \vee x \in \mathbf{C} \\ \Leftrightarrow x &\in (\mathbf{A} \cup \mathbf{B}) \cup \mathbf{C}. \end{aligned}$$

We now continue with the intersection operation  $\cap$ . We need to prove

$$\mathbf{A} \cap (\mathbf{B} \cap \mathbf{C}) = (\mathbf{A} \cap \mathbf{B}) \cap \mathbf{C}.$$

We have:

$$\begin{aligned} x &\in \mathbf{A} \cap (\mathbf{B} \cap \mathbf{C}) \\ \Leftrightarrow x &\in \mathbf{A} \wedge (x \in \mathbf{B} \wedge x \in \mathbf{C}) \\ \Leftrightarrow (x &\in \mathbf{A} \wedge x \in \mathbf{B}) \wedge x \in \mathbf{C} \\ \Leftrightarrow x &\in (\mathbf{A} \cap \mathbf{B}) \cap \mathbf{C}. \end{aligned}$$

Essentially, we have used the associativity of the  $\wedge$  and  $\vee$  connectives.

**Solution of Exercise 4.** We start by the union operation  $\cup$ . We need to prove that  $\mathbf{A} \cup \mathbf{B} = \mathbf{B} \cup \mathbf{A}$ . We have:

$$\begin{aligned} x &\in \mathbf{A} \cup \mathbf{B} \\ \Leftrightarrow x &\in \mathbf{A} \vee x \in \mathbf{B} \\ \Leftrightarrow x &\in \mathbf{B} \vee x \in \mathbf{A} \\ \Leftrightarrow x &\in \mathbf{B} \cup \mathbf{A}. \end{aligned}$$

We continue with the intersection operation  $\cap$ . We need to prove that  $\mathbf{A} \cap \mathbf{B} = \mathbf{B} \cap \mathbf{A}$ . We have:

$$\begin{aligned} x &\in \mathbf{A} \cap \mathbf{B} \\ \Leftrightarrow x &\in \mathbf{A} \wedge x \in \mathbf{B} \\ \Leftrightarrow x &\in \mathbf{B} \wedge x \in \mathbf{A} \\ \Leftrightarrow x &\in \mathbf{B} \cap \mathbf{A}. \end{aligned}$$

Essentially, we have used the commutativity of the  $\wedge$  and  $\vee$  connectives.

**Solution of Exercise 5.** We have:

1. 2.
2. 4.
3. 8.
4. 1.

In general, the size of  $\text{Pow } \mathbf{A}$  is 2 to the power of the size of  $\mathbf{A}$ .

**Solution of Exercise 6.** We analyze the functions one by one.

1.  $x + 10 = y + 10$  implies  $x = y$ , therefore the function is injective. Clearly, for any real number  $y$  there is an  $x$  such that  $f(x) = y$ . Therefore the *mapa* is surjective. It follows that  $f$  is bijective.

2.  $g$  is not injective, since  $g(x) = g(-x) = x^2$ . Also, it clearly is not surjective, since for any negative real  $y$  there is no  $x$  such that  $g(x) = y$ .
3. When  $x, y \in \mathbb{N}$ ,  $x^2 = y^2$  implies  $x = y$ , meaning that  $h$  is injective.  $h$  is not surjective, since, e.g., there is no  $x \in \mathbb{N}$  such that  $h(x) = 3$ .
4. Clearly,  $k$  is injective. It is not surjective, since, e.g., there is no  $x \in \mathbb{N}$  such that  $k(x) = 2$ .

**Solution of Exercise 7.** *Per absurdum*, assume that a map  $f : \mathbf{A} \rightarrow \mathbf{B}$  possesses two different inverses  $g, h : \mathbf{B} \rightarrow \mathbf{A}$ . Following the definition of inverse, we have

$$f \circ g = f \circ h = \text{id}_{\mathbf{A}}, \quad (42)$$

and

$$g \circ f = h \circ f = \text{id}_{\mathbf{B}}. \quad (43)$$

Now, we can write

$$\begin{aligned} h &= \text{id}_{\mathbf{B}} \circ h \\ &= (g \circ f) \circ h \\ &= g \circ (f \circ h) \\ &= g \circ \text{id}_{\mathbf{A}} \\ &= g, \end{aligned} \quad (44)$$

which contradicts the initial assumption.

**Solution of Exercise 8.** We show the two directions in turn:

$$\frac{f \text{ isomorphism}}{f \text{ bijective}} \quad \text{and} \quad \frac{f \text{ bijective}}{f \text{ isomorphism}}. \quad (45)$$

Consider an isomorphism  $f : \mathbf{A} \rightarrow \mathbf{B}$  and its inverse  $g : \mathbf{B} \rightarrow \mathbf{A}$ . Take a  $y \in \mathbf{B}$  and let  $x = g(y)$ . We know that  $f(x) = f(g(y)) = (g \circ f)(y) = y$ . Therefore,  $f$  must be *surjective*. To show injectivity, consider  $x, x' \in \mathbf{A}$  such that  $f(x) = f(x')$ . Let  $y = f(x)$  and  $x'' = g(y)$ . Then, we have

$$\begin{aligned} x' &= \text{id}_{\mathbf{A}}(x') \\ &= (f \circ g)(x') \\ &= g(f(x')) \\ &= g(y) \\ &= x''. \end{aligned} \quad (46)$$

However, we also know

$$\begin{aligned} x &= \text{id}_{\mathbf{A}}(x) \\ &= (f \circ g)(x) \\ &= g(f(x)) \\ &= g(y) \\ &= x''. \end{aligned} \quad (47)$$

Therefore,  $x = x'$  and  $f$  is injective (and therefore bijective).

Now, consider a map  $f : \mathbf{A} \rightarrow \mathbf{B}$  which is bijective. One can define  $g : \mathbf{B} \rightarrow \mathbf{A}$  in the following way. Take a  $y \in \mathbf{B}$ , and since  $f$  is surjective (it is bijective), there exists a  $x \in \mathbf{A}$  such that  $f(x) = y$ . Let  $g(y) = x$ . Since  $f$  is injective,  $x$  must be unique, meaning that  $g$  is well-defined. Now we check that indeed  $g$  must be the inverse of  $f$ . Consider  $x \in \mathbf{A}$  and  $y = f(x)$ . By definition,  $g(y) = x$ , and hence  $(f \circ g)(x) = f(g(y)) = f(x) = y$ , implying  $f \circ g = \text{id}_{\mathbf{A}}$ . Similarly, take  $y \in \mathbf{B}$  and  $x = g(y)$ . Then, by definition we have  $f(x) = y$ , and hence  $(g \circ f)(y) =$

$f(g(y)) = f(x) = y$ , implying  $g \circ f = \text{id}_{\mathbf{B}}$ . Therefore,  $f$  is an isomorphism.

**Solution of Exercise 9.** Yes. Such a relation would be of the form  $R \subseteq \emptyset \times \mathbf{B} = \emptyset$ , where  $\mathbf{B}$  here is an arbitrary set. In this situation,  $R = \emptyset$  is a relation  $\emptyset \rightarrow \mathbf{B}$ .

**Solution of Exercise 10.** Yes. Such a relation would be of the form  $R \subseteq \mathbf{A} \times \emptyset = \emptyset$ , where  $\mathbf{A}$  is an arbitrary set. In this situation,  $R = \emptyset$  is a relation  $\mathbf{A} \rightarrow \emptyset$ .





# PART C. ALGEBRA



---

6. Sets with operations	103
7. Morphisms	125
8. Actions	141

---







# 6. Sets with operations

This chapter introduces three of the most basic algebra structures: semigroups, monoids, and groups. These are sets with one *composition* operation. Additional properties of the operation distinguish among the structures.

6.1 Magmas	104
6.2 Semigroups	105
6.3 Code exercises - Semigroups	109
6.4 Code exercises - Free semigroup	111
6.5 Monoids	112
6.6 Dynamical systems	115
6.7 Code exercises - Monoids	116
6.8 Groups	117
6.9 Code exercises - Groups	122
6.10 Rings, fields	123
6.11 Vector spaces	124

“Unus pro omnibus, omnes pro uno” is a Latin phrase that means “One for all, all for one”. Interestingly, this is the unofficial motto of Switzerland. A French version, “Un pour tous, tous pour un”, was made famous by Alexandre Dumas in the 1844 novel “The Three Musketeers”.

## 6.1. Magmas

Oftentimes we are going to study certain *structures* and then their *refinements*. By *refinement*, we mean another type of structure that has additional properties or constraints.

The simplest algebraic structure that has to do with composition is that of a *magma*: it just assumes that there is a set with a binary operation defined on it.









### Definition 6.1 (Magma)

A *magma*  $\mathbf{S}$  is a set  $\mathbf{S}$ , together with a binary operation

$$\circ : \mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}, \quad (1)$$

which we refer to as “composition”.

**Table 6.1.:** Composition table.

$\circ$		
		
		

Given a finite set  $\mathbf{A}$ , one way to specify a composition operation  $\circ$  on  $\mathbf{A}$  is simply by writing out what it does with each pair of elements of  $\mathbf{A}$ . Since  $\circ$  is a function of two variables, this can be conveniently displayed as a table, sometimes called a *multiplication table* or a *Cayley table*. We will use the name *composition table*.

**Example 6.2.** Consider the set

$$\mathbf{A} = \{ \text{red flower}, \text{white flower} \}, \quad (2)$$

representing painting colors. A composition operation  $\circ$  is specified in Table 6.1. The rule describes the process of “painting over” another color, meaning that the last color that has been used to paint is the dominant one. We read it as saying

$$\begin{aligned} \text{red flower} \circ \text{red flower} &= \text{red flower} & \text{red flower} \circ \text{white flower} &= \text{white flower} \\ \text{white flower} \circ \text{red flower} &= \text{red flower} & \text{white flower} \circ \text{white flower} &= \text{white flower} \end{aligned}$$

### Definition 6.3

A magma  $\mathbf{S} = \langle \mathbf{S}, \circ \rangle$  is called *commutative* (or: Abelian) if

$$x \circ y = y \circ x, \quad (3)$$

for all  $x, y \in \mathbf{S}$ .

**Exercise 11.** Is the magma specified in Table 6.1 commutative?

*See solution on page 149.*

Magmas are quite general and simplistic. There is not all much to say about them.

We can build more interesting structures by considering, for example, properties that the composition operation might have.

In the next sections we will study:

- ▷ *semigroups*, magmas in which the operation is associative;
- ▷ *monoids*, semigroups with an identity;
- ▷ *groups*, monoids with an inverse operation.

## 6.2. Semigroups

A *semigroup* is a magma for which composition is associative.

### Definition 6.4 (Semigroup)

A *semigroup*  $\mathbf{S}$  is defined by:

#### Constituents

1. A set  $\mathbf{S}$ ;
2. A binary operation  $\circ : \mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$  called *composition*.

#### Conditions

1. Associative law

$$(x \circ y) \circ z = x \circ (y \circ z), \quad (4)$$

for all  $x, y, z \in \mathbf{S}$ .

**Remark 6.5.** Given a fixed set  $\mathbf{S}$ , there will in general be many choices of composition operation which make  $\mathbf{S}$  into a semigroup. Therefore, technically, a semigroup  $\mathbf{S}$  is a pair  $\langle \mathbf{S}, \circ \rangle$  consisting of a set  $\mathbf{S}$  and a choice of composition  $\circ$ . The set  $\mathbf{S}$  is the *underlying set* of the semigroup.

Often we will be slightly imprecise and refer to a semigroup simply by the name of its underlying set; this is practical when it is clear from context which composition operation we are considering, or when it is not necessary to refer to the composition operation explicitly.

Also note that any semigroup is in particular a magma  $\mathbf{S} = \langle \mathbf{S}, \circ \rangle$ , and so it also has an “underlying magma”.

### Definition 6.6 (Commutative semigroup)

A semigroup  $\mathbf{S} = \langle \mathbf{S}, \circ \rangle$  is called *commutative* (or: Abelian) if its underlying magma is commutative.

## Some examples

**Example 6.7.** Consider the semigroup  $\langle \mathbb{N}, + \rangle$ , which defines composition as

$$x \circ y := x + y. \quad (5)$$

This is a semigroup, since, for all  $l, m, n \in \mathbb{N}$ , we have

$$(l + m) + n = l + (m + n). \quad (6)$$

It is also a commutative semigroup because addition is commutative.

**Example 6.8.** Pair-wise average on  $\mathbb{R}$ ,

$$x \circ y := \frac{x + y}{2}, \quad (7)$$

does not define semigroup composition, because it is not associative.

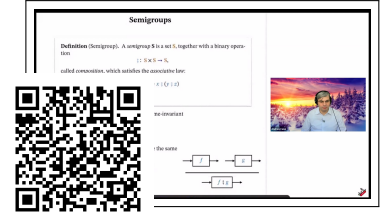
For example:

$$(4 \circ 8) \circ 16 = 11 \neq 8 = 4 \circ (8 \circ 16). \quad (8)$$

**Example 6.9 (Booleans).** Consider the set  $\mathbf{Bool} = \{\perp, \top\}$ , and  $\langle \mathbf{Bool}, \wedge \rangle$ , where the operation  $\wedge$  (“and”) is defined via Table 6.2.

This forms a semigroup, given the associativity of  $\wedge$ .

Watch: *Magnas and semigroups* (7 minutes).



**Table 6.2.:** Composition table for booleans.

$\wedge$	$\perp$	$\top$
$\perp$	$\perp$	$\perp$
$\top$	$\perp$	$\top$

**Graded exercise C.1** (CompositionTable)

Consider the composition presented in Table 6.1. Does this composition operation define a semigroup?

**Exercise 12.** [Cross-product] Consider  $\mathbf{S} = \mathbb{R}^3$  and the operation usually referred to as the “cross-product”:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \circ \begin{bmatrix} x \\ y \\ z \end{bmatrix} := \begin{bmatrix} bz - cy \\ cx - az \\ ay - bx \end{bmatrix}. \quad (9)$$

This is a binary operation and therefore  $\langle \mathbb{R}^3, \circ \rangle$  forms a magma. Show that this does not form a semigroup.

*See solution on page 149.*

**Example 6.10.** Consider a finite set  $\mathbf{A}$ , which we think of as an alphabet. For instance, consider

$$\mathbf{A} = \{\bullet, \circ\}. \quad (10)$$

Let  $\mathbf{S} = \text{List } \mathbf{A}$  be the set of non-empty lists of elements of  $\mathbf{A}$ . For example,

$$[\bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet, \circ] \quad (11)$$

is a non-empty list of elements of  $\mathbf{A}$ .

We may define a composition operation on  $\mathbf{S}$  simply by concatenating lists. Given the lists

$$[\bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet, \circ] \text{ and } [\circ, \bullet, \bullet, \circ], \quad (12)$$

their concatenation

$$[\bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet, \circ] \circ [\circ, \bullet, \bullet, \circ] \quad (13)$$

is the list

$$[\bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet, \circ, \circ, \bullet, \bullet, \circ]. \quad (14)$$

It is readily seen that concatenation satisfies the associative law, so  $\mathbf{S}$ , together with this multiplication, forms a semigroup. It is often called *free semigroup* on the set  $\mathbf{A}$ , a terminology which we will explain later.

**Graded exercise C.2** (VariationsOnConcatenation)

Consider the set  $\mathbf{S}$  of finite non-empty lists of symbols from the alphabet  $\mathbf{A}$ , as in Example 6.10.

Can you think of other candidates for multiplication operations on  $\mathbf{S}$ , besides the straightforward concatenation of lists considered above? Do your candidates define semigroup multiplications—that is, do they obey the associative law?

For example, one might consider the operation where, given an ordered pair of lists, one first doubles the last symbol of the first list and then concatenates. Is this operation associative? Justify your answers.

**Example 6.11.** The function  $\max : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  defines a multiplication operation which equips  $\mathbb{N}$  with the structure of a semigroup. It is easy to show that it satisfies associativity. Given  $x, y, z \in \mathbb{N}$ , we have:

$$\max(\max(x, y), z) = \max(x, \max(y, z)). \quad (15)$$



**Exercise 13.** Verify the statement made in Example 6.11; that is, check that the associative law holds.

Does  $\min : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  also define a semigroup structure on  $\mathbb{N}$ ?

See solution on page 149.

**Example 6.12.** Consider the set  $\mathbf{A} = \{\text{sprout}, \text{young}, \text{mature}, \text{old}, \text{dead}\}$  which describes five possible states of a plant. Let  $f : \mathbf{A} \rightarrow \mathbf{A}$  be the function that describes “development” (Fig. 1):

$$f(\text{sprout}) = \text{young}, \quad (16)$$

$$f(\text{young}) = \text{mature}, \quad (17)$$

$$f(\text{mature}) = \text{old}, \quad (18)$$

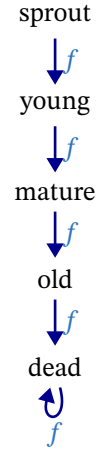
$$f(\text{old}) = \text{dead}, \quad (19)$$

$$f(\text{dead}) = \text{dead}. \quad (20)$$

In other words, we think of  $f$  as the change of state of the plant during a given time interval (say, three months). Composing the function  $f$  with itself corresponds to considering multiples of the given time interval. For example, the function

$$(f \circ f \circ f) : \mathbf{A} \rightarrow \mathbf{A} \quad (21)$$

models the change over the course of nine months. In general, for the  $n$ -fold composition of  $f$  with itself we write  $f^n$ . The set  $\mathbf{T} = \{f^n \mid n \in \mathbb{N}\}$ , together with the multiplication given by the composition operation, forms a semigroup.



**Figure 1.:** Graphical representation of plant transitions.

## Opposite semigroup

Any semigroup has an opposite: it is obtained by “flipping” the order of composition.

**Definition 6.13** (Opposite semigroup)

$\mathbf{S} = \langle \mathbf{S}, \circ \rangle$  be a semigroup. Its *opposite*  $\mathbf{S}^{\text{op}}$  is the semigroup  $\mathbf{S}^{\text{op}} = \langle \mathbf{S}, \circ^{\text{op}} \rangle$  whose composition operation  $\circ^{\text{op}}$  is the composite

$$\mathbf{S} \times \mathbf{S} \xrightarrow{\text{br}} \mathbf{S} \times \mathbf{S} \xrightarrow{\circ} \mathbf{S} \quad (22)$$

where  $\text{br}$  is the braiding function

$$\begin{aligned} \text{br} : \mathbf{S} \times \mathbf{S} &\rightarrow \mathbf{S} \times \mathbf{S}, \\ \langle x, y \rangle &\mapsto \langle y, x \rangle. \end{aligned} \quad (23)$$

In other words,

$$x \circ^{\text{op}} y = y \circ x \quad \forall x, y \in \mathbf{S}. \quad (24)$$

## Subsemigroups

**Definition 6.14** (Subsemigroup)

Let  $\mathbf{S} = \langle \mathbf{S}, \circ \rangle$  be a semigroup. A *subsemigroup* of  $\mathbf{S}$  is:

Constituents

1. A subset  $\mathbf{T} \subseteq \mathbf{S}$ .

Conditions



1. The set  $\mathbf{T}$  is closed under the composition operation  $\circ$  from  $\mathbf{S}$ :

$$\frac{x \in \mathbf{T} \quad y \in \mathbf{T}}{(x \circ y) \in \mathbf{T}}. \quad (25)$$

**Example 6.15.** Consider the semigroup  $\langle \mathbb{N}, + \rangle$  introduced in Example 6.7. We can take the subset of even natural numbers  $\mathbf{T} \subset \mathbb{N}$ . One can show that the sum of two even numbers is always even, satisfying the closure of  $\mathbf{T}$  under the composition operation  $+$ .

### Induced $n$ -ary multiplication

Given a semigroup  $\langle \mathbf{A}, \circ \rangle$ , for each integer  $n \geq 1$ , we can define an induced  $n$ -ary multiplication operation

$$\begin{aligned} \circ^n : \mathbf{A}^n &\rightarrow \mathbf{A}, \\ \langle x_1, x_2, \dots, x_n \rangle &\mapsto x_1 \circ x_2 \dots \circ x_n. \end{aligned} \quad (26)$$

Thanks to the associative law, this is well-defined — that is, we do not need to use parentheses. We will say that an element  $x \in \mathbf{A}$  is an  $n$ -fold multiplication if it is in the image of this  $n$ -ary multiplication operation. At times, we may not wish to specify the arity of the multiplication, in which case we just speak of a *multiplication*.

## 6.3. Semigroups

**Remark 6.16.** In this course we assume, unless otherwise noted, that you have done the previous exercises, so that you can build on top of the code that you already have. The first exercise and tutorial was in Section 3.3.

We define two classes: `Semigroup` and `FiniteSemigroup`. Figure 2 shows their relation.

A `Semigroup` is something that *has* a carrier set, which is a `Setoid`. It also has a `compose` function.

**Listing 37:** The `Semigroup` interface.

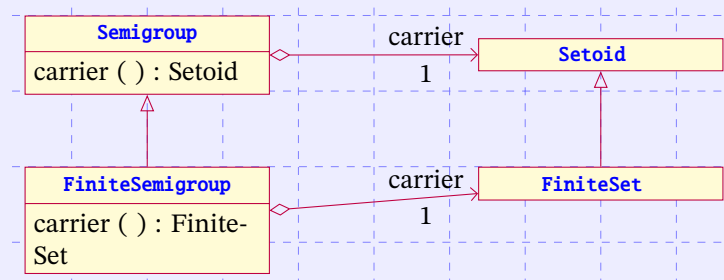
```
class Semigroup(Generic[E], ABC):
    @abstractmethod
    def carrier(self) -> Setoid[E]: ...

    @abstractmethod
    def compose(self, a: E, b: E) -> E: ...
```

A `FiniteSemigroup` is a special `Semigroup` whose carrier is a `FiniteSet`.

**Listing 38:** The `FiniteSemigroup` interface.

```
class FiniteSemigroup(Generic[E], Semigroup[E], ABC):
    @abstractmethod
    def carrier(self) -> FiniteSet[E]: ...
```



**Figure 2.**

```

carrier:
  elements: [0, 1]
composition:
- [[0, 0], 0]
- [[0, 1], 0]
- [[1, 0], 0]
- [[1, 1], 1]

```

Figure 3.

## Semigroup representation

Figure 3 shows the data format to define a finite semigroup.

The data structure contains two fields:

1. The field `carrier` is the serialization of a `FiniteSet A`.
2. The field `composition` is the serialization of a `FiniteMap`. The source for this map must be a product of sets ( $A \times A$ ), and the target must be equal to `A`.

### Code exercise C.1 (TestFiniteSemigroupRepresentation)

Create a function to load the data, with the interface shown in Listing 39.

Implement the interface in Listing 39.

Test using:

```
act4e-check TestFiniteSemigroupRepresentation
```

**Listing 39:** The `FiniteSemigroupRepresentation` interface.

```

class FiniteSemigroupRepresentation(Serializer[FiniteSemigroup[Any],
  ↪ FiniteSemigroup_desc], ABC):
  pass

```

*Hint:* You want to reuse the code you already have to load a `FiniteSet` and a `FiniteMap`.

## 6.4. ○ Free semigroup

### Code exercise C.2 (TestFiniteSemigroupConstruct)

Given a finite set, construct the free semigroup.

Implement the interface in Listing 40.

Test using:

```
act4e-check TestFiniteSemigroupConstruct
```

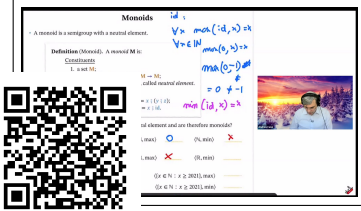
**Listing 40:** The `FiniteSemigroupConstruct` interface.

```
class FiniteSemigroupConstruct(ABC):
    @abstractmethod
    def free(self, fs: FiniteSet[C]) -> FreeSemigroup[C, Any]:
        """Construct the free semigroup on a set."""
```

A `FreeSemigroup` is a semigroup with the additional method `unit()` that converts an element of the carrier set to an element of the free group (e.g., from element to a list with one element.)

**Listing 41:** The `FreeSemigroup` interface.

```
class FreeSemigroup(Generic[C, E], Semigroup[E], ABC):
    @abstractmethod
    def unit(self, a: C) -> E:
        """From an element of the carrier, returns the element of the free
        ↪ semigroup"""
```

Watch: *Monoids* (13 minutes).

## 6.5. Monoids

Algebraic structures are often defined in *layers*.

For example, in the definition of semigroup, we start with a set **S** as a basic building block, and we add a layer of structure to it, namely a multiplication operation  $\circ : \mathbf{S} \times \mathbf{S} \rightarrow \mathbf{S}$ . The multiplication operation for semigroups was not only a new *structure* which we added, but we also required this structure to obey a *condition*, namely that it satisfies the associative law. One might also say that the multiplication operation was a new *constituent* or a new *datum*, and that satisfying the associative law is a *property*.

Mathematicians often use such words in an intuitive, non-rigorous way as a tool for structuring their thinking. We will do the same. For clarity, we will aim to stick with the words *constituents* and *conditions*. Roughly speaking, we think of constituents as building blocks, and we think of conditions as rules for how those blocks fit together and behave.

Using the constituent vs condition distinction we will, in particular, present some definitions in the following succinct, list-like fashion:

### Definition 6.17 (Monoid)

A *monoid* **M** is given by:

#### Constituents

1. A set **M**;
2. A binary operation  $\circ : \mathbf{M} \times \mathbf{M} \rightarrow \mathbf{M}$ ;
3. A specified element  $\text{id} \in \mathbf{M}$ , called *neutral element*.

#### Conditions

1. Associative law:  $(x \circ y) \circ z = x \circ (y \circ z) \quad \forall x, y, z \in \mathbf{M}$ ;
2. Neutrality Laws:  $\text{id} \circ x = x = x \circ \text{id} \quad \forall x \in \mathbf{M}$ .

**Remark 6.18.** The way that we presented the definition of a monoid is certainly not unique. For example, we could have done the following.

A *monoid* **M** is:

#### Constituents

1. a semigroup  $\langle \mathbf{M}, \circ \rangle$ ;
2. a specified element  $\text{id} \in \mathbf{M}$ , called *neutral element*.

#### Conditions

1. Neutrality laws:  $\text{id} \circ x = x = x \circ \text{id}$ .

In this version, two constituents and one condition from Def. 6.17 are “compressed” into the information that we are using here a semigroup as a constituent. This kind of “compression” has its pros and cons; depending on the context will use it to varying degrees.

There is a similar dilemma when considering the software interfaces to describe these structures. In terms of software engineering, the two strategies are *composition* (a monoid has a semigroup as a constituent) and *inheritance* (a monoid is a semigroup with additional data).

**Remark 6.19.** A monoid is called *commutative* (or: Abelian) if its underlying semigroup is commutative.

## Some examples

**Example 6.20.** Consider  $\langle \mathbb{R}, +, 0 \rangle$ . This is a monoid, since, for all  $x, y, z \in \mathbb{R}$ , we have

$$(x + y) + z = x + (y + z), \quad (27)$$

and

$$x + 0 = x = 0 + x. \quad (28)$$

Similarly,  $\langle \mathbb{N}, +, 0 \rangle$ ,  $\langle \mathbb{Z}, +, 0 \rangle$ , and  $\langle \mathbb{Q}, +, 0 \rangle$  are monoids.

**Example 6.21.** The set  $\mathbb{Z}$ , together with the operation of multiplication of whole numbers, forms a monoid. The neutral element is the number 1.

**Example 6.22.** Given a set  $\mathbf{A}$ , the set  $\mathbf{End}(\mathbf{A})$  of functions from  $\mathbf{A}$  to  $\mathbf{A}$  comes “naturally equipped” with a monoid structure: take monoid composition to be function composition, and let the identity element be given by the identity function on  $\mathbf{A}$ .

**Example 6.23.** Consider  $\langle \mathbf{Bool}, \wedge \rangle$  as in Example 6.9, and consider  $\top$  as neutral element. This forms a monoid, since  $b \wedge \top = b = \top \wedge b$ , for all  $b \in \mathbf{B}$ .

**Lemma 6.24.** Let  $\langle \mathbf{S}, \circ \rangle$  be a semigroup. If there exist elements  $1 \in \mathbf{S}$  and  $1' \in \mathbf{S}$  such that  $\langle \mathbf{S}, \circ, 1 \rangle$  and  $\langle \mathbf{S}, \circ, 1' \rangle$  are each monoids, then  $1 = 1'$  must hold. In other words, the neutral element of a monoid is uniquely determined by the underlying semigroup structure.

### Graded exercise C.3 (UniqueNeutralMonoid)

Prove Lemma 6.24.

**Example 6.25.** Consider  $\langle \mathbb{R}_{\geq 0}, \max, 0 \rangle$ . This is a monoid, since, for all  $x, y \in \mathbb{R}_{\geq 0}$ , we have:

$$\max(\max(x, y), z) = \max(x, \max(y, z)), \quad (29)$$

and

$$\max(x, 0) = x = \max(0, x). \quad (30)$$

**Remark 6.26.** Note that in the above example, we could have just as well instead considered the set  $\mathbb{R}_{\geq 7.5}$  of real numbers greater than 7.5, together with “max” as composition and 7.5 as neutral element. In other words, we can choose any real number  $a \in \mathbb{R}$  and obtain a monoid  $\langle \mathbb{R}_{\geq a}, \max, a \rangle$ .

**Example 6.27.**  $\langle \mathbb{N}, \max, 0 \rangle$  forms a monoid.

### Definition 6.28

Let  $\mathbf{A}$  be a set. We denote by  $\mathbf{List} \mathbf{A}$  the set of all lists of elements of  $\mathbf{A}$ .

**Example 6.29.** For any set  $\mathbf{A}$ , the set  $\mathbf{List} \mathbf{A}$  of lists of elements of  $\mathbf{A}$  can naturally be equipped with a monoid structure: composition is concatenation (just like in Example 6.10), and the neutral element is the empty list  $[]_{\mathbf{A}}$ . This monoid is known as the *free monoid* on the set  $\mathbf{A}$ .

**Remark 6.30.** Just like for semigroups, any monoid has an opposite. It is defined similarly: given a monoid  $\mathbf{M} = \langle \mathbf{M}, \circ_{\mathbf{M}}, \text{id}_{\mathbf{M}} \rangle$ , its opposite is  $\mathbf{M}^{\text{op}} = \langle \mathbf{M}, \circ^{\text{op}}, \text{id} \rangle$  where  $\circ^{\text{op}}$  is defined by setting

$$x \circ^{\text{op}} y = y \circ x \quad \forall x, y \in \mathbf{M}. \quad (31)$$

## Submonoids

### Definition 6.31 (Submonoids)

Let  $\mathbf{M} = \langle \mathbf{M}, \circ_{\mathbf{M}}, \text{id}_{\mathbf{M}} \rangle$  be a monoid. A *submonoid* of  $\mathbf{M}$  is:

#### Constituents

1. A subset  $\mathbf{N} \subseteq \mathbf{M}$ .

#### Conditions

1. The set  $\mathbf{N}$  is closed under the composition operation  $\circ$  from  $\mathbf{M}$ :

$$\frac{x \in \mathbf{N} \quad y \in \mathbf{N}}{x \circ y \in \mathbf{N}}; \quad (32)$$

2. The neutral element  $\text{id} \in \mathbf{M}$  is an element of  $\mathbf{N}$ .

**Example 6.32.**  $\langle \mathbb{N}, +, 0 \rangle$ ,  $\langle \mathbb{Z}, +, 0 \rangle$ , and  $\langle \mathbb{Q}, +, 0 \rangle$  are all submonoids of  $\langle \mathbb{R}, +, 0 \rangle$ .

**Example 6.33.**  $\mathbb{N}$  is a submonoid of  $\langle \mathbb{Z}, \cdot, 1 \rangle$ .

**Example 6.34.**  $\{x \in \mathbb{N} \mid x \text{ is even}\}$  is a submonoid of  $\langle \mathbb{N}, +, 0 \rangle$ , provided that 0 is considered an even natural number.

## 6.6. Dynamical systems and transition functions

In this section we discuss an example of how monoids are related to the evolution of a dynamical system in time.

**Definition 6.35** (Continuous-time dynamical system)

A dynamical system on  $\mathbb{R}^n$  may be defined by a function

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^n. \quad (33)$$

A trajectory of a dynamical system is a function  $x : \mathbb{R} \rightarrow \mathbb{R}^n$  such that

$$\dot{x}(t) = f(x(t)). \quad (34)$$

We use the notation  $\dot{x}$  to abbreviate  $dx/dt$ .

Suppose that we are dealing with dynamical systems such that for any point  $x_0$ , there is exactly one trajectory beginning at  $x_0$ . (For this we need to put reasonable constraints on the function  $f$ .)

We can then ask the following: given a point  $x$ , where would its trajectory be after  $\delta$ ? This question induces a family of functions  $T_\delta$ , called transition functions. For each particular  $\delta$ , we have a function

$$T_\delta : \mathbb{R}^n \rightarrow \mathbb{R}^n \quad (35)$$

that maps a point to its position  $\delta$  in the future.

We can spot here a semigroup structure. Suppose we want to know the position of a point  $\delta_1 + \delta_2$  in the future. We can take  $T_{\delta_1}$  and compose it with  $T_{\delta_2}$ ; or take directly  $T_{\delta_1 + \delta_2}$ . By construction, we will have that

$$T_{\delta_1 + \delta_2} = T_{\delta_1} \circ T_{\delta_2}. \quad (36)$$

We can also easily prove associativity:

$$\begin{aligned} T_{\delta_1} \circ (T_{\delta_2} \circ T_{\delta_3}) &= T_{\delta_1} \circ T_{\delta_2 + \delta_3} \\ &= T_{\delta_1 + \delta_2 + \delta_3} \\ &= T_{\delta_1 + \delta_2} \circ T_{\delta_3} \\ &= (T_{\delta_1} \circ T_{\delta_2}) \circ T_{\delta_3}. \end{aligned} \quad (37)$$

This shows that the set of transition functions for a particular system with the operation of function composition form a semigroup.

This semigroup is a monoid because there is an identity. The identity is  $T_0$ , the map that tells us what happens after 0 seconds. That is  $\text{id}_{\mathbb{R}^n}$ , the identity on  $\mathbb{R}^n$ . To show that  $T_0 = \text{id}_{\mathbb{R}^n}$  is an identity, we can fix any  $\delta$  and substituting in (36) we have

$$\begin{aligned} T_{\delta + 0} &= T_\delta \circ T_0 \\ &= T_\delta. \end{aligned} \quad (38)$$



## 6.7. Monoids

A **Monoid** is a **Semigroup** with an identity.

**Listing 42:** The **Monoid** interface.

```
class Monoid(Generic[E], Semigroup[E], ABC):
    @abstractmethod
    def identity(self) -> E: ...
```

A **FiniteMonoid** is a particular **FiniteSemigroup**.

**Listing 43:** The **FiniteMonoid** interface.

```
class FiniteMonoid(Generic[E], Monoid[E], FiniteSemigroup[E], ABC):
    ...
```

### Representation

The file format for monoids is an extension of those for semigroups. There is an extra field **neutral** that gives you the neutral element. Figure 4 shows an example.

```
carrier:
  elements: [0, 1]
composition:
- [[0, 0], 0]
- [[0, 1], 0]
- [[1, 0], 0]
- [[1, 1], 1]
neutral: 1
```

**Figure 4.:** The simple monoid  $\{0, +1\}$  with multiplication

#### Code exercise C.3 (TestFiniteMonoidRepresentation)

Create a function to load and save monoids.

Implement the interface in Listing 44.

Test using:

```
act4e-check TestFiniteMonoidRepresentation
```

**Listing 44:** The **FiniteMonoidRepresentation** interface.

```
class FiniteMonoidRepresentation(Serializer[FiniteMonoid[Any],
    ↵ FiniteMonoid_desc], ABC):
    ...
```

## 6.8. Groups

Groups appear in many areas of mathematics, both pure and applied.

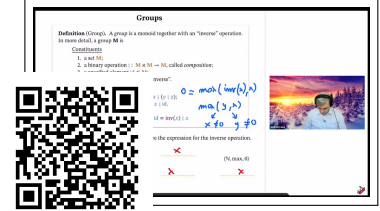
One important use of groups is to describe symmetries. Roughly speaking, a symmetry is an invertible transformation or reconfiguration of a figure (or object) that leaves the essential features of that figure invariant.

For example, consider a perfectly square sheet of monochrome paper lying on a table-top. If we rotate the piece of paper by 90 degrees around its center, how it appears to us after this rotation will be essentially indistinguishable from before it was rotated. Thus, this rotation is a symmetry.

Groups of symmetries play an important role in physics and chemistry, for example when studying the repeating patterns of lattices of atoms or molecules in materials, or when studying the geometric patterns of atoms and molecules themselves.

For engineering, a particularly important class of groups are matrix groups, in particular those that describe various types of motion in space. These are discussed in Section 6.8.

Watch: *Groups* (10 minutes).



### Definition 6.36 (Group)

A *group* is a monoid together with an “inverse” operation. In more detail, a group  $G$  is

#### Constituents

1. a set  $G$ ;
2. a binary operation  $\circ : G \times G \rightarrow G$ , called *composition*;
3. a specified element  $id \in G$ ;
4. a map  $inv : G \rightarrow G$ , called *inverse*.

#### Conditions

1. Associative law:  $(x \circ y) \circ z = x \circ (y \circ z)$ ,  $\forall x, y, z \in G$ ;
2. Neutrality laws:  $id \circ x = x = x \circ id$ ,  $\forall x \in G$ ;
3. Inverse laws:

$$inv(x) \circ x = id = x \circ inv(x), \quad \forall x \in G. \quad (39)$$

### Definition 6.37 (Commutative group)

A group is called *commutative* (or: *Abelian*) if its underlying semigroup is commutative.

**Remark 6.38.** The size of the underlying set of a group is often called the *order* or *cardinality* of the group. We’ll sometimes use this terminology, or just call it the size of the group.

## Some examples

**Example 6.39.** The following is a group: the set  $\mathbb{Z}$ , together with addition as the composition operation, the element 0 as neutral element, and “taking the negative” as the inverse operation:

$$inv(x) := -x, \quad \forall x \in \mathbb{Z}. \quad (40)$$

**Example 6.40.** The monoid  $(\mathbb{R}_{\setminus \{0\}}, \cdot, 1)$  becomes a group when equipped with

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

the inverse operation defined by

$$\text{inv}(x) := \frac{1}{x}, \quad \forall x \in \mathbb{R}. \quad (41)$$

**Example 6.41.** For the monoids  $\langle \mathbb{N}, +, 0 \rangle$  and  $\langle \mathbb{N}, \cdot, 1 \rangle$  we cannot find an inverse operation that would turn these monoids into groups.

**Exercise 14.** Can one find an inverse operation for the monoid  $\langle \mathbb{N}, \max, 0 \rangle$ ?

*See solution on page 149.*

**Example 6.42.** The monoid  $\langle \mathbf{Bool}, \wedge, \top \rangle$  from Example 6.23 cannot become a group, because there cannot be an inverse for  $\perp$ : there is no possible choice for  $\text{inv}(\perp)$  such that  $\text{inv}(\perp) \wedge \perp = \top$ .

**Example 6.43.** Given a set  $\mathbf{A}$ , an invertible function  $\mathbf{A} \rightarrow \mathbf{A}$  is called an *automorphism* of  $\mathbf{A}$ . There is a “naturally given” group structure on the set  $\mathbf{Aut}(\mathbf{A})$  of automorphisms of  $\mathbf{A}$ : we can take the composition operation to be the composition of functions, the neutral element is the identity function on  $\mathbf{A}$ , and inverses are given by the inverses of functions.

As a sub-example, consider the set

$$\mathbf{A} = \{1, 2, 3, \dots, n-1, n\}, \quad (42)$$

where  $n \in \mathbb{N}$ . Then  $\mathbf{Aut}(\mathbf{A})$  is the group of permutations of  $n$  elements. The usual notation for this group is  $\mathbf{Perms}_n$ . Its size is  $n! = n \cdot (n-1) \cdot \dots \cdot 3 \cdot 2 \cdot 1$ .

**Example 6.44.** Consider the set  $\{0, 1\}$ , equipped with the composition operation  $\%$  defined to be “addition modulo 2”. The composition table for this is Table 6.3

**Table 6.3.:** Addition modulo 2 on the set  $\{0, 1\}$ .

+	0	1
0	0	1
1	1	0

Choose 0 as the neutral element, and let  $\text{inv}(0) = 0$  and  $\text{inv}(1) = 1$ . Then  $\langle \{0, 1\}, \%, \text{id}, \text{inv} \rangle$  is a group.

**Example 6.45.** Consider the following set of complex numbers:

$$\{1, e^{\frac{1}{3}2\pi i}, e^{\frac{2}{3}2\pi i}\} \subseteq \mathbb{C}. \quad (43)$$

Taking the usual multiplication of complex numbers as the composition operation, these three numbers form a group.

**Table 6.4.:** Cyclic group of order 4.

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

**Example 6.46.** The set  $\{0, 1, 2, 3\}$  may be equipped with a group structure whose composition operation is addition modulo 4, and where 0 is the neutral element. The composition table is Table 6.4.

**Example 6.47.** Consider the shape of a rectangle in the plane, oriented vertically, and assume the rectangle is not a square. Then there are four symmetries of this shape:

1. doing nothing (leaving everything in place);
2. reflecting the shape along the vertical axis;
3. reflecting the shape along the horizontal axis;
4. rotating the shape by 180 degrees.

Let us call these symmetries  $I$ ,  $V$ ,  $H$ , and  $R$ , respectively. We can model these for instance using bijective functions of the plane  $\mathbb{R}^2$ . The set  $\{I, V, H, R\}$  can be given the structure of a group, with  $I$  being its neutral element, and with composition and inverses given as in Table 6.5. We think of composition meaning

**Table 6.5.:** The Klein four group

+	$I$	$V$	$H$	$R$
$I$	$I$	$V$	$H$	$R$
$V$	$V$	$I$	$R$	$H$
$H$	$H$	$R$	$I$	$V$
$R$	$R$	$H$	$V$	$I$

that we perform one symmetry transformation and then the other (function composition), and inverses correspond to taking the inverse transformation. This group is known as the Klein four-group, named after the mathematician Felix Klein.

**Example 6.48.** Similar to the previous example, all sorts of geometric shapes have groups of symmetries associated with them.

For example, each of the five Platonic solids (Fig. 5) have a group of symmetries associated with them.

It turns out that the cube and the octahedron have the same group of symmetries, and similarly the dodecahedron and the icosahedron have the same symmetry group. This has to do with the fact that these Platonic solids are dual to each other, respectively, in the following sense. An octahedron can be obtained from a cube by replacing faces with vertices and vertices with faces, and the same goes for the dodecahedron and the icosahedron.

The symmetry group of the tetrahedron has order 24, the symmetry group of the cube and octahedron has order 48, and the symmetry group of the dodecahedron and icosahedron has order 120.

**Example 6.49.** There is a group that describes all the possible manipulations of a Rubik's cube (Fig. 6). The size of this group is

$$43,252,003,274,489,856,000 = 12! \cdot 2^{10} \cdot 8! \cdot 3^7. \quad (44)$$

## Matrix groups

There are various *matrix groups* (Table 8.1) that represent linear transformations having special properties. Here we consider only matrices with entries from the real number field  $\mathbb{R}$ .

**Definition 6.50** (General linear group  $GL(n, \mathbb{R})$ )

The (real) general linear group of order  $n$ , written  $GL(n, \mathbb{R})$ , is the group of  $n \times n$  invertible matrices with entries in  $\mathbb{R}$ .

**Definition 6.51** (Orthogonal group  $O(n, \mathbb{R})$ )

The (real) orthogonal group of order  $n$ , written  $O(n, \mathbb{R})$ , is the group of  $n \times n$  real matrices that satisfy

$$\mathbf{M}\mathbf{M}^T = \mathbf{M}^T\mathbf{M} = \mathbf{1}. \quad (45)$$

**Definition 6.52**

Special linear group  $SL(n, \mathbb{R})$ ] The (real) special linear group of order  $n$ , written  $SL(n, \mathbb{R})$ , is the group of  $n \times n$  invertible real matrices with determinant equal to 1.

**Definition 6.53**

Special orthogonal group  $SO(n, \mathbb{R})$ ] The (real) special orthogonal group of order  $n$ , written  $SO(n, \mathbb{R})$ , is the group of  $n \times n$  real matrices that satisfy

$$\mathbf{M}\mathbf{M}^T = \mathbf{M}^T\mathbf{M} = \mathbf{1}, \quad (46)$$

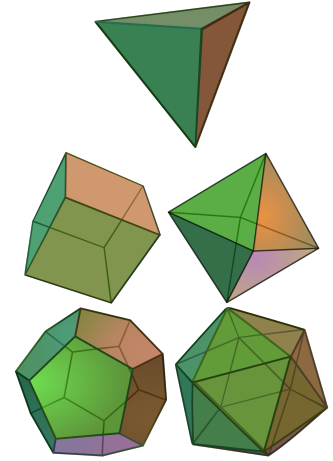


Figure 5.: The Platonic solids

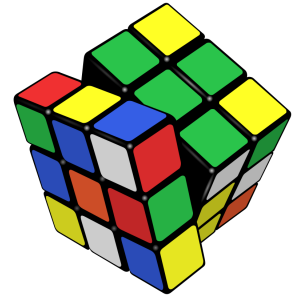


Figure 6.: A Rubik's cube

and  $\det(\mathbf{M}) = 1$ .

## Properties of groups

**Lemma 6.54.** Let  $\langle \mathbf{G}, \circ, \text{id}, \text{inv} \rangle$  be a group. Then

1.  $\text{inv}(\text{id}) = \text{id}$ ;
2.  $\text{inv}(\text{inv}(x)) = x, \quad \forall x \in \mathbf{G}$ ;
3.  $\text{inv}(x \circ y) = \text{inv}(y) \circ \text{inv}(x), \quad \forall x, y \in \mathbf{G}$ .

**Lemma 6.55.** Let  $\mathbf{G} = \langle \mathbf{G}, \circ, \text{id}, \text{inv} \rangle$  be a group and let  $x, y \in \mathbf{G}$ . If  $x$  and  $y$  satisfy the equation

$$x \circ y = \text{id}, \quad (47)$$

then  $y = \text{inv}(x)$  and  $x = \text{inv}(y)$ .

*Proof.* If  $x \circ y = \text{id}$ , then, by composing both sides of this equation with  $\text{inv}(x)$  from the left, and using associativity to remove brackets, we find  $\text{inv}(x) \circ x \circ y = \text{id} \circ \text{inv}(x)$ . Applying the inverse laws on the left-hand side, we obtain  $\text{id} \circ y = \text{id} \circ \text{inv}(x)$ , and using the neutrality laws on both side of this equation yields  $y = \text{inv}(x)$ . The fact that  $x = \text{inv}(y)$  may be proved similarly.  $\square$

**Corollary 6.56.** Let  $\mathbf{M} = \langle \mathbf{M}, \circ_{\mathbf{M}}, \text{id}_{\mathbf{M}} \rangle$  be a monoid. If  $\text{inv}_1$  and  $\text{inv}_2$  are both operations of inverse which make  $\mathbf{M}$  into a group, then  $\text{inv}_1 = \text{inv}_2$  holds. In other words, if a monoid can be made into a group by adding an inverse operation, then the resulting group is uniquely determined by the underlying monoid.

*Proof.* Let  $x \in \mathbf{M}$ . Since  $\langle \mathbf{M}, \circ, \text{id}, \text{inv}_2 \rangle$  is a group, we have  $x \circ \text{inv}_2(x) = \text{id}$ . Also  $\langle \mathbf{M}, \circ, \text{id}, \text{inv}_1 \rangle$  is a group, and in terms of this group, the equation  $x \circ \text{inv}_2(x) = \text{id}$  implies, by Lemma 6.55, that  $\text{inv}_2(x) = \text{inv}_1(x)$ , by thinking of  $\text{inv}_2(x)$  in the role of  $y$  from Lemma 6.55. Since  $x \in \mathbf{M}$  was arbitrary,  $\text{inv}_2 = \text{inv}_1$  as functions on  $\mathbf{M}$ .  $\square$

**Example 6.57** (Orthogonal matrices). Fix an integer  $n \geq 1$  and consider the set of real *orthogonal* matrices  $\mathbf{A} \in \mathbb{R}^{n \times n}$ : real, square matrices with orthonormal columns and rows. One way to express orthogonality of a matrix is:

$$\mathbf{A}^T \mathbf{A} = \mathbf{A} \mathbf{A}^T = \mathbf{1}, \quad (48)$$

where  $\mathbf{1}$  is the *identity matrix*. The set  $\mathbb{R}^{n \times n}$  equipped with matrix multiplication as a binary operation, the identity matrix as the neutral element, and the transposition  $(\cdot)^T$  (which for this specific type of matrices corresponds to the inverse) forms a group, usually denoted  $O(n)$ . Any orthogonal matrix  $\mathbf{A}$  has the property  $\det(\mathbf{A}) \in \{-1, +1\}$ . The subset of orthogonal  $n \times n$  matrices with determinant 1 forms the so-called *special orthogonal group*  $SO(n, \mathbb{R})$ .

### Graded exercise C.4 (GroupWithThreeElements)

In Example 6.45, what is the neutral element? What is the inverse operation? Draw the composition table for this group.

### Graded exercise C.5 (GroupInverseProperties)

Prove Lemma 6.54.

**Remark 6.58.** Just like for semigroups and monoids, any group has an opposite. It is defined analogously.

## Subgroups

### Definition 6.59 (Subgroup)

Let  $G = \langle G, \circ, id, inv \rangle$  be a group. A *subgroup* of  $G$  is:

#### Constituents

1. A subset  $H \subseteq G$ .

#### Conditions

1. The set  $H$  is closed under the composition operation  $\circ$  from  $G$ :

$$\frac{x \in H \quad y \in H}{x \circ y \in H}; \quad (49)$$

2. The neutral element  $id \in G$  is an element of  $H$ ;
3. The set  $H$  is closed under the inverse operation  $inv$  from  $G$ :

$$\frac{x \in H}{inv(x) \in H}. \quad (50)$$

## 6.9. Groups

A **Group** is a **Monoid** that has an inverse **Mapping**.

**Listing 45:** The **Group** interface.

```
class Group(Generic[E], Monoid[E], ABC):
    @abstractmethod
    def inverse(self, e: E) -> E:
        """Returns the inverse of an element"""
```

A **FiniteGroup** is a particular case of **Group** and **FiniteMonoid** where the inverse is a **FiniteMap**.

**Listing 46:** The **FiniteGroup** interface.

```
class FiniteGroup(Generic[E], Group[E], FiniteMonoid[E], ABC):
    ...
```

### Representation

The file format for groups is an extension of the one for monoids (which was an extension of the one for semigroups). There is an extra field **inverse** which gives you the inverse map (Fig. 7).

```
carrier:
  elements: [-1, 1]
composition:
- [[-1, -1], 1]
- [[-1, 1], -1]
- [[1, -1], -1]
- [[1, 1], 1]

neutral: 1
inverse:
- [-1, -1]
- [1, 1]
```

**Figure 7.:** Simple group  $\{-1, +1\}$  with multiplication

#### Code exercise C.4 (TestFiniteGroupRepresentation)

Create a function to load groups.

Implement the interface in Listing 47.

Test using:

```
act4e-check TestFiniteGroupRepresentation
```

**Listing 47:** The **FiniteGroupRepresentation** interface.

```
class FiniteGroupRepresentation(Serializer[FiniteGroup[Any],
    ↪ FiniteGroup_desc], ABC):
    ...
```

## 6.10. Rings, fields

### Definition 6.60 (Ring)

A *ring* is

#### Constituents

1. a set  $\mathbf{R}$ ;
2. a binary operation  $\cdot : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ , called *multiplication*;
3. a binary operation  $+ : \mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$ , called *addition*;
4. a specified element  $1 \in \mathbf{R}$  called *one* (or *unit*);
5. a specified element  $0 \in \mathbf{R}$  called *zero*.

#### Conditions

1.  $\langle \mathbf{R}, \cdot \rangle$  is a monoid, with neutral element 1;
2.  $\langle \mathbf{R}, + \rangle$  is a commutative group, with neutral element 0;
3. Distributive law:

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z), \quad \forall x, y, z \in \mathbf{R}. \quad (51)$$

### Definition 6.61 (Field)

A field is a ring  $\langle \mathbf{K}, \cdot, +, 1, 0 \rangle$  such that  $\langle \mathbf{K} \setminus \{0\}, \cdot \rangle$  is a commutative group.



## 6.11. Vector spaces

**Definition 6.62** (Real vector space)

A real vector space is

Constituents

1. a set  $\mathbf{V}$ , the elements of which are called *vectors*;
2. a binary operation  $+$  :  $\mathbf{V} \times \mathbf{V} \rightarrow \mathbf{V}$ , called *vector addition*;
3. an element  $0 \in \mathbf{V}$ ;
4. an operation  $\cdot$  :  $\mathbb{R} \times \mathbf{V} \rightarrow \mathbf{V}$ , called *scalar multiplication*.

Conditions

1.  $\langle \mathbf{V}, + \rangle$  is a commutative group, with neutral element 0;
2. Scalar multiplication is an action:
  - a)  $(\lambda\mu) \cdot x = \lambda \cdot (\mu \cdot x) \quad \forall \lambda, \mu \in \mathbb{R}, \forall x \in \mathbf{V}$ ;
  - b)  $1 \cdot x = x \quad \forall x \in \mathbf{V}$ ;
  - c)  $(\lambda + \mu) \cdot x = (\lambda \cdot x) + (\mu \cdot x) \quad \forall \lambda, \mu \in \mathbb{R}, \forall x \in \mathbf{V}$ ;
  - d)  $\lambda \cdot (x + y) = (\lambda \cdot x) + (\lambda \cdot y), \quad \forall \lambda \in \mathbb{R}, \forall x, y \in \mathbf{V}$ .

**Remark 6.63.** The general definition of a vector space is obtained by replacing  $\mathbb{R}$  in the above definition with an arbitrary field  $\mathbf{K}$ .

**Example 6.64.**  $\langle \mathbb{R}^n, +, \cdot \rangle$

**Graded exercise C.6** (RealPolynomials)

Let  $\mathbf{A}$  denote the set of polynomials in one variable and with coefficients in  $\mathbb{R}$ . In other words, elements of  $\mathbf{A}$  are polynomials of the form

$$p(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0, \quad (52)$$

where  $n \in \mathbb{N}$  and  $a_n, a_{n-1}, \dots, a_1, a_0 \in \mathbb{R}$  may vary.

1. Check (prove) that  $\mathbf{A}$  forms a commutative monoid when equipped with the usual addition operation for polynomials and the zero polynomial  $p(X) = 0$  as neutral element.
2. Is the commutative monoid  $\langle \mathbf{A}, +, 0 \rangle$  in fact a group?
3. Check that  $\langle \mathbf{A}, +, 0 \rangle$  forms a real vector space when we equip it with the following (usual) scalar multiplication:

$$\lambda \cdot (a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0) := \lambda a_n X^n + \lambda a_{n-1} X^{n-1} + \cdots + \lambda a_1 X + \lambda a_0 \quad (53)$$

for any  $\lambda \in \mathbb{R}$ .



## 7. Morphisms

In this chapter we look at morphisms, which are maps between two semigroups (or monoids, groups) that “preserve the structure”.

<b>7.1 Semigroup morphisms . . . . .</b>	<b>126</b>
<b>7.2 Encoding as morphism . . . . .</b>	<b>128</b>
<b>7.3 Morse coding . . . . .</b>	<b>129</b>
<b>7.4 Monoid morphisms . . . . .</b>	<b>131</b>
<b>7.5 Group morphisms . . . . .</b>	<b>133</b>
<b>7.6 Generators and relations . . . . .</b>	<b>135</b>
<b>7.7 Code exercises - Morphisms . . .</b>	<b>138</b>

A car shuttle train, is a shuttle train used to transport cars, and usually also other types of road vehicles, for a relatively short distance. Switzerland is well known for many such shuttle trains, usually operating through tunnels.

## 7.1. Semigroup morphisms

A *morphism* is a map between semigroups that “preserves the structure” of composition.

### Definition 7.1 (Semigroup morphism)

A morphism  $F : \mathbf{S} \rightarrow \mathbf{T}$  between semigroups

$$\mathbf{S} = \langle \mathbf{S}, \circ_{\mathbf{S}} \rangle \quad \text{and} \quad \mathbf{T} = \langle \mathbf{T}, \circ_{\mathbf{T}} \rangle \quad (1)$$

is a function  $F : \mathbf{S} \rightarrow \mathbf{T}$  such that for all  $x, y \in \mathbf{S}$ ,

$$F(x \circ_{\mathbf{S}} y) = F(x) \circ_{\mathbf{T}} F(y). \quad (2)$$

Note that we use  $F : \mathbf{S} \rightarrow \mathbf{T}$  when we want to highlight the function between sets, and we use  $F : \mathbf{S} \rightarrow \mathbf{T}$  when we want to highlight the relation between semigroup structures. We think of (2) as a way of saying that the function  $F : \mathbf{S} \rightarrow \mathbf{T}$  is *compatible* with the multiplication operations on  $\mathbf{S}$  and  $\mathbf{T}$ , respectively.

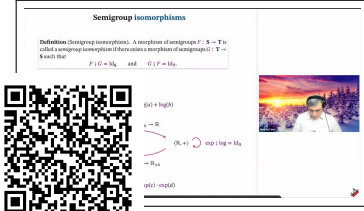
### Definition 7.2 (Identity morphism)

Let  $\mathbf{S}$  be a semigroup. The identity function  $\text{id}_{\mathbf{S}} : \mathbf{S} \rightarrow \mathbf{S}$  is always a morphism of semigroups. We can easily check that (2) is satisfied:

$$\text{id}_{\mathbf{S}}(x \circ_{\mathbf{S}} y) = x \circ_{\mathbf{S}} y = \text{id}_{\mathbf{S}}(x) \circ_{\mathbf{S}} \text{id}_{\mathbf{S}}(y). \quad (3)$$

We call this the *identity morphism* of  $\mathbf{S}$ .

Watch: *Semigroup isomorphisms* (3 minutes).



### Definition 7.3 (Semigroup isomorphism)

A morphism of semigroups  $F : \mathbf{S} \rightarrow \mathbf{T}$  is called a *semigroup isomorphism* if there exists a morphism of semigroups  $G : \mathbf{T} \rightarrow \mathbf{S}$  such that

$$F \circ G = \text{id}_{\mathbf{S}} \quad \text{and} \quad G \circ F = \text{id}_{\mathbf{T}}. \quad (4)$$

**Lemma 7.4.** The composition of semigroup morphisms is a morphism:

$$\frac{F : \mathbf{S} \rightarrow \mathbf{T} \quad G : \mathbf{T} \rightarrow \mathbf{U}}{(F \circ G) : \mathbf{S} \rightarrow \mathbf{U}}. \quad (5)$$

**Exercise 15.** Prove Lemma 7.4.

See solution on page 149.

**Example 7.5** (Logarithms and exponentials). The positive reals with multiplication  $\langle \mathbb{R}_{>0}, \cdot \rangle$  is a semigroup. The reals with addition  $\langle \mathbb{R}, + \rangle$  is a semigroup.

Now consider as a bridge between the two: the logarithmic function. We have

$$\log : \mathbb{R}_{>0} \rightarrow \mathbb{R}, \quad (6)$$

and its inverse

$$\exp : \mathbb{R} \rightarrow \mathbb{R}_{>0}. \quad (7)$$

We already know that these are inverse of each other:

$$\begin{aligned} \exp \circ \log &= \text{id}_{\mathbb{R}}, \\ \log \circ \exp &= \text{id}_{\mathbb{R}_{>0}}. \end{aligned} \quad (8)$$

We can verify that  $\log$  is also a semigroup morphism, because of the following property of the logarithms:

$$\log(a \cdot b) = \log(a) + \log(b). \quad (9)$$

Because  $\log$  is a bijection and  $\exp$  is its inverse, it already follows that  $\exp$  is a morphism in the opposite direction. Alternatively we can see that is the case because of the property of the exponential function:

$$\exp(c + d) = \exp(c) \cdot \exp(d). \quad (10)$$

(9) and (10) are both (2) in disguise.

**Example 7.6** (Transition function, continuation of Section 6.6). Consider the map

$$f : \mathbb{R}_{\geq 0} \rightarrow (\mathbb{R}^n \rightarrow \mathbb{R}^n) \quad (11)$$

that associates to a delta  $\delta$  its transition function  $T_\delta$ . Re-reading (36), we can see that it is a morphism between the semigroup  $\langle \mathbb{R}_{\geq 0}, + \rangle$  and the semigroup of endomorphisms of  $\mathbb{R}^n$ .

**Example 7.7** (State dimension of discrete dynamical systems). Consider the semigroup of discrete dynamical systems described in Section 6.2, and denote it by  $\mathbf{S}$ . As we have seen composing two DDS  $f, g$  gives a DDS  $f \circ_{\mathbf{S}} g$ , state of which has the dimension of the state of  $f$  plus the dimension of the state of  $g$ . We now consider a map  $\dim : \mathbf{DDS} \rightarrow \mathbb{N}$ , which given a DDS returns the dimension of its state space. Now, by considering the semigroup  $\langle \mathbb{N}, + \rangle$ , denoted by  $\mathbf{T}$ , one knows

$$\begin{aligned} \dim(f \circ_{\mathbf{S}} g) &= \dim(f) + \dim(g) \\ &= \dim(f) \circ_{\mathbf{T}} \dim(g), \end{aligned} \quad (12)$$

showing that  $f$  is a valid semigroup morphism between the semigroup of DDS and  $\langle \mathbb{N}, + \rangle$ .

#### Graded exercise C.7 (IsoViaTables)

Consider the set  $\mathbf{A} = \{\circlearrowleft, \approx\}$  and the following three composition tables, each of which defines a semigroup structure on  $\mathbf{A}$ .

$\circlearrowleft_1$	$\circlearrowleft$	$\approx$
$\circlearrowleft$	$\circlearrowleft$	$\circlearrowleft$
$\approx$	$\circlearrowleft$	$\approx$

$\circlearrowleft_2$	$\circlearrowleft$	$\approx$
$\circlearrowleft$	$\circlearrowleft$	$\approx$
$\approx$	$\approx$	$\circlearrowleft$

$\circlearrowleft_3$	$\circlearrowleft$	$\approx$
$\circlearrowleft$	$\circlearrowleft$	$\approx$
$\approx$	$\approx$	$\approx$

Which of the three semigroups defined in this way are isomorphic to each other? Justify your answer.

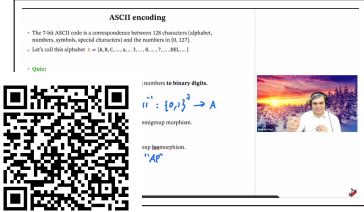
#### Graded exercise C.8 (SemigroupUpToIso)

How many different non-isomorphic semigroups are there with precisely one element? How many with precisely two elements? Can you prove your answer?

#### Graded exercise C.9 (CharacterizeSemigroupIsos)

Let  $F : \mathbf{S} \rightarrow \mathbf{T}$  be a morphism of semigroups. Prove that  $F$  is an isomorphism of semigroups if and only if the function  $F : \mathbf{S} \rightarrow \mathbf{T}$  is bijective.

Watch: *ASCII encoding* (9 minutes).



## 7.2. Encoding as morphism

**Example 7.8** (ASCII code). ASCII encoding takes any alphanumerical characters and symbols into a number between 0 and 127 (Fig. 1). Call `char` the set of those 128 symbols. We can see ASCII encoding as a semigroup morphism of `List char` to the free semigroup on the integers `List [0, 127]`:

$$\text{ASCII} : \text{List char} \rightarrow \text{List}[0, 127]. \quad (13)$$

Because we can also go back, by using the inverse function,

$$\text{ASCII}^{-1} : \text{List}[0, 127] \rightarrow \text{List char}, \quad (14)$$

ASCII encoding is also an isomorphism of semigroups.

**Example 7.9** (ASCII code to binary). Currently, computers use binary to store data. (There were, in fact, *trinary* computers.) In Fig. 1, you can see represented also the binary encoding of each character. Therefore, we can see ASCII as a morphism between `List char` and binary lists `List{0, 1}`.

**Exercise 16.** Show that the morphism

$$\text{ASCII} : \text{List char} \rightarrow \text{List}\{0, 1\} \quad (15)$$

is *not* an isomorphism.

See solution on page 149.

### USASCII code chart

					0	1	2	3	4	5	6	7
Row	0	1	2	3	4	5	6	7				
0	NUL	DLE	SP	0	@	P	\	p				
1	SOH	DC1	!	1	A	Q	a	q				
2	STX	DC2	"	2	B	R	b	r				
3	ETX	DC3	#	3	C	S	c	s				
4	EOT	DC4	\$	4	D	T	d	t				
5	ENQ	NAK	%	5	E	U	e	u				
6	ACK	SYN	&	6	F	V	f	v				
7	BEL	ETB	'	7	G	W	g	w				
8	BS	CAN	(	8	H	X	h	x				
9	HT	EM	)	9	I	Y	i	y				
10	LF	SUB	*	:	J	Z	j	z				
11	VT	ESC	+	;	K	[	k	{				
12	FF	FS	,	<	L	\	l					
13	CR	GS	-	=	M	]	m	}				
14	SO	RS	.	>	N	^	n	~				
15	SI	US	/	?	O	_	o	DEL				

**Figure 1.:** 7-bit US-ASCII encoding.



Is this map a morphism of list semigroups?

*See solution on page 149.*

## 7.4. Monoid morphisms

We have defined semigroup morphisms. A monoid morphism has the same properties, and one additional one: the constraint that it be compatible with the identity elements.

### Definition 7.11 (Monoid morphism)

A morphism  $F : \mathbf{M} \rightarrow \mathbf{N}$  between monoids

$$\mathbf{M} = \langle \mathbf{M}, \circ_{\mathbf{M}}, \text{id}_{\mathbf{M}} \rangle \quad \text{and} \quad \mathbf{N} = \langle \mathbf{N}, \circ_{\mathbf{N}}, \text{id}_{\mathbf{N}} \rangle \quad (25)$$

is a function  $F : \mathbf{M} \rightarrow \mathbf{N}$  such that for all  $x, y \in \mathbf{M}$ ,

$$F(x \circ_{\mathbf{M}} y) = F(x) \circ_{\mathbf{N}} F(y), \quad (26)$$

and

$$F(\text{id}_{\mathbf{M}}) = \text{id}_{\mathbf{N}}. \quad (27)$$

**Example 7.12.** The set  $\mathbf{M} = \{-1, 0, +1\}$ , together with multiplication of whole numbers and with 1 as neutral element, forms a monoid. The inclusion map  $\mathbf{M} \rightarrow \mathbb{Z}$  is a morphism of monoids.

**Example 7.13.** Consider the monoid  $\mathbf{N}$  in Example 6.29 and the monoid  $\mathbf{M} = \langle \mathbb{N}, +, 0 \rangle$ . Define a map

$$\text{length} : \mathbf{N} \rightarrow \mathbf{M} \quad (28)$$

that maps each string to its length. This is a monoid morphism, since:

$$\text{length}(x \circ_{\mathbf{N}} y) = \text{length}(x) +_{\mathbf{M}} \text{length}(y). \quad (29)$$

In other words, the length of the concatenation of two lists is the sum of the lengths of the two lists.

### Definition 7.14 (Identity morphism)

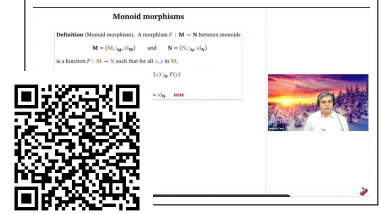
Let  $\mathbf{M}$  be a monoid. Similar to the case of semigroups, the identity function induces a morphism of monoids  $\text{id}_{\mathbf{M}} : \mathbf{M} \rightarrow \mathbf{M}$ .

### Definition 7.15 (Monoid isomorphism)

A morphism of monoids  $F : \mathbf{M} \rightarrow \mathbf{N}$  is called a *monoid isomorphism* if there is a morphism of monoids  $G : \mathbf{N} \rightarrow \mathbf{M}$  such that

$$F \circ G = \text{id}_{\mathbf{M}} \quad \text{and} \quad G \circ F = \text{id}_{\mathbf{N}}. \quad (30)$$

Watch: *Monoid morphisms* (7 minutes).





**Graded exercise C.10** (MorphismMonoidIsomorphism)

Prove: a morphism of monoids  $F : \mathbf{M} \rightarrow \mathbf{N}$  is an isomorphism of monoids if and only if the function  $F : \mathbf{M} \rightarrow \mathbf{N}$  is bijective.

**Graded exercise C.11** (TraceAndDeterminant)

Let  $\mathbb{R}^{n \times n}$  denote the set of real  $n \times n$  matrices,  $n \in \mathbb{N}$ . These form a monoid  $\langle \mathbb{R}^{n \times n}, \cdot, \mathbf{1} \rangle$ , with matrix multiplication as composition, and the identity matrix as neutral element. They also form a monoid  $\langle \mathbb{R}^{n \times n}, +, 0 \rangle$  with matrix addition as composition, and the zero matrix as neutral element. We also note that that  $\langle \mathbb{R}, \cdot, 1 \rangle$  and  $\langle \mathbb{R}, +, 0 \rangle$  are two different monoid structures on  $\mathbb{R}$ .

Recall that the trace of a real  $n \times n$  matrix is the sum of its diagonal entries. This defines a function

$$\text{Tr} : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}. \quad (31)$$

Computing the determinant also corresponds to a function

$$\det : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}. \quad (32)$$

1. Does  $\text{Tr}$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, \cdot, \mathbf{1} \rangle \rightarrow \langle \mathbb{R}, \cdot, 1 \rangle$ ?
2. Does  $\text{Tr}$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, \cdot, \mathbf{1} \rangle \rightarrow \langle \mathbb{R}, +, 0 \rangle$ ?
3. Does  $\det$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, \cdot, \mathbf{1} \rangle \rightarrow \langle \mathbb{R}, \cdot, 1 \rangle$ ?
4. Does  $\det$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, \cdot, \mathbf{1} \rangle \rightarrow \langle \mathbb{R}, +, 0 \rangle$ ?
5. Does  $\text{Tr}$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, +, 0 \rangle \rightarrow \langle \mathbb{R}, \cdot, 1 \rangle$ ?
6. Does  $\text{Tr}$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, +, 0 \rangle \rightarrow \langle \mathbb{R}, +, 0 \rangle$ ?
7. Does  $\det$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, +, 0 \rangle \rightarrow \langle \mathbb{R}, \cdot, 1 \rangle$ ?
8. Does  $\det$  define a morphism of monoids  $\langle \mathbb{R}^{n \times n}, +, 0 \rangle \rightarrow \langle \mathbb{R}, +, 0 \rangle$ ?

Short answers (without proof) are fine.

## 7.5. Group morphisms

After semigroup morphisms and monoid morphisms, we define group morphisms.

### Definition 7.16 (Group morphism)

A morphism  $F : \mathbf{G} \rightarrow \mathbf{H}$  between groups

$$\mathbf{G} = \langle \mathbf{G}, \circ_{\mathbf{G}}, \text{id}_{\mathbf{G}}, \text{inv}_{\mathbf{G}} \rangle \quad \text{and} \quad \mathbf{H} = \langle \mathbf{H}, \circ_{\mathbf{H}}, \text{id}_{\mathbf{H}}, \text{inv}_{\mathbf{H}} \rangle \quad (33)$$

is a function  $F : \mathbf{G} \rightarrow \mathbf{H}$  such that for all  $x, y \in \mathbf{G}$ ,

$$F(x \circ_{\mathbf{G}} y) = F(x) \circ_{\mathbf{H}} F(y). \quad (34)$$

What could be surprising is that, while a group has more structure than a monoid, there are fewer conditions than in the definition of monoid morphism.

Where are the equations

$$F(\text{id}_{\mathbf{G}}) = \text{id}_{\mathbf{H}} \quad (35)$$

and

$$F(\text{inv}_{\mathbf{G}}(x)) = \text{inv}_{\mathbf{H}}(F(x)) ? \quad (36)$$

The answer is that they are not needed, because they can be deduced from the group axioms (and so we omit them, because they don't need to be checked when we want to know if something is a group morphism or not).

**Exercise 18.** Prove that (35) and (36) are implied by Def. 7.16.

*See solution on page 150.*

**Exercise 19.** Let  $\mathbf{G} = \{+1, -1, +i, -i\}$  where  $i$  is the imaginary unit. Consider the group  $\mathbf{G} = \langle \mathbf{G}, \cdot, 1, \text{inv}_{\mathbf{G}} \rangle$  and the group  $\mathbf{H}$  of all integers under addition. Prove that  $F : \mathbf{H} \rightarrow \mathbf{G}$  such that  $f(n) = i^n$  for all  $n \in \mathbf{H}$  is a group morphism.

*See solution on page 150.*

**Example 7.17.** Consider the group

$$\mathbf{G} = \left\langle \mathbb{R}^{2 \times 2}, +, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, - \right\rangle \quad (37)$$

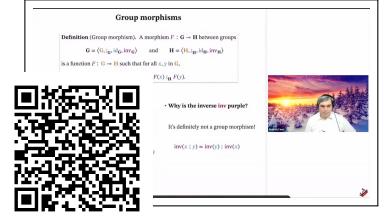
of real  $2 \times 2$  matrices, together with sum of matrices as a binary operation, “zero” matrix as the neutral element, and the “-” operation as inverse. Furthermore, consider the group  $\langle \mathbb{R}, +, 0, - \rangle$ . Taking the *trace* of a matrix corresponds to a group morphism. Indeed, the operation

$$\begin{aligned} \text{Tr} : \mathbb{R}^{2 \times 2} &\rightarrow \mathbb{R}, \\ \begin{bmatrix} a & b \\ c & d \end{bmatrix} &\mapsto a + d, \end{aligned} \quad (38)$$

satisfies the required condition:

$$\begin{aligned} \text{Tr} \left( \begin{bmatrix} a & b \\ c & d \end{bmatrix} \circ_{\mathbf{G}} \begin{bmatrix} e & f \\ g & h \end{bmatrix} \right) &= \text{Tr} \left( \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix} \right) \\ &= \text{Tr} \left( \begin{bmatrix} a & b \\ c & d \end{bmatrix} \right) \circ_{\mathbf{H}} \text{Tr} \left( \begin{bmatrix} e & f \\ g & h \end{bmatrix} \right) \\ &= (a+d) \circ_{\mathbf{H}} (e+h). \end{aligned} \quad (39)$$

Watch: *Group morphisms* (4 minutes).



**Example 7.18** (Square matrices with full rank). Fix an integer  $n \geq 1$  and consider the set of square matrices with full rank  $\mathbf{A} \in \mathbb{R}^{n \times n}$ , which is to say  $\det(\mathbf{A}) \neq 0$ . This set, equipped with the usual matrix multiplication as the binary operation ( $\mathbf{A} \circledast \mathbf{B} := \mathbf{AB}$ ), the identity matrix  $\mathbf{1}$  as the neutral element, and matrix inverse as the inverse ( $\text{inv}(\mathbf{A}) := \mathbf{A}^{-1}$ ), forms a group. Furthermore, note that for this type of matrices, we have the properties:

1.  $\det(\mathbf{A} \mathbf{B}) = \det(\mathbf{A}) \cdot \det(\mathbf{B})$ ;
2.  $\det(\mathbf{A}^{-1}) = (\det(\mathbf{A}))^{-1}$ ;
3.  $(\mathbf{A} \mathbf{B})^{-1} = \mathbf{B}^{-1} \mathbf{A}^{-1}$ .

This makes  $\det$  a group morphism from the group of invertible square matrices to the real numbers with multiplication.

As before for semigroups and monoids, the identity map on the underlying set of a group is a group morphism.

**Definition 7.19** (Identity morphism)

Given a group  $\mathbf{G} = \langle \mathbf{G}, \circledast, \text{id}, \text{inv} \rangle$ , the identity function  $\mathbf{G} \rightarrow \mathbf{G}$  induces a morphism of groups  $\text{id}_{\mathbf{M}} : \mathbf{M} \rightarrow \mathbf{M}$ .

**Definition 7.20** (Group isomorphism)

A morphism of monoids  $F : \mathbf{G} \rightarrow \mathbf{H}$  is called a *group isomorphism* if there is a morphism of groups  $G : \mathbf{H} \rightarrow \mathbf{G}$  such that

$$F \circledast G = \text{id}_{\mathbf{G}} \quad \text{and} \quad G \circledast F = \text{id}_{\mathbf{H}}. \quad (40)$$

## 7.6. Generators and relations

### Generating subsets

In Example 6.12 we considered a set of states

$$\mathbf{A} = \{\text{sprout, young, mature, old, dead}\}, \quad (41)$$

a function  $f : \mathbf{A} \rightarrow \mathbf{A}$ , and the semigroup

$$\mathbf{S} = \{f^n \mid n \in \mathbb{N}\}. \quad (42)$$

Note that  $\mathbf{S}$  has a special form: all of its elements can be expressed in terms one of its elements,  $f$ , and the multiplication operation (which, in this case, is function composition). To describe this state of affairs we say that  $\mathbf{S}$  is *generated* by the element  $f$ .

#### Definition 7.21 (Generating subsets)

Let  $\mathbf{S} = \langle \mathbf{S}, \circ \rangle$  be a semigroup, and let  $\mathbf{A} \subseteq \mathbf{S}$  be a subset. We say that  $\mathbf{S}$  is *generated* by  $\mathbf{A}$  if every element of  $\mathbf{S}$  can be expressed as a finite composition of elements of  $\mathbf{A}$ .

**Remark 7.22.** Mutatis mutandis, the same definition also holds for monoids. For groups, we say  $\mathbf{A}$  generates the group if every element of the group can be expressed as a finite composition of elements of  $\mathbf{A}$  or their inverses.

**Example 7.23.** Consider Example 6.10, where elements of the semigroup  $\mathbf{S}$  were non-empty lists built using the elements of the “alphabet” set  $\mathbf{A} = \{\bullet, \cdot\}$ . In this case,  $\mathbf{S}$  is generated by  $\mathbf{A}$ .

**Example 7.24.** Consider the natural numbers (without zero) as a semigroup, where addition is the semigroup composition operation (see Example 6.7). This semigroup is generated by the subset  $\{1\}$ .

### Relations

Let us return to the semigroup (42). Recall that  $f$  was defined by

$$f(\text{sprout}) = \text{young}, \quad (43)$$

$$f(\text{young}) = \text{mature}, \quad (44)$$

$$f(\text{mature}) = \text{old}, \quad (45)$$

$$f(\text{old}) = \text{dead}, \quad (46)$$

$$f(\text{dead}) = \text{dead}. \quad (47)$$

Observe that the function  $f^4$  will map all elements of  $\mathbf{A}$  to the element “dead”. For example, if we start with the element “sprout”, the result of applying  $f$  four times is

$$\text{sprout} \xrightarrow{f} \text{young} \xrightarrow{f} \text{mature} \xrightarrow{f} \text{old} \xrightarrow{f} \text{dead}. \quad (48)$$

Note also that for any  $n \geq 4$ , the function  $f^n$  will map all elements of  $\mathbf{A}$  to the element “dead”. If we consider  $f^6$ , for example, then, for any  $x \in \mathbf{A}$ ,

$$f^6(x) = f^2(f^4(x)) = f^2(\text{dead}) = f(f(\text{dead})) = f(\text{dead}) = \text{dead}. \quad (49)$$

It follows that all  $f^n$ , for  $n \geq 4$ , are actually *all the same map*: the one that sends

every state to the dead state. Thus,  $S = \{f^n \mid n \in \mathbb{N}\}$  actually only has at most four elements:  $f$ ,  $f^2$ ,  $f^3$ , and  $f^4$ .

**Graded exercise C.12** (CheckRelations)

Are any of the four maps  $f$ ,  $f^2$ ,  $f^3$ , and  $f^4$  actually equal? Justify your answer by argumentation or by explicitly checking via calculation.

When two elements which a priori could be distinct from each other (such as  $f^6$  and  $f^4$  above, for example) turn out to be equal, we call this a *relation* between the elements of  $S$ .

**Definition 7.25**

A *relation* on a semigroup  $\langle S, \circ \rangle$  is an equation between compositions of elements of  $S$ .

**Remark 7.26.** Again, we have analogous definitions for monoids and groups. In these cases, we interpret the neutral element  $\text{id}$  as a “zero-fold” multiplication, so it can also be part of equations that express relations.

**Remark 7.27.** This is not the same notion as that of a (binary) relation, which was the topic of Chapter 5 and takes up a much more important role in this book than the notion that we are discussing here.

**Example 7.28.** For the semigroup (42), the relations  $f^5 = f^4$ ,  $f^6 = f^5$ , and  $f^6 = f^4$ , etc. are satisfied. However, the relation  $f^3 = f$  is not satisfied, for example.

**Example 7.29.** Consider the semigroup  $\langle \mathbb{N}, + \rangle$ . The equation  $l + k = k + l$  is an example of a relation that holds for all  $l, k \in \mathbb{N}$ .

**Example 7.30.** Consider the group  $G$  discussed in Example 6.45, where

$$G = \{1, e^{\frac{1}{3}2\pi i}, e^{\frac{2}{3}2\pi i}\} \subseteq \mathbb{C} \quad (50)$$

and the composition operation is multiplication of complex numbers. The element  $x := e^{\frac{1}{3}2\pi i}$  satisfies the relation  $x^3 = \text{id}_G$ .

**Example 7.31.** Consider the group  $G$  given in Example 6.47 which describe symmetries of a rectangle. We had

$$G = \{I, V, H, R\} \quad (51)$$

where  $I = \text{id}$  corresponds to “doing nothing”,  $V$  is reflecting the rectangle along its long axis,  $H$  is reflecting on the short axis, and  $R$  is rotation by 180 degrees. In this group, the relations  $V^2 = I$ ,  $H^2 = I$ ,  $R^2 = I$  are satisfied, for example.

## Freeness

When, in Def. 7.21 we spoke about the semigroup  $S = \langle S, \circ \rangle$  being generated by a subset  $A \subseteq S$ , we supposed that we already had a semigroup  $S$  to work with. However, if we start with just a set, say  $A = \{\bullet, \circ\}$ , then we saw in Example 6.10 that we can build a semigroup from this set by considering lists of elements of  $A$ , with concatenation as the composition operation. The resulting semigroup in that example has a special characteristic: its elements do not satisfy any relations

other than the ones that are required by the definition of a semigroup, namely those relations dictated by the associative law. Such a semigroup is called *free*. If we think of relations as “constraints” (they are equations) between the elements of a semigroup, then free semigroups are “free of constraints”.

For a given set, say  $\mathbf{A} = \{\bullet, \circ\}$ , there will in general be different ways of formally building a free semigroup from it. For instance, instead of considering lists of elements of  $\mathbf{A}$

$$[\bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet], \quad (52)$$

we could instead consider strings of elements

$$\bullet \circ \bullet \circ \bullet \circ \bullet, \quad (53)$$

or tuples of elements

$$\langle \bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet, \circ, \bullet \rangle, \quad (54)$$

both of which could also be composed in a way which is analogous to concatenation.

A common feature of all three of these formalizations is that we are writing a finite sequence of elements of  $\mathbf{A}$ , keeping account of the ordering. Both approaches will in fact build a semigroup from the set  $\mathbf{A}$  which is *free*. And in both cases there is a natural way of seeing  $\mathbf{A}$  as *generating* the resulting semigroup. However, the two set-ups are *formally* distinct because we are using a different way of writing things down with symbols. We will see in a later chapter that the resulting semigroups are essentially “the same” (they are isomorphic) and their “freeness” can be given an elegant characterization.

Because in fact *all* free semigroups constructed from  $\mathbf{A}$  are “the same”, independent of the formal symbolic specifics of how they are constructed, we refer to them all as *the* free semigroup generated by  $\mathbf{A}$ . Furthermore, if we were to work with a set  $\mathbf{B} = \{a, b\}$  instead of  $\mathbf{A} = \{\bullet, \circ\}$ , and generate a free semigroup from  $\mathbf{B}$ , then this would also produce a semigroup which is “the same”. Therefore, sometimes one speaks simply of “the free group on two generators”, in view of the fact that the essential feature is that both  $\mathbf{A}$  and  $\mathbf{B}$  have two elements.

Note that although  $\mathbf{A}$  and  $\mathbf{B}$  each only have two elements, the free semigroups that they generate will have infinitely many elements. Indeed, there are infinitely many lists that we can build from the elements of  $\mathbf{A}$ . As we concatenate lists, the resulting compositions grow longer and longer, and there are no relations which would allow us to “simplify” a string to one which is shorter.

## 7.7. ○ Morphisms

We define an interface `SemigroupMorphism` as follows. The methods `source()` and `target()` give the two semigroups linked by the morphism. The method `mapping()` returns the mapping between the sets. This mapping is assumed to satisfy the conditions of a semigroup morphism.

**Listing 48:** The `SemigroupMorphism` interface.

```
class SemigroupMorphism(Generic[E1, E2], ABC):
    @abstractmethod
    def source(self) -> Semigroup[E1]: ...

    @abstractmethod
    def target(self) -> Semigroup[E2]: ...

    @abstractmethod
    def mapping(self) -> Mapping[E1, E2]: ...
```

The derived interfaces `MonoidMorphism` and `GroupMorphism` are defined in the obvious way.

**Listing 49:** The `MonoidMorphism` interface.

```
class MonoidMorphism(Generic[E1, E2], SemigroupMorphism[E1, E2], ABC):
    @abstractmethod
    def source(self) -> Monoid[E1]: ...

    @abstractmethod
    def target(self) -> Monoid[E2]: ...
```

**Listing 50:** The `GroupMorphism` interface.

```
class GroupMorphism(Generic[E1, E2], MonoidMorphism[E1, E2], ABC):
    @abstractmethod
    def source(self) -> Group[E1]: ...

    @abstractmethod
    def target(self) -> Group[E2]: ...
```

We also define the corresponding finite versions as `FiniteSemigroupMorphism`, `FiniteMonoidMorphism`, `FiniteGroupMorphism`.

## Representation

The concrete representation for morphisms is

```
source: ...
target: ...
mapping: ...
```

where `source` and `target` are representations of semigroups/monoids/groups, and `mapping` is the representation of a `FiniteMap`.

The following are three very similar exercises for loading the three types of exercises.

**Code exercise C.5** (`TestFiniteSemigroupMorphismRepresentation`)  
Implement the interface in Listing 51.

Test using:

```
act4e-check TestFiniteSemigroupMorphismRepresentation
```

**Listing 51:** The `FiniteSemigroupMorphismRepresentation` interface.

```
class FiniteSemigroupMorphismRepresentation(  
  Serializer[FiniteSemigroupMorphism[Any, Any], FiniteSemigroupMorphism_desc], ABC  
) :  
  ...
```

**Code exercise C.6** (`TestFiniteMonoidMorphismRepresentation`)

Implement the interface in Listing 52.

Test using:

```
act4e-check TestFiniteMonoidMorphismRepresentation
```

**Listing 52:** The `FiniteMonoidMorphismRepresentation` interface.

```
class FiniteMonoidMorphismRepresentation(  
  Serializer[FiniteMonoidMorphism[Any, Any], FiniteMonoidMorphism_desc], ABC  
) :  
  pass
```

**Code exercise C.7** (`TestFiniteGroupMorphismRepresentation`)

Implement the interface in Listing 53.

Test using:

```
act4e-check TestFiniteGroupMorphismRepresentation
```

**Listing 53:** The `FiniteGroupMorphismRepresentation` interface.

```
class FiniteGroupMorphismRepresentation(  
  Serializer[FiniteGroupMorphism[Any, Any], FiniteGroupMorphism_desc], ABC  
) :  
  ...
```



## Checking properties

**Code exercise C.8** (TestFiniteSemigroupMorphismsChecks)

Implement the interface in Listing 54.

Test using:

```
act4e-check TestFiniteSemigroupMorphismsChecks
```

**Listing 54:** The `FiniteSemigroupMorphismsChecks` interface.

```
class FiniteSemigroupMorphismsChecks(ABC):
    @abstractmethod
    def is_semigroup_morphism(self, a: FiniteSemigroup[A], b: FiniteSemigroup[B], f: FiniteMap[A, B]) -> bool:
        """Check that the triple forms a semigroup morphism."""

    @abstractmethod
    def is_monoid_morphism(self, a: FiniteMonoid[A], b: FiniteMonoid[B], f: FiniteMap[A, B]) -> bool:
        """Check that the triple forms a monoid morphism."""

    @abstractmethod
    def is_group_morphism(self, a: FiniteGroup[A], b: FiniteGroup[B], f: FiniteMap[A, B]) -> bool:
        """Check that the triple forms a monoid morphism."""
```



# 8. Actions

We have talked about semigroups, monoids, and groups as mathematical structures in isolation. In this chapter we discuss the concept of *action* of a semigroup on *another set*. For example, we discuss how linear transformations are a semigroup that *act* on vectors.

- 8.1 A simplest dynamical system . . 142
- 8.2 Actions . . . . . 143
- 8.3 Contravariant actions . . . . . 146
- 8.4 Actions of matrix groups . . . . . 147

## 8.1. A simplest dynamical system

Dynamical systems are an important topic in engineering, and there are various ways to formulate this concept mathematically. At a very basic level, there are usually the following ingredients:

1. a *state space* which comprises the possible states of the system in question
2. a *dynamics rule* which governs how states of the system may change

Typically, there is a model of *time* involved (e.g. either in discrete time-steps, or as continuous time) for describing how states change over time, or, instead of a model of time, one may use a set of *events* as the triggers for changes of state.

## 8.2. Actions

There are many examples in engineering of situations where there is a semigroup whose elements can be thought of transformations that are applied to a second space. This concept is formalized as an “action”.

**Definition 8.1** (Semigroup covariant action (preliminary version))  
A covariant semigroup action of a semigroup  $\mathbf{S}$  on a set  $\mathbf{A}$  is a map

$$\text{Covact} : \mathbf{S} \times \mathbf{A} \rightarrow \mathbf{A} \quad (1)$$

such that, for all  $a \in \mathbf{A}$ ,

$$\text{Covact}(y, \text{Covact}(x, a)) = \text{Covact}(x \circ_{\mathbf{S}} y, a). \quad (2)$$

We read (2) as saying that, if we first apply the action of  $x$  to obtain  $\text{Covact}(x, a)$ , and then we apply the action of  $y$   $\text{Covact}(y, -)$ , it is the same thing as applying the action of  $x \circ_{\mathbf{S}} y$ .

This definition is the standard algebraic definition. We will compress it a bit using the language of category theory.

Let’s introduce Dr. Haskell Curry (Fig. 1). His first name, Haskell, named the language. His last name, Curry, named the operation of *currying* that we are going to need.

Curry noticed that for describing the domain of a function, we do not need to have the cartesian product. If we have a function

$$f : \mathbf{A} \times \mathbf{B} \rightarrow \mathbf{C}, \quad (3)$$

we can rewrite it as a function of higher type

$$f : \mathbf{A} \rightarrow (\mathbf{B} \rightarrow \mathbf{C}). \quad (4)$$

If you feel like a cool computer scientist, you can also drop the parenthesis and write

$$f : \mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{C}, \quad (5)$$

because the expression is not ambiguous, as  $\rightarrow$  associates from the left.

This specifies  $f$  as a function that, given a  $\mathbf{A}$ , provides a function that, given a  $\mathbf{B}$ , provides a  $\mathbf{C}$ ; this is the same as a function that needs the two arguments  $\mathbf{A}$  and  $\mathbf{B}$  before giving the  $\mathbf{C}$ . To describe the isomorphism we can write it as

$$\mathbf{A} \rightarrow (\mathbf{B} \rightarrow \mathbf{C}) \simeq \mathbf{A} \times \mathbf{B} \rightarrow \mathbf{C}. \quad (6)$$

Now armed with currying, we can take a second look at (1) and realize that we can rewrite

$$\text{Covact} : \mathbf{S} \times \mathbf{A} \rightarrow \mathbf{A} \quad (7)$$

as

$$\text{Covact} : \mathbf{S} \rightarrow (\mathbf{A} \rightarrow \mathbf{A}). \quad (8)$$

We gave a name to functions of type  $\mathbf{A} \rightarrow \mathbf{A}$ : these are the endomorphisms of  $\mathbf{A}$ , written as  $\text{End}(\mathbf{A})$ . Thus, we rewrite this as

$$\text{Covact} : \mathbf{S} \rightarrow \text{End}(\mathbf{A}). \quad (9)$$

Now we take a second look at (2):

$$\text{Covact}(y, \text{Covact}(x, a)) = \text{Covact}(x \circ_{\mathbf{S}} y, a). \quad (10)$$

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.



Figure 1.: Haskell Curry (1912-1980)

If we rewrite it as an equality of functions, we obtain

$$\text{Covact}(x) \circ_{\text{End}(\mathbf{A})} \text{Covact}(y) =_{\text{End}(\mathbf{A})} \text{Covact}(x \circ_{\mathbf{S}} y). \quad (11)$$

Looking at (9) and (11) we recognize that together they indicated that  $\text{Covact}$  is a semigroup morphism (Def. 7.1). This brings us to a more compact description of what a semigroup action is.

**Definition 8.2** (Semigroup covariant action)

A *covariant semigroup action* of a semigroup  $\mathbf{S}$  onto a set  $\mathbf{A}$  is a semigroup morphism

$$\text{Covact} : \mathbf{S} \rightarrow \text{End}(\mathbf{A}). \quad (12)$$

As it turns out, what could look like a new notion, is actually a special case of a general notion we already encountered, namely the notion of semigroup morphism.

For completeness, we also define monoid actions and group actions.

**Definition 8.3** (Monoid covariant action)

A *covariant monoid action* of a monoid  $\mathbf{M}$  onto a set  $\mathbf{A}$  is a monoid morphism

$$\text{Covact} : \mathbf{M} \rightarrow \text{End}(\mathbf{A}). \quad (13)$$

The neutral element of the monoid  $\text{End}(\mathbf{A})$  is the identity function  $\text{id}_{\mathbf{A}}$ . Thus, a monoid action must map the neutral element of  $\mathbf{M}$  to  $\text{id}_{\mathbf{A}}$ .

For defining a group covariant action, we must introduce a slight variation. The endomorphisms  $\text{End}(\mathbf{A})$  are not a group, because they also contain non-invertible maps. Recall that an invertible endomorphism is called an *automorphism*, and that  $\text{Aut}(\mathbf{A})$ , the set of automorphisms of  $\mathbf{A}$ , comes naturally equipped with a group structure.

**Definition 8.4** (Group covariant action)

A *covariant group action* of a group  $\mathbf{G}$  onto a set  $\mathbf{A}$  is a group morphism

$$\text{Covact} : \mathbf{G} \rightarrow \text{Aut}(\mathbf{A}). \quad (14)$$

**Lemma 8.5.** The data of a covariant group action

$$\text{Covact} : \mathbf{G} \rightarrow \text{Aut}(\mathbf{A}) \quad (15)$$

is equivalent to the data of having a function

$$\alpha : \mathbf{G} \times \mathbf{A} \rightarrow \mathbf{A}, \quad (16)$$

which satisfies the conditions

1.  $\alpha(\text{id}_{\mathbf{G}}, a) = a \quad \forall a \in \mathbf{A};$
2.  $\alpha(y, \alpha(x, a)) = \alpha(x \circ_{\mathbf{G}} y, a) \quad \forall a \in \mathbf{A}, \forall x, y \in \mathbf{G}.$

**Graded exercise C.13** (UncurryingGroupActions)

Prove Lemma 8.5.

**Graded exercise C.14** (MatrixMultAction)

Let  $\mathbf{A} = \mathbb{R}^n$ , and let  $\mathrm{GL}(n, \mathbb{R})$  be the group of invertible  $n \times n$  matrices. Let

$$\begin{aligned} \alpha : \mathrm{GL}(n, \mathbb{R}) \times \mathbf{A} &\rightarrow \mathbf{A} \\ \langle \mathbf{M}, \mathbf{v} \rangle &\mapsto \mathbf{M}\mathbf{v} \end{aligned} \tag{17}$$

be the usual multiplication of matrices with vectors. Check that (17) defines a covariant group action of  $\mathrm{GL}(n, \mathbb{R})$  on  $\mathbf{A}$ .

### 8.3. Contravariant actions

There are two versions for actions: a “covariant” and a “contravariant” version, depending on the order of the transformations. \*

**Definition 8.6** (Semigroup contravariant action (preliminary version))  
A *contravariant semigroup action* of a semigroup  $\mathbf{S}$  on a set  $\mathbf{A}$  is a map

$$\text{Contravact} : \mathbf{S} \times \mathbf{A} \rightarrow \mathbf{A} \quad (18)$$

such that, for all  $a \in \mathbf{A}$ ,

$$\text{Contravact}(y, \text{Contravact}(x, a)) = \text{Contravact}(y \circ_{\mathbf{S}} x, a). \quad (19)$$

In contrast to (2), (19) says that, if we first apply the action of  $x$  to obtain  $\text{Covact}(x, a)$ , and then we apply the action of  $y$   $\text{Covact}(y, -)$ , it is the same thing as applying the action of  $y \circ x$  (note the inverted order).

A contravariant semigroup action can be defined as a covariant action of *the opposite* semigroup. And similarly for monoids and groups.

**Definition 8.7** (Semigroup contravariant action)

A *contravariant semigroup action* of a semigroup  $\mathbf{S}$  onto a set  $\mathbf{A}$  is a semigroup morphism

$$\text{Contravact} : \mathbf{S}^{\text{op}} \rightarrow \text{End}(\mathbf{A}). \quad (20)$$

\* Traditionally, people have used the names “left” and “right” to describe the two types. However, those names are connected to a particular choice of notation. We name the two versions by highlighting the properties of covariance and contravariance.

## 8.4. Actions of matrix groups

### Special Euclidean group

**Definition 8.8** (General Euclidean group  $E(n, \mathbb{R})$ )

The general (real) Euclidean group of order  $n$ , written  $E(n, \mathbb{R})$ , is the group of  $(n + 1) \times (n + 1)$  real matrices of the form

$$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}, \quad (21)$$

where  $\mathbf{R} \in O(n, \mathbb{R})$  and  $\mathbf{t} \in \mathbb{R}^n$ .

**Definition 8.9**

Special Euclidean group  $SE(n, \mathbb{R}^n)$  The special (real) Euclidean group of order  $n$ , written  $SE(n, \mathbb{R}^n)$ , is the group of  $(n + 1) \times (n + 1)$  real matrices of the form

$$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}, \quad (22)$$

where  $\mathbf{R} \in SO(n, \mathbb{R})$  and  $\mathbf{t} \in \mathbb{R}^n$ .

The groups  $SE(2, \mathbb{R})$  and  $SE(3, \mathbb{R})$  are particularly important in robotics because they represent the roto-translations of the plane and 3D space, respectively.

From (21) we know we can represent one by a pair  $\langle \mathbf{R}, \mathbf{t} \rangle$ , with  $\mathbf{R} \in SO(n, \mathbb{R})$  and  $\mathbf{t} \in \mathbb{R}^n$ .

If we look at how matrices compose, we get

$$\begin{bmatrix} \mathbf{R}_2 & \mathbf{t}_2 \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 & \mathbf{t}_1 \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_2 \mathbf{R}_1 & \mathbf{R}_2 \mathbf{t}_1 + \mathbf{t}_2 \\ \mathbf{0} & 1 \end{bmatrix}. \quad (23)$$

The formula for composition is

$$\langle \mathbf{R}_1, \mathbf{t}_1 \rangle \circ_{SE(n, \mathbb{R}^n)} \langle \mathbf{R}_2, \mathbf{t}_2 \rangle = \langle \mathbf{R}_2 \mathbf{R}_1, \mathbf{R}_2 \mathbf{t}_1 + \mathbf{t}_2 \rangle. \quad (24)$$

The group  $SE(n, \mathbb{R}^n)$  induces a transformation on the points of  $\mathbb{R}^n$ . We are going to call this an *action*.

The action is the following function:

$$\begin{aligned} \text{apply} : SE(n, \mathbb{R}^n) \times \mathbb{R}^n &\rightarrow \mathbb{R}^n, \\ \langle \langle \mathbf{R}, \mathbf{t} \rangle, \mathbf{p} \rangle &\mapsto \mathbf{R}\mathbf{p} + \mathbf{t}. \end{aligned} \quad (25)$$

Given a roto-translation and a point, the function returns the roto-translated point. We can also see this in matrix form as follows. We need to substitute for a

**Table 8.1.:** Matrix groups

$GL(n, \mathbb{R})$	General linear group	arbitrary linear transformations
$SL(n, \mathbb{R})$	Special linear group	invertible linear transformations
$O(n, \mathbb{R})$	Orthogonal group	preserve length of vectors
$SO(n, \mathbb{R})$	Special orthogonal group	rotations
$E(n, \mathbb{R})$	Euclidean group	preserve distances and angles
$SE(n, \mathbb{R}^n)$	Special Euclidean group	rigid motions



point  $\mathbf{p} \in \mathbb{R}^n$  a homogenous point  $\begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \in \mathbb{R}^{n+1}$ .

$$\begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}\mathbf{p} + \mathbf{t} \\ 1 \end{bmatrix}. \quad (26)$$

If we apply two rototranslations, first  $x = \langle \mathbf{R}_x, \mathbf{t}_x \rangle$  and then  $y = \langle \mathbf{R}_y, \mathbf{t}_y \rangle$ , we find:

$$\begin{aligned} \text{apply}(\langle \mathbf{R}_y, \mathbf{t}_y \rangle, \text{apply}(\langle \mathbf{R}_x, \mathbf{t}_x \rangle, \mathbf{p})) &= \text{apply}(\langle \mathbf{R}_y, \mathbf{t}_y \rangle, \mathbf{R}_x\mathbf{p} + \mathbf{t}_x) \\ &= \mathbf{R}_y\mathbf{R}_x\mathbf{p} + \mathbf{R}_y\mathbf{t}_x + \mathbf{t}_y. \end{aligned} \quad (27)$$

It is easy to see that it is equal to compose the two transformations in the inverse order

$$\langle \mathbf{R}_x, \mathbf{t}_x \rangle \circ_{\text{SE}(n, \mathbb{R}^n)} \langle \mathbf{R}_y, \mathbf{t}_y \rangle = \langle \mathbf{R}_y\mathbf{R}_x, \mathbf{R}_y\mathbf{t}_x + \mathbf{t}_y \rangle, \quad (28)$$

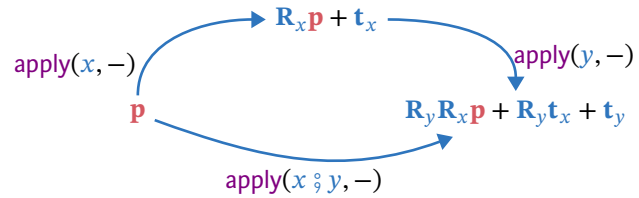
and then apply it to the object

$$\text{apply}(\langle \mathbf{R}_y\mathbf{R}_x, \mathbf{R}_y\mathbf{t}_x + \mathbf{t}_y \rangle, \mathbf{p}) = \mathbf{R}_y\mathbf{R}_x\mathbf{p} + \mathbf{R}_y\mathbf{t}_x + \mathbf{t}_y. \quad (29)$$

Thus, we have proved this property

$$\text{apply}(y, \text{apply}(x, \mathbf{p})) = \text{apply}(x \circ y, \mathbf{p}), \quad (30)$$

which is graphically reported in Fig. 2.



**Figure 2.:** Graphical representation of roto-translation action.

The notion of semigroup action generalizes this property.

## Solutions to selected exercises

**Solution of Exercise 11.** No.

**Solution of Exercise 12.** We show a counterexample. Clearly we have

$$\left( \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} \circ \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) \circ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \\ -1 \end{bmatrix}. \quad (31)$$

However,

$$\begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix} \circ \left( \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \circ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}, \quad (32)$$

violating the associative law.

**Solution of Exercise 13.** Given  $x, y, z \in \mathbb{N}$ , we have:

$$\min(\min(x, y), z) = \min(x, \min(y, z)). \quad (33)$$

**Solution of Exercise 14.** No. Consider the condition  $0 = \max(\text{inv}(x), x)$ . In general, this is true only if  $x = \text{inv}(x) = 0$ .

**Solution of Exercise 15.** We have:

$$\begin{aligned} (F \circ G)(x \circ_S y) &= G(F(x \circ_S y)) \\ &= G(F(x) \circ_T F(y)) \\ &= G(F(x)) \circ_U G(F(y)) \\ &= (F \circ G)(x) \circ_U (F \circ G)(y). \end{aligned} \quad (34)$$

**Solution of Exercise 16.** We can show that we cannot find an inverse morphism

$$\text{ASCII}^{-1} : \text{List}\{0, 1\} \rightarrow \text{List char}. \quad (35)$$

At first sight everything seems in order: if we can find an isomorphism to  $\text{List}[0, 127]$ , and we can express integers in binary, what could hold us back?

What fails here is something so simple it could go unnoticed: the hypothetical function  $g$  is not well-defined for all points of its domain. We know how to translate a binary string of length 7, 14, 21, ... back to symbols; but what would be the output of  $g$  on the string 111?

The function  $g$  is a left inverse for ASCII, in the sense that  $\text{ASCII} \circ g = \text{id}_{\text{List char}}$ , but it is not a right inverse.

**Solution of Exercise 17.** The answer is **no** because the encoding is context dependent; I don't know if a single letter is followed by a space or another letter. For example, take the string

$$\text{I AM MAX}. \quad (36)$$

We can decompose it as follows

$$\text{I A M } \circ \text{ } \circ \text{ M } \circ \text{ AX}. \quad (37)$$

If Morse encoding was a morphism  $F$  then we would be able to encode the string as follows:

$$\text{morse}(\text{I A}) \circ \text{morse}(\text{M}) \circ \text{morse}(\text{ }) \circ \text{morse}(\text{M}) \circ \text{morse}(\text{AX}). \quad (38)$$

However, this cannot work, because in the second instance of  $M$  we would need to output a letter separator, while in the first case we don't.

Can you find a way to fix it?

For example, consider the alphabet obtained by taking the *product* of the letters and numbers with the set of spaces  $\{\text{█}, \text{█}\}$ :

$$((A \text{ to } Z) \cup (0 \text{ to } 9)) \times \{\text{█}, \text{█}\}, \quad (39)$$

where we annotate if each symbol is followed by a letter or by a space.

In this representation, the string can be written as

$$\langle I, \text{█} \rangle \langle A, \text{█} \rangle \langle M, \text{█} \rangle \langle M, \text{█} \rangle \langle A, \text{█} \rangle \langle X, \text{█} \rangle. \quad (40)$$

Based on this representation we can define context-independent rules that make a morphism.

**Solution of Exercise 18.** We start with the first one. Consider  $x \in \mathbf{G}$ . We know that

$$F(\text{id}_{\mathbf{G}} \circ_{\mathbf{G}} x) = F(x). \quad (41)$$

On the other hand, we know:

$$F(\text{id}_{\mathbf{G}} \circ_{\mathbf{G}} x) = F(\text{id}_{\mathbf{G}}) \circ_{\mathbf{H}} F(x). \quad (42)$$

These two are equivalent if and only if  $F(\text{id}_{\mathbf{G}}) = \text{id}_{\mathbf{H}}$ .

For the second statement, consider again  $x \in \mathbf{G}$ . We now that

$$\begin{aligned} F(\text{inv}_{\mathbf{G}}(x) \circ_{\mathbf{G}} x) &= F(\text{id}_{\mathbf{G}}) \\ &= \text{id}_{\mathbf{H}}, \end{aligned} \quad (43)$$

and

$$F(\text{inv}_{\mathbf{G}}(x) \circ_{\mathbf{G}} x) = F(\text{inv}_{\mathbf{G}}(x)) \circ_{\mathbf{H}} F(x). \quad (44)$$

These two are equivalent if and only if  $F(\text{inv}_{\mathbf{G}}(x)) = \text{inv}_{\mathbf{H}}(F(x))$ .

**Solution of Exercise 19.** We have:

$$\begin{aligned} F(m + n) &= i^{m+n} \\ &= i^m \cdot i^n \\ &= F(m) \cdot F(n). \end{aligned} \quad (45)$$

# PART D.ORDER



---

9. Posets	153
10. Constructing posets	175
11. Monotonicity	189

---







## 9. Posets

Life is about trade-offs: there is seldom a uniformly best outcome; rather we need to reason with incomparable attributes. Partially ordered sets (posets) are the mathematical structure used to reason about trade-offs. They are also important as one of the simplest examples of categories.

9.1 Trade-offs . . . . .	154
9.2 Ordered sets . . . . .	158
9.3 Counting orders . . . . .	163
9.4 Code exercises - Posets representation . . . . .	164
9.5 Power poset . . . . .	166
9.6 Code exercises - Power poset . . . . .	167
9.7 Chains and Antichains . . . . .	168
9.8 Code exercises - Chains and antichains . . . . .	170
9.9 Measuring posets . . . . .	171
9.10 Code exercises - Measuring posets	172
9.11 More general preferences . . . . .	173

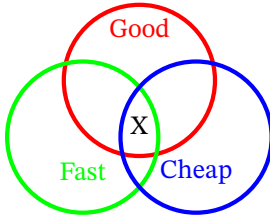


Figure 1.

## 9.1. Trade-offs

Trade-offs characterize all engineering disciplines.

Do you want to build something?

- ▷ If you want it to be done well and quickly, it won't be cheap.
- ▷ If you want it to be done well and cheaply, it won't be quick.
- ▷ If you want it cheaply and quickly, it won't be done well.

To characterize engineering trade-offs, we will use the mathematical structure of partial orders. In the next section, we will explore some examples, to better contextualize trade-offs.

### Functionality and resources

In this section, we introduce concepts which will be important throughout the book, when talking about theories of design. We distinguish semantically between **functionalities** and **requirements/costs**. In general, you prefer **functionalities** to be “large” (Fig. 2b) and **requirements/costs** to be “small” (Fig. 2a).



Figure 2.

We think of three achievable accuracy plots (Fig. 3).

- ▷ In Fig. 3a we plot trade-offs in costs and add a “feasibility” curve. Everything above this curve is feasible and will cost more than what is *on* the curve.
- ▷ In Fig. 3b we plot trade-offs in functionalities and add a “feasibility” curve. Everything below the curve is feasible, but is below the “standards” required by the curve.
- ▷ In Fig. 3c we plot functionality and resource together, representing the trade-offs between “how good a product is” and “how much one needs to pay for it”. Feasible pairs are represented via the feasibility curve. Everything above the curve will be feasible (by paying more).

It is a good exercise to open any engineering book, find the graphs talking about “achievable” performance and “resources” needed, and classify into one of the ones reported in Fig. 3.

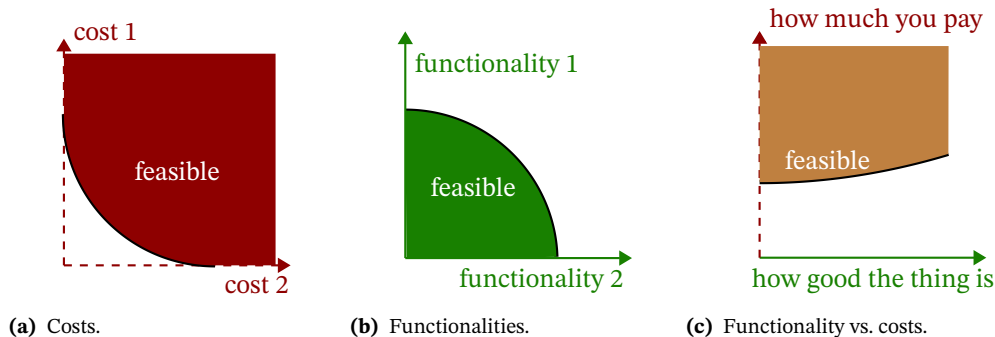


Figure 3.

## Trade-offs for the human body

The human body is a great example of trade-offs and adaptability. Consider sports: when looking at different disciplines, various physical abilities are desired and trade-offs between them characterize athletes.

For instance, we can think about trade-offs between **speed** and **strength** for humans (Fig. 4). These are functionalities, which different athletes might want to maximize. Consider Usain Bolt, who owns the 100 meters, 200 meters, and 4×100 meters relay world records. Without doubts, in the human speed-strength trade-off curve he positions himself close to the highest achievable speeds. At the same time, however, Usain Bolt is not among the strongest men in the world. To see the other end of the curve, we need to introduce Oleksii Novikov, who won the 2020 World's Strongest Man competition. Similarly to Bolt, he is among the best in his discipline, reaching very high strength. Again, the speed-strength trade-off implies that Oleksii cannot be among the fastest men in the world, if he wants to be among the strongest ones.

In this case, the resource needed to obtain speed or strength is the amount of **training** (Fig. 5). If we want to relate the invested training and the resulting strength reached by the athletes, we will notice that with a lot of training, Novikov will improve his results, approaching perfection. On the other hand, the kind of training Bolt undergoes is not optimizing strength, and therefore his results will be less effective towards maximizing strength.

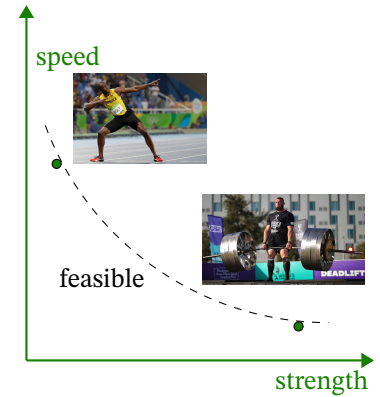


Figure 4.

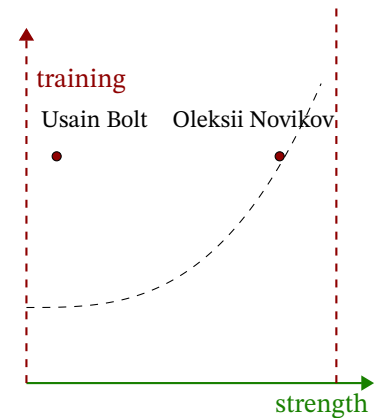


Figure 5.



Masks

Orders give us a rich way to describe products under various lenses. Recently, we all needed to become experts of protective masks. In this section, we will show various ways in which we can order the latter by functionality.

By first thinking about the effectiveness of the mask in protecting the wearer from a virus, we can order masks as in Fig. 6. In general masks are classified following their filter abilities and inward leakages. The FFP1 class filters at least 80 % of airborne particles and allows less than 22 % inward leakage. The FFP2 class filters at least 96 % of airborne particles and allows less than 8 % inward leakage, and the FFP3 class filters at least 99 % of airborne particles and allows less than 2 % inward leakage.

Obviously, based on the protection level, the most performant in Fig. 6 is FFP3, and the worst is the fashion one. However, this is not the only way in which we can classify masks. If, for instance, we want to consider a functionality “how much does the mask say about the wearer”, we can order the masks differently. Arguably, the ordering could look like the one in Fig. 7a.

Indeed, choosing a fashion mask might say that the wearer cares more about aesthetics than safety, and choosing a FFP3 highlights responsible behaviors, care, and research in masks models.

Similarly, we could order masks based on different performance criteria, adding the functionality “how much does it protect others?” (Fig. 7b).

On the other hand, we could think about the trade-offs between the mask performance and its cost, presenting a functionality-resource plot (Fig. 7c).

More performant masks are typically more expensive, and the fashion mask will be probably the least performance and most expensive.

This example once again highlights the flexibility and richness of the “orders approach”. This will be much more evident in the next example.

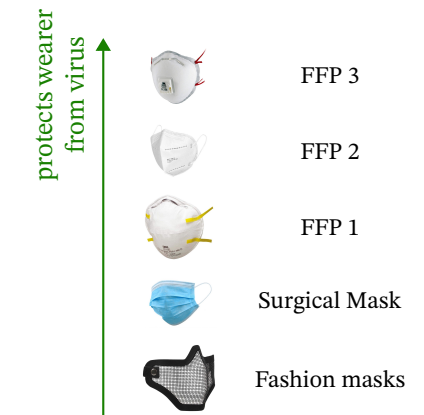


Figure 6.: Ordering masks by protection levels

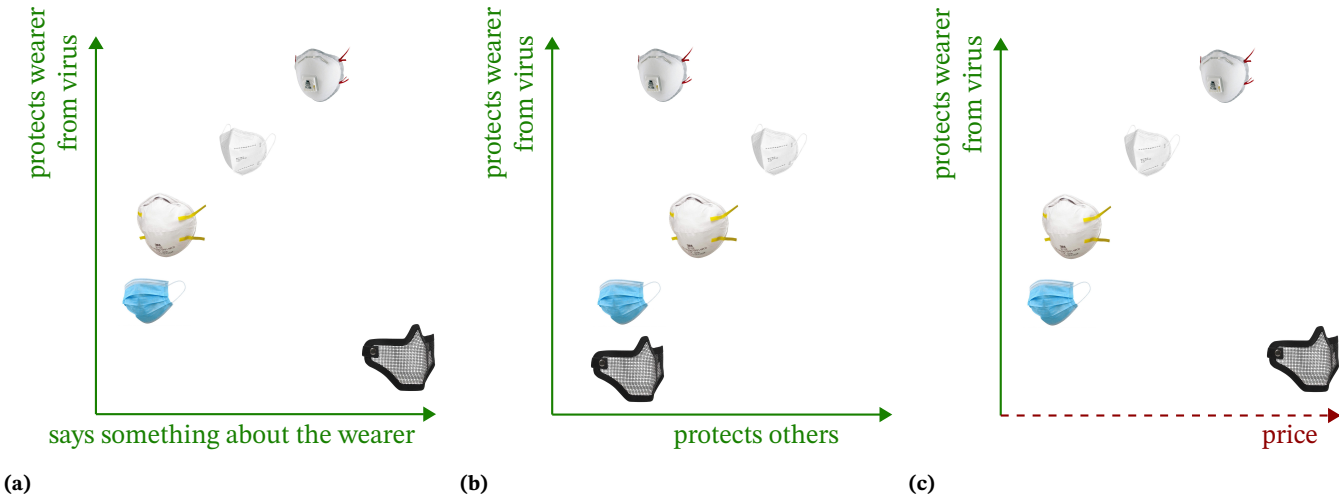


Figure 7.: Ordering masks by other considerations

## Hats and headphones

Another good example of ordering of multiple functionalities and costs is the one of headphones. Consider a set of headphones and order them based on their abilities to “keep warm” and to “reproduce music” (Fig. 8a).

Clearly, these two functionalities represent different objectives and diverse product ranges will satisfy them in different ways. For instance, winter hats clearly cannot reproduce music, but keep very warm. On the other hand, large headphones are the best in reproducing music, but cannot keep as warm as winter hats. Functionalities come at a cost. For instance, we could plot the trade-off between “keep warm” and price (Fig. 8b). Other interesting costs could be expressed via the frequency of charging (Fig. 8d) or the hassle of dealing with wires (Fig. 8c).

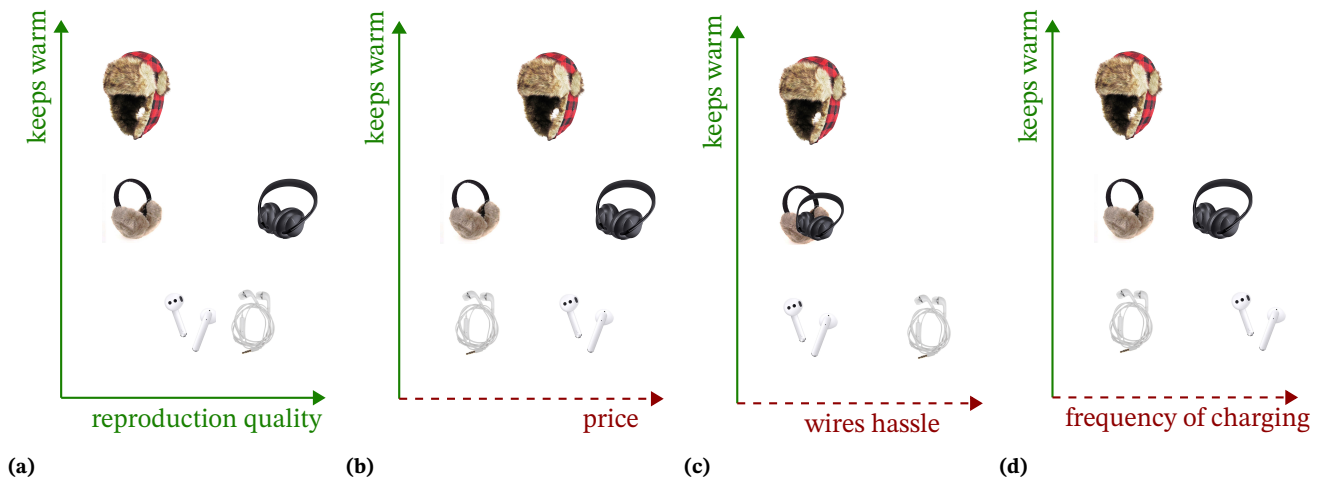


Figure 8.: Ordering hats and headphones

## The law of successful products

By considering all the aforementioned characteristics together

$$(\text{keeps warm} \times \text{reproduction quality}) \times (\text{price} \times \text{frequency of charging} \times \text{wires hassle}), \quad (1)$$

no product dominates another.

This is the *law of successful products*. At equilibrium, in an efficient and free market, no product completely dominates another by both functionality and costs. Otherwise, the dominated product would not sell. Once we *specify the design purpose* and the related constraints, we can (partially) order products.

Watch: *Pre-orders, partial orders, and total orders* (2 minutes).

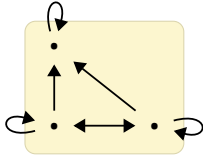
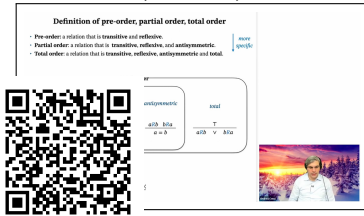


Figure 9.: A pre-order represented as a graph.

## 9.2. Ordered sets

So far, the discussion has been purely qualitative. In this section, we introduce pre-orders, partial orders, and total orders. Davey and Priestley [4] and Roman [27] are possible reference texts.

### Pre-order

A pre-order is a set together with a relation that is both reflexive (Def. 5.19) and transitive (Def. 5.24).

#### Definition 9.1 (Pre-ordered set)

A *pre-ordered set* is a tuple  $P = \langle P, \leq_P \rangle$ , where  $P$  is a set, called the *carrier set* or *underlying set*, together with a relation  $\leq_P$  that is reflexive and transitive.

An example of a pre-ordered set represented as a graph is shown in Fig. 9. In the graph representation of a pre-order  $P$ , we draw an arrow between  $x$  and  $y$  if  $x \leq_P y$ .

**Example 9.2.** The reachability relationship in any directed graph (potentially including cycles) is a pre-order. The pre-order  $P$  is defined as follows. The set  $P$  is the set of nodes of the graph. Take any two nodes  $x, y \in P$ . One has  $x \leq_P y$  if and only if there is a path from  $x$  to  $y$  in the directed graph. There is always a path from a node to itself (reflexivity), and given a path from  $x$  to  $y$ , and one from  $y$  to  $z$ , we know that there is a path from  $x$  to  $z$  (transitivity).

**Exercise 20.** Consider the set  $P = \{x, y, z\}$ . Which of the following are pre-orders? Why?

1.  $P = \{\langle x, x \rangle, \langle x, y \rangle, \langle y, x \rangle, \langle y, y \rangle\}$ .
2.  $Q = \{\langle x, y \rangle, \langle y, z \rangle, \langle z, x \rangle\}$ .

See solution on page 199.

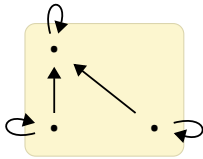


Figure 10.: A partial order represented as a graph.

## Partial order

By adding the condition of *antisymmetry* (Def. 5.18) to a pre-order, we obtain a partially-ordered set.

### Definition 9.3 (Partially ordered set)

A pre-ordered set  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  is a *partially-ordered set* (poset) if the relation  $\leq_{\mathbf{P}}$  is antisymmetric.

An example of a partially ordered set represented as a graph is shown in Fig. 10. By comparing this with Fig. 9, we notice that the double-headed arrow is not allowed anymore (indeed, its existence would imply that source and target of the arrow are the same element in the poset).

**Exercise 21.** Does the reachability relationship in any directed graph define a poset? Why? If not, can you modify the initial statement to make it work?

*See solution on page 199.*

**Example 9.4.** The following defines a partial order  $\leq$  on the set of natural numbers  $\mathbb{N}$ . Define, for all  $x, y \in \mathbb{N}$ ,

$$x \leq y \quad \text{if, and only if} \quad x \text{ divides } y. \quad (2)$$

By definition, a natural number  $x$  divides another natural number  $y$  if there exists some other natural number  $z$  such that  $xz = y$ . The notation for “ $x$  divides  $y$ ” is  $x|y$ .

### Graded exercise D.1

Consider the set  $\mathbf{A}$  of natural numbers which divide the number 60, and equipped with the partial order defined by

$$x \leq y \quad \text{if, and only if} \quad x \text{ divides } y \quad (3)$$

for all  $x, y \in \mathbf{A}$ . Draw the Hasse diagram of this partially ordered set.

### Graded exercise D.2 (PolynomialDivisibility)

Let  $\mathbf{A}$  be the set of all polynomials with coefficients in  $\mathbb{R}$ . Recall that a polynomial  $p$  divides a polynomial  $q$  if there exists a polynomial  $m$  such that  $p \cdot m = q$ . If  $p$  divides  $q$  we denote this by  $p|q$ . Divisibility defines an endorelation on  $\mathbf{A}$  by saying  $p$  is related to  $q$  iff  $p|q$ . Does this define a pre-order structure on  $\mathbf{A}$ ? Does this define a poset structure on  $\mathbf{A}$ ? Justify your answer.

## Total order

By imposing *totality* (Def. 5.20), we obtain a total order.

### Definition 9.5 (Totally ordered set)

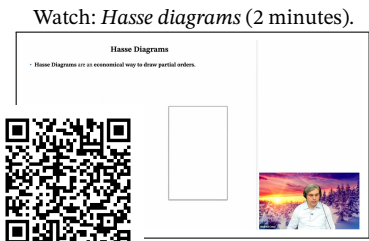
A partially ordered set  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  is a *totally ordered set* if the relation  $\leq_{\mathbf{P}}$  is total.

An example of a totally ordered set represented as a graph is reported in Fig. 11.

**Example 9.6 (Reals).** The real numbers  $\mathbb{R}$  form a totally ordered poset  $\langle \mathbb{R}, \leq \rangle$  with order relation given by the usual ordering.



Figure 11.: A total order.



### Hasse diagrams

We can represent partial orders in various ways. We now take a proxy partially ordered set and represent it using different conventions. Consider  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$ , where  $\mathbf{P} = \{x, y, z\}$  and  $x \leq_{\mathbf{P}} y, y \leq_{\mathbf{P}} z$ . First, we could represent this using the same visualization we had for relations (Fig. 12a).

However, this is quite heavy, and does not exploit the fact that partial orders are endorelations. Therefore, we could think to only draw the carrier set once, and to drop the order relations arising from reflexivity (Fig. 12b).

However, the arrow from  $x$  to  $y$  is implicit in partial orders, because of transitivity.

A *Hasse diagram* is an economical (in terms of arrows) way to visualize a poset. In a Hasse diagram elements are points, and if  $p \leq_{\mathbf{P}} q$  then  $p$  is drawn lower than  $q$  and with an edge connected to it, if no other point is in between (Fig. 12c). Hasse diagrams are directed graphs.

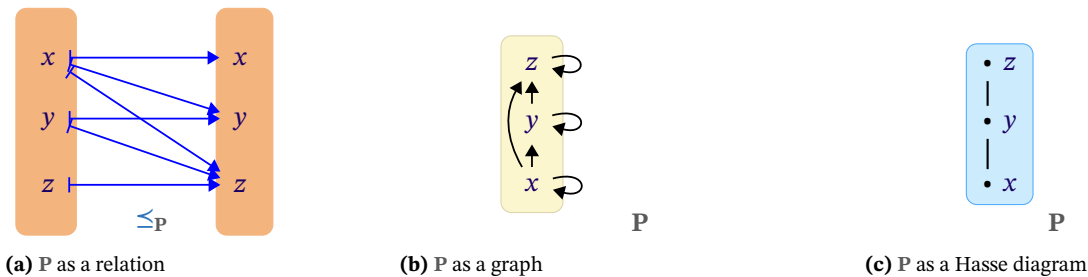


Figure 12.: Three different representations for a poset

**Example 9.7** (Discrete partially ordered sets). Every set  $\mathbf{P}$  can be considered as a *discrete poset*  $\mathbf{P} = \langle \mathbf{P}, = \rangle$  using equality as the partial order. (Notice that equality is symmetric, transitive, and antisymmetric.) When visualized as a Hasse diagram, discrete posets are a collection of points (Fig. 13).

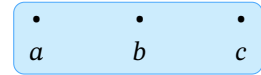


Figure 13.

**Definition 9.8** (Boolean poset **Bool**)

The set of booleans  $\mathbf{Bool} = \{\perp, \top\}$  can be made into a poset by choosing the order  $\perp \leq_{\mathbf{Bool}} \top$ . This is equivalent to using “ $\Rightarrow$ ” as a relation. We obtain the poset

$$\mathbf{Bool} := \langle \mathbf{Bool}, \Rightarrow \rangle. \quad (4)$$

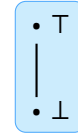


Figure 14.

**Example 9.9** (Qualitative information). In the example of the battery choice, both mass and money can be thought of as partially ordered sets. Imagine that you have batteries which are “cheap”, “midrange”, and “expensive”. Clearly, if the partially ordered set represents cost, we can say that cheap  $\leq$  midrange  $\leq$  expensive. While this is a quantitative judgement (indeed, if I care about cost, I will prefer a cheap battery over a midrange one), it is not a numeric one (cheap could represent a number, but also a range of numbers or just a price category). This can be represented as in Fig. 15.

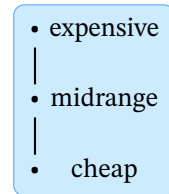


Figure 15.: The cost of a battery can be represented as a poset.

**Example 9.10.** Consider a poset  $\mathbf{P}$  representing a person’s food preference over the set  $\mathbf{P} = \{\text{pretzel}, \text{chili}, \text{burger}, \text{apple}\}$  with  $\text{pretzel} \leq_{\mathbf{P}} \text{chili}$ ,  $\text{burger} \leq_{\mathbf{P}} \text{chili}$ ,  $\text{chili} \leq_{\mathbf{P}} \text{apple}$ , and  $\text{burger} \leq_{\mathbf{P}} \text{apple}$ . This can be represented with a Hasse diagram as in Fig. 16.

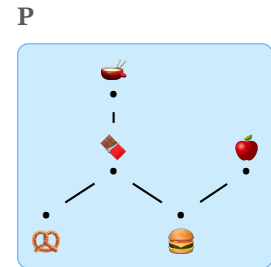
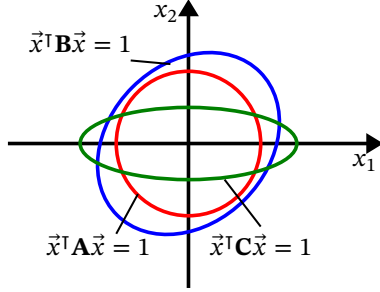
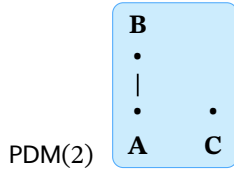


Figure 16.: Example of Hasse diagram of  $\mathbf{P}$ .



(a) Example of ellipses representing positive definite matrices.



(b) Example of order between positive semi-definite matrices.

Figure 17.

### Example: positive definite matrices as ellipsoids

#### Definition 9.11 (Positive definite matrix)

A symmetric matrix  $\mathbf{M} \in \mathbb{R}^{n \times n}$  is *positive definite* if  $\vec{x}^T \mathbf{M} \vec{x} > 0$  for all non-zero  $\vec{x} \in \mathbb{R}^n$ . We call the set of all such matrices  $\text{PDM}(n)$ .

Positive definite matrices have real, positive eigenvalues, which can be interpreted as axes lengths of ellipsoids. Any matrix  $\mathbf{A} \in \text{PDM}(n)$  describes an ellipsoid, which can be written as a quadratic equation:

$$\vec{x}^T \mathbf{A} \vec{x} = 1, \quad \vec{x} \in \mathbb{R}^n. \quad (5)$$

We can define a partial order on  $n$  as

$$\frac{\mathbf{A} \leq_{\text{PDM}(n)} \mathbf{B}}{\vec{x}^T \mathbf{A} \vec{x} \leq \vec{x}^T \mathbf{B} \vec{x} \quad \forall \vec{x} \in \mathbb{R}^n}. \quad (6)$$

The order can be interpreted as an inclusion of ellipsoids. Take for instance the matrices

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 3/4 & -1/8 \\ -1/8 & 3/4 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix}. \quad (7)$$

The order  $\mathbf{P}$  on the set  $\{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$  is reported in Fig. 17b, and it is easily explained via Fig. 17a. The ellipse representing  $\mathbf{A}$  (in red) is included by the one representing matrix  $\mathbf{B}$  (in blue), but not by the one representing matrix  $\mathbf{C}$  (in green). Furthermore, the one representing  $\mathbf{B}$  includes the one representing  $\mathbf{C}$ .

## 9.3. Counting orders

Let's count the number of posets.

If there is only one element, there is only one way to order it (Fig. 18a).

With a 2-elements set, there are 2 posets (panel b), “up to isomorphism”, that is, if we do not care about the labels of points.

On 3-elements sets, we have 5 posets (panel c).

On 4-elements sets, we have 16 posets (panel d).

Watch: *Counting orders* (7 minutes).

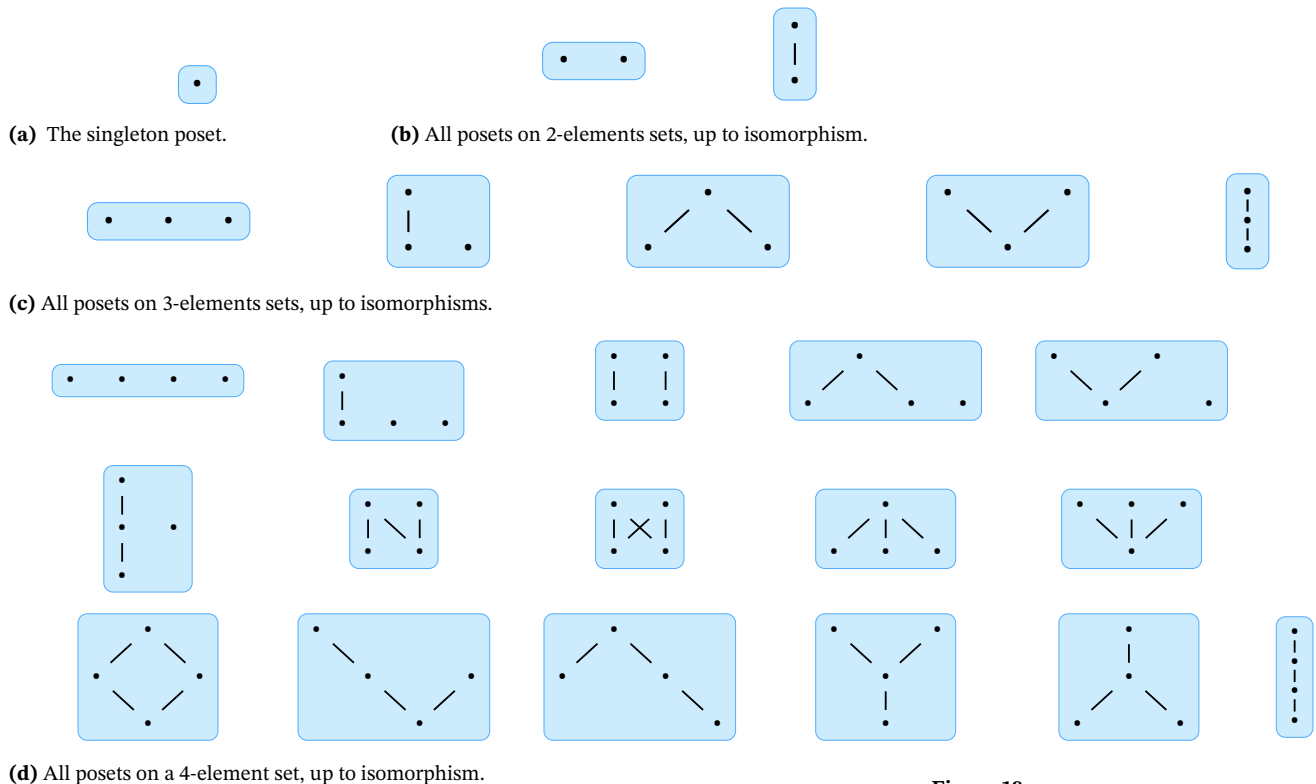


Figure 18.



### 9.4. ○ Posets representation

Listing 55 shows the interface for posets. A poset is an endorelation: a relation with same source and target. (Unfortunately, using the Python type system we cannot express the fact that `holds()` must be a partial order.)

Listing 56 shows the interface for finite posets: a `FinitePoset` is a `Poset` whose underlying carrier set is finite.

Listing 55: The `Poset` interface.

```
class Poset(Generic[E], Relation[E, E], ABC):
    def source(self) -> Setoid[E]:
        return self.carrier()

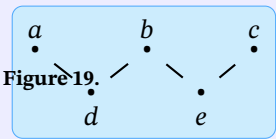
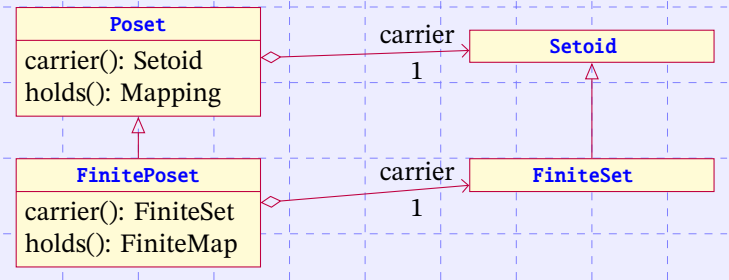
    def target(self) -> Setoid[E]:
        return self.carrier()

    @abstractmethod
    def carrier(self) -> Setoid[E]: ...

    @abstractmethod
    def holds(self, a: E, b: E) -> bool: ...
```

Listing 56: The `FinitePoset` interface.

```
class FinitePoset(Generic[E], Poset[E], ABC):
    @abstractmethod
    def carrier(self) -> FiniteSet[E]: ...
```



```
carrier:
  elements: [a, b, c, d, e]
hasse:
- [d, a]      # comment
- [d, b]
- [e, b]
- [e, c]
```

Figure 20.

```
carrier:
  elements: []
hasse: []
```

Figure 21.: An empty poset.

### Concrete representation

The representation of a finite poset (Fig. 20) is described compactly using the Hasse representation. Compared to the `FiniteRelation` representation, in which we describe all relations among elements, here we do not need to specify the redundant information that can be recovered using the properties of antisymmetry, reflexivity, and transitivity.

#### Code exercise D.1 (TestFinitePosetRepresentation)

Create a function to load and save the data.

Implement the interface in Listing 57.

Test using:

```
act4e-check TestFinitePosetRepresentation
```

Listing 57: The `FinitePosetRepresentation` interface.

```
class FinitePosetRepresentation(Serializer[FinitePoset[Any],  
    ↪ FinitePoset_desc], ABC):  
    pass
```

## 9.5. Power poset

We have introduced the concept of power set in Section 4.3. There is a natural order on subsets, given by set inclusion. We can thus define the *power poset*.

**Definition 9.12** (Power poset)

Given a set  $A$ , define the *power poset*  $\text{Pow } A = \langle \text{Pow } A, \subseteq \rangle$  by ordering the subsets in its power set  $\text{Pow } A$  by inclusion.

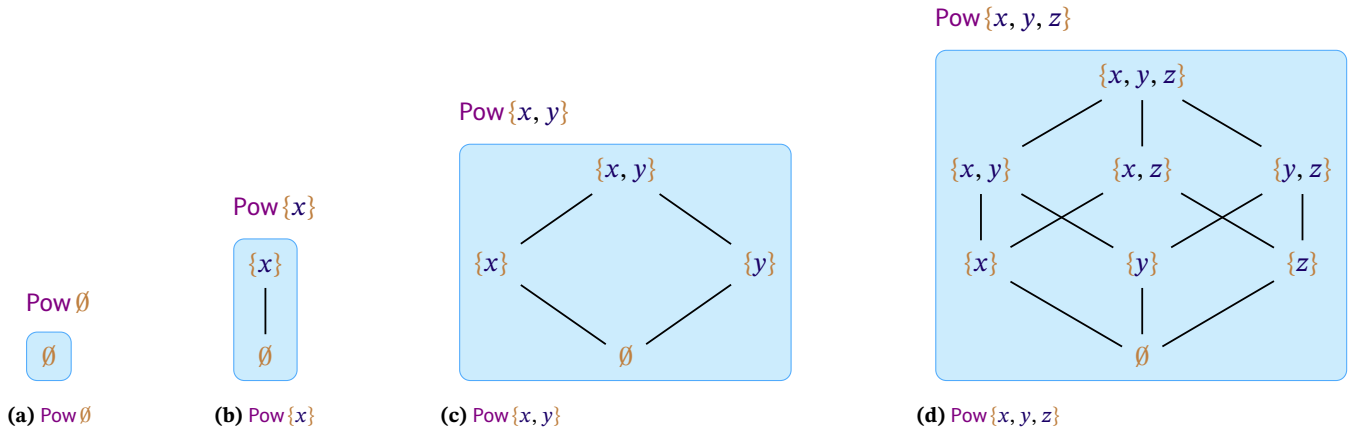
A subset  $S$  precedes  $T$  if  $S \subseteq T$ :

$$\frac{S \leq_{\text{Pow } A} T}{S \subseteq T}. \quad (8)$$

This is illustrated in Fig. 22 for sets of 0, 1, 2, 3 elements.

**Exercise 22.** Check formally that  $\leq_{\text{Pow } A}$  defined in (8) is a partial order.

*See solution on page 199.*



**Figure 22.:** Power set as a poset.

## 9.6. ☉ Power poset

### Code exercise D.2 (TestFinitePosetConstructionPower)

Write code to compute the power poset of a finite set.

Implement the interface in Listing 58.

Test using:

```
act4e-check TestFinitePosetConstructionPower
```

**Listing 58:** The `FinitePosetConstructionPower` interface.

```
class FinitePosetConstructionPower(ABC):
    @abstractmethod
    def powerposet(self, s: FiniteSet[C]) -> FinitePosetOfFiniteSubsets[C,
        Any]: ...
```

A `PosetOfFiniteSubsets` is a poset whose carrier is a `SetOfFiniteSubsets`. We have already encountered this class in the exercise Section 4.7 (Listing 17), which asked to create a power set.

In this exercise you should reuse the code to create the carrier set. What is new is that you must write the code that defines the order on the subsets, which should be given by set inclusion.

You have already written the code for checking for subset inclusion in the exercise Section 4.6.

**Listing 59:** The `PosetOfFiniteSubsets` interface.

```
class PosetOfFiniteSubsets(Generic[C, E], Poset[E], ABC):
    """A poset of subsets."""

    @abstractmethod
    def carrier(self) -> SetOfFiniteSubsets[C, E]: ...
```

**Listing 60:** The `FinitePosetOfFiniteSubsets` interface.

```
class FinitePosetOfFiniteSubsets(Generic[C, E], PosetOfFiniteSubsets[C, E],
    FinitePoset[E], ABC):
    @abstractmethod
    def carrier(self) -> FiniteSetOfFiniteSubsets[C, E]: ...
```

## 9.7. Chains and Antichains

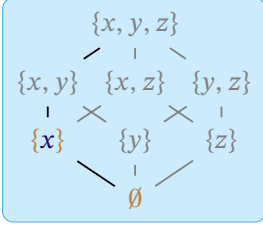
There are two special types of subsets of a poset: chains and antichains. Their definitions are dual.

### Definition 9.13 (Chain in a poset)

Given a poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$ , a *chain* is a subset  $\mathbf{S} \subseteq \mathbf{P}$  such that any two elements of  $\mathbf{S}$  are comparable:

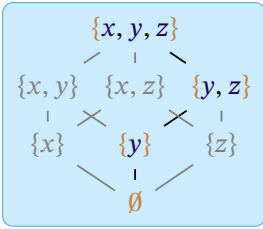
$$\frac{x, y \in \mathbf{S}}{(x \leq_{\mathbf{P}} y) \vee (y \leq_{\mathbf{P}} x)}. \quad (9)$$

$\text{Pow}\{x, y, z\}$



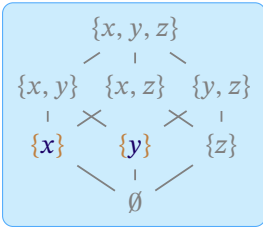
(a) A chain.

$\text{Pow}\{x, y, z\}$



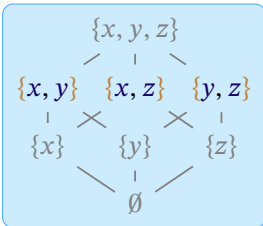
(b) A chain.

$\text{Pow}\{x, y, z\}$



(c) An antichain.

$\text{Pow}\{x, y, z\}$



(d) An antichain.

**Figure 23.:** Examples of chains (a-b) and antichains (c-d) in the poset  $\text{Pow}\{x, y, z\}$ .

### Definition 9.14 (Antichain in a poset)

An *antichain* is a subset  $\mathbf{S}$  of a poset where no two distinct elements are comparable:

$$\frac{x, y \in \mathbf{S} \quad x \not\leq_{\mathbf{P}} y}{x = y}. \quad (10)$$

**Remark 9.15.** Note that the empty set  $\emptyset$  is both a chain and an antichain.

We denote the set of antichains of a poset  $\mathbf{P}$  by  $\text{Anti } \mathbf{P}$ .

**Example 9.16** (Chains and antichains in a power poset). Consider the poset in Fig. 22d. Examples of chains are

$$\{\emptyset, \{x\}\} \quad \text{and} \quad \{\emptyset, \{y\}, \{y, z\}, \{x, y, z\}\}, \quad (11)$$

depicted in Fig. 23a and Fig. 23b, respectively.

Examples of antichains are

$$\{\{x\}, \{y\}\} \quad \text{and} \quad \{\{x, y\}, \{x, z\}, \{y, z\}\}, \quad (12)$$

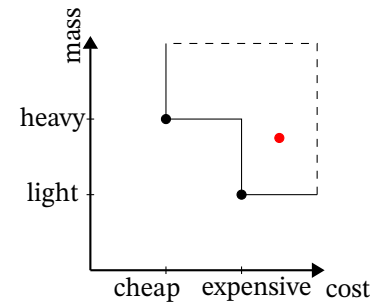
depicted in Fig. 23c and Fig. 23d, respectively.

**Example 9.17.** In the context of battery choices, consider the diagram in Fig. 24. The black markers represent an antichain of choices

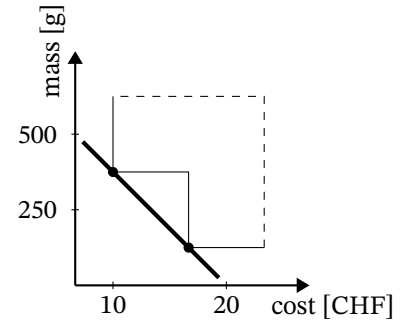
$$\{\langle \text{cheap}, \text{heavy} \rangle, \langle \text{expensive}, \text{light} \rangle\}. \quad (13)$$

It is a set of pairs because they do not dominate each other: one is cheaper, but is heavier, and the other is more expensive, but lighter, making them incomparable. If a battery with the properties as the red marker existed (very expensive, between light and heavy), that would be an element that cannot be part of the antichain, since it would be dominated by  $\langle \text{expensive}, \text{light} \rangle$ .

Similarly, we could think of a continuous law which relates battery cost and mass. Assume that cheap means 10 CHF, expensive means 20 CHF, light means 250 g, and heavy means 500 g. For instance, consider the antichain given by  $\text{mass} = 500 - 25 \cdot \text{cost}$ , with maximum possible cost 20 CHF (Fig. 25).

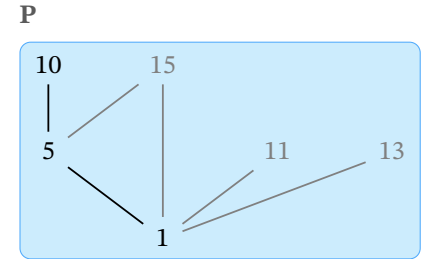


**Figure 24.:** Example of discrete antichains.

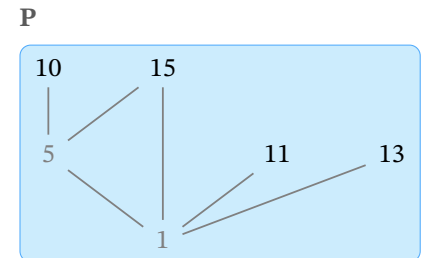


**Figure 25.:** Example of continuous antichains.

**Example 9.18.** Consider the poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  where  $(p \leq_{\mathbf{P}} q)$  if  $p$  is a divisor of  $q$  and  $\mathbf{P} = \{1, 5, 10, 11, 13, 15\}$ . A chain of  $\mathbf{P}$  is  $\{1, 5, 10\}$  (Fig. 26). An antichain of  $\mathbf{P}$  is  $\{10, 11, 13, 15\}$  (Fig. 27).



**Figure 26.**



**Figure 27.**

## 9.8. Chains and antichains

### Code exercise D.3 (TestFinitePosetSubsetProperties)

Check if a subset is a chain or an antichain.

Implement the interface in Listing 61.

Test using:

```
act4e-check TestFinitePosetSubsetProperties
```

**Listing 61:** The `FinitePosetSubsetProperties` interface.

```
class FinitePosetSubsetProperties(ABC):
    @abstractmethod
    def is_chain(self, fp: FinitePoset[E], s: Collection[E]) -> bool:
        """True if the given elements form a chain.
        Note: s can be any collection, not necessarily ordered.
        """

    @abstractmethod
    def is_antichain(self, fp: FinitePoset[E], s: Collection[E]) -> bool:
        """True if the given elements form an antichain."""
```

*Hint:* To check if a subset is an antichain, it is sufficient to check that no pair of different elements are comparable. This requires  $n(n - 1)$  comparisons, and you cannot do better in general, without any additional assumption on the poset.

*Hint:* To prove that a subset is a chain, you must check that all elements are comparable. It is not necessary to produce an order. *Hint:* On the other hand, it might be more efficient to try to find the chain order; it could save a lot of computation. An efficient way to do this is using a sorting algorithm (which can possibly fail, if some elements are not comparable.)

## 9.9. Measuring posets

We can define two measurements for a poset: the height and the width. These measurements allow to quantify the performance of several algorithms we will see in the latter parts of the book.

**Definition 9.19** (Width of a poset)

The *width of a poset*, denoted  $\text{width}(\mathbf{P})$ , is the maximum cardinality of an antichain in  $\mathbf{P}$ .

**Definition 9.20** (Height of a poset)

The *height of a poset*, denoted  $\text{height}(\mathbf{P})$ , is the maximum cardinality of a chain in  $\mathbf{P}$ .

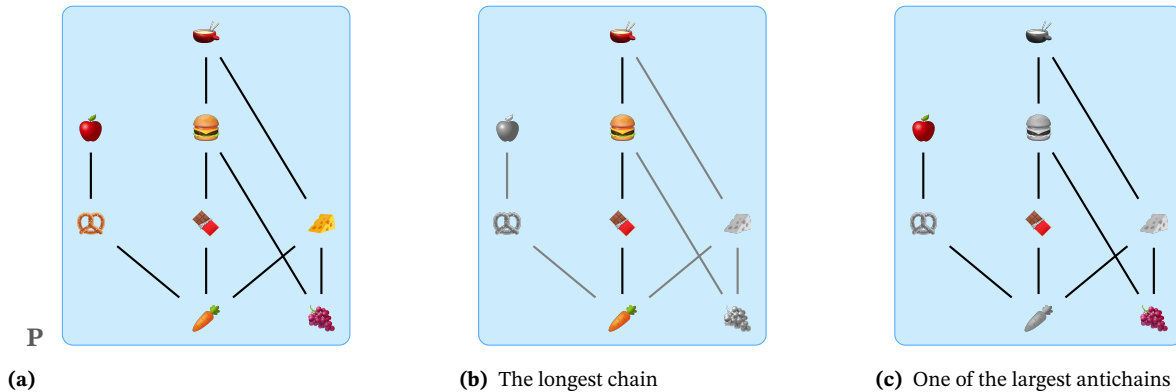
Note that an empty poset has exactly one chain and one antichain: the empty set. Therefore, the height and width are zero.

**Example 9.21.** Consider the poset  $\mathbf{P}$  in Fig. 28a. The longest antichains of  $\mathbf{P}$  are  $\{\text{apple}, \text{burger}, \text{cheese}\}$ ,  $\{\text{apple}, \text{diamond}, \text{cheese}\}$ ,  $\{\text{pretzel}, \text{diamond}, \text{cheese}\}$ ,  $\{\text{apple}, \text{diamond}, \text{grapes}\}$ ,  $\{\text{pretzel}, \text{diamond}, \text{grapes}\}$ , and  $\{\text{pretzel}, \text{burger}, \text{cheese}\}$ . Therefore,

$$\text{width}(\mathbf{P}) = 3. \quad (14)$$

The longest chain in the poset is given by  $\{\text{carrot}, \text{diamond}, \text{burger}, \text{hotdog}\}$ , and therefore

$$\text{height}(\mathbf{P}) = 4. \quad (15)$$



**Figure 28.:** Example for height and width of a poset.

**Graded exercise D.3** (MeasurePowerPoset)

Let  $\mathbf{A}$  be a finite set with  $n$  elements. Obtain an expression (without proof) for

1.  $\text{width}(\text{Pow } \mathbf{A})$ ;
2.  $\text{height}(\text{Pow } \mathbf{A})$ .



## 9.10. 🕒 Measuring posets

### Code exercise D.4 (TestFinitePosetMeasurement)

Compute the width and height of a poset.

Implement the interface in Listing 62.

Test using:

```
act4e-check TestFinitePosetMeasurement
```

**Listing 62:** The `FinitePosetMeasurement` interface.

```
class FinitePosetMeasurement(ABC):
    @abstractmethod
    def width(self, fp: FinitePoset[Any]) -> int:
        """Return the width of the poset."""

    @abstractmethod
    def height(self, fp: FinitePoset[Any]) -> int:
        """Return the height of the poset."""
```

*Hint:* For computing the height of a finite (small) poset, it is easiest to just enumerate the chains and check which one is the longest. You can enumerate the chains by looking at the paths from the minimal to the maximal elements.

Note: There is one test case for a large poset. This will probably time out for a naive algorithm. This timeout is not counted as failure for the purpose of the grading.

*Hint:* Computing the *width* of a poset is more complicated.

Note that enumerating all subsets and check which one is an antichain and record the largest we have complexity  $2^n$ , so it is impracticable even for small sets.

We mention here two results that can help.

Dilworth's theorem shows that the *width* of a poset (size of maximum *antichain*) relates to a partition in *chains*, while Mirsky's theorem shows that the *height* of a poset (size of the longest *chain*) relates to a partition in *antichains*.

**Theorem 9.22** (Dilworth's theorem). A finite poset of width  $w$  can be partitioned into  $w$  chains.

**Theorem 9.23** (Mirsky's theorem). A finite poset of height  $h$  can be partitioned into  $h$  antichains.

Given Dilworth's theorem, one approach that works for small posets is to enumerate the chains and then find the smallest subset that covers the poset.

There are many results published that give faster algorithms; see, e.g., [2].

## 9.11. More general preferences

There are more general ways to describe preferences, which might be useful in certain applications.

We give two such examples.

### Definition 9.24 (Quasitransitive relation)

A *quasitransitive relation* is a relation  $\leq_P$  over a set  $\mathbf{P}$  for which

$$\frac{(p \leq_P q) \quad \neg(q \leq_P p) \quad (q \leq_P r) \quad \neg(r \leq_P q)}{(p \leq_P r) \wedge \neg(r \leq_P p)} . \quad (16)$$

Semiorders are particular quasitransitive relations.

### Definition 9.25 (Semiorder)

A *semiorder* is a relation  $\leq_P$  over a set  $\mathbf{P}$  such that:

▷ *Asymmetry* (not antisymmetry) holds:

$$\frac{p \leq_P q}{\neg(q \leq_P p)} . \quad (17)$$

▷ Denote two elements  $q, r \in \mathbf{P}$  which are *incomparable* by  $q \sim r$ . We have:

$$\frac{p \leq_P q \quad q \sim r \quad r \leq_P s}{p \leq_P s} . \quad (18)$$

▷ *Semi-transitivity*:

$$\frac{p \leq_P q \quad q \leq_P r}{(p \leq_P s) \vee (s \leq_P r)} . \quad (19)$$

**Example 9.26** (Interval semi-orders: expressing tolerances). An example of semiorder is the following. Say that you have to express your preference over numbers in  $\mathbf{P} = \{10, 11, 12\}$ . You are indifferent between 10 and 11, and between 11 and 12. However, you prefer 10 to 12. This order satisfies the condition for it to be a semiorder.

More in general, suppose that, when evaluating the cost of a widget, represented as a nonnegative real number, you are indifferent to prices within 1% of the cost. For two real numbers  $x, y \in \mathbb{R}$ , you are indifferent if

$$\frac{x \sim y}{\frac{|x - y|}{(x + y)/2} \leq 0.01} . \quad (20)$$

This is also a semiorder.

Watch: *More general preference structures* (1 minutes).







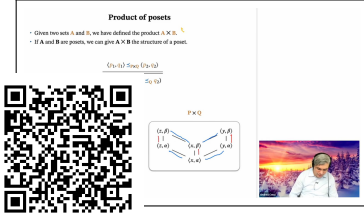
## 10. Constructing posets

In this chapter we look at a few standard recipes how we can construct posets from sets or other posets.

<b>10.1 Product of posets . . . . .</b>	<b>176</b>
<b>10.2 Code exercises - Poset product .</b>	<b>178</b>
<b>10.3 Disjoint union of posets . . . . .</b>	<b>179</b>
<b>10.4 Code exercises - Disjoint union of posets . . . . .</b>	<b>180</b>
<b>10.5 Opposite of a poset . . . . .</b>	<b>181</b>
<b>10.6 Code exercises - Opposite poset .</b>	<b>182</b>
<b>10.7 “Twisted” poset of intervals . . .</b>	<b>183</b>
<b>10.8 Code exercises - Twisted poset construction . . . . .</b>	<b>185</b>
<b>10.9 Arrow poset of intervals . . . . .</b>	<b>186</b>
<b>10.10 Code exercises - Arrow poset construction . . . . .</b>	<b>187</b>



Watch: *Product of posets* (2 minutes).



## 10.1. Product of posets

Just like the product of sets, we can construct the product of posets. That is a poset with the underlying set being the product of the underlying sets.

**Definition 10.1** (Product of posets)

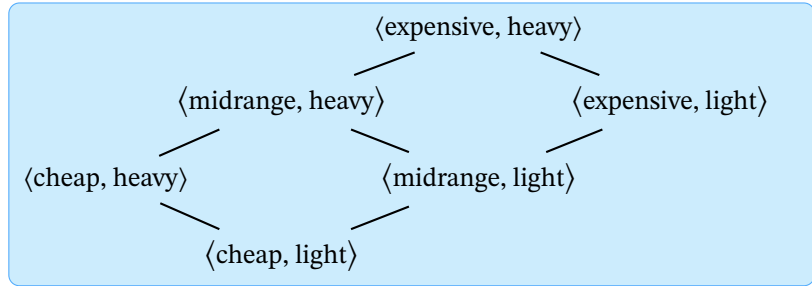
Given posets  $P = \langle P, \leq_P \rangle$  and  $Q = \langle Q, \leq_Q \rangle$ , the *product poset*

$$P \times Q = \langle P \times Q, \leq_{P \times Q} \rangle, \quad (1)$$

is the set  $P \times Q$  equipped with the order  $\leq_{P \times Q}$  given by

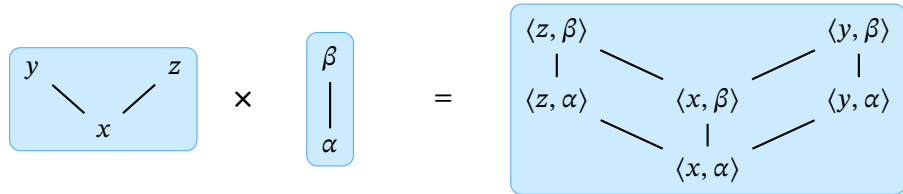
$$\frac{\langle p_1, q_1 \rangle \leq_{P \times Q} \langle p_2, q_2 \rangle}{(p_1 \leq_P p_2) \wedge (q_1 \leq_Q q_2)}. \quad (2)$$

Recalling the battery choice example, we have the two posets representing cost and weight. Given that we want to minimize both cost and weight, by considering the cost poset containing elements “cheap”, “midrange”, and “expensive”, and the weight poset containing elements “light”, and “heavy”, we can represent the product as in Fig. 1.



**Figure 1.:** Product poset of cost and weight for battery choices.

**Example 10.2.** Consider now two posets and their product, given in Fig. 2.



**Figure 2.:** Product of two posets.

## Measuring the product

The following lemma gives expressions for the width and height of the product of two posets.

**Lemma 10.3.** If  $\mathbf{P}, \mathbf{Q}$  are non-empty finite posets, then we know the height of their product:

$$\text{height}(\mathbf{P} \times \mathbf{Q}) = \text{height}(\mathbf{P}) + \text{height}(\mathbf{Q}) - 1 \quad (3)$$

We can derive this bound for the width of the product:

$$\text{width}(\mathbf{P}) \cdot \text{width}(\mathbf{Q}) \leq \text{width}(\mathbf{P} \times \mathbf{Q}) \leq \min \{\text{card}(\mathbf{P}) \cdot \text{width}(\mathbf{Q}), \text{card}(\mathbf{Q}) \cdot \text{width}(\mathbf{P})\}. \quad (4)$$

This bound is tight, in the sense that there exist posets that reach this bound.

*Proof.* The bound (4) can be found in [1]. As for (3), we have the following proof. First, we can construct the longest chain in  $\mathbf{P}$ :

$$\mathbf{A} = \{p_1, \dots, p_{\text{height}(\mathbf{P})}\}. \quad (5)$$

Furthermore, we can construct the longest chain in  $\mathbf{Q}$ :

$$\mathbf{B} = \{q_1, \dots, q_{\text{height}(\mathbf{Q})}\}. \quad (6)$$

Out of them, we can construct the chain

$$\mathbf{C} = \{\langle p_1, q_1 \rangle, \langle p_2, q_1 \rangle, \dots, \langle p_{\text{height}(\mathbf{P})}, q_1 \rangle, \langle p_{\text{height}(\mathbf{P})}, q_2 \rangle, \dots\}, \quad (7)$$

which has height  $\text{height}(\mathbf{P}) + \text{height}(\mathbf{Q}) - 1$ . So we know a lower bound for the height:

$$\text{height}(\mathbf{P} \times \mathbf{Q}) \geq \text{height}(\mathbf{P}) + \text{height}(\mathbf{Q}) - 1. \quad (8)$$

Now, consider a chain  $\{\langle p_1, q_1 \rangle, \dots, \langle p_n, q_n \rangle\}$  in  $\mathbf{P} \times \mathbf{Q}$ . In general, this means that at least a coordinate of  $\langle p_i, q_i \rangle$  must increase in  $\langle p_{i+1}, q_{i+1} \rangle$ . The first coordinate can only increase by  $\text{height}(\mathbf{P}) - 1$  times, and the second one by  $\text{height}(\mathbf{Q}) - 1$  times. Summing up, the total number of elements in the chain is *at most*  $\text{height}(\mathbf{P}) + \text{height}(\mathbf{Q}) - 1$ :

$$\text{height}(\mathbf{P} \times \mathbf{Q}) \leq \text{height}(\mathbf{P}) + \text{height}(\mathbf{Q}) - 1. \quad (9)$$

Because upper and lower bounds are the same, we have an exact expression for the height. Note that this result holds only assuming that  $\mathbf{P}$  and  $\mathbf{Q}$  are not empty (for that case,  $\text{height}(\mathbf{P} \times \mathbf{Q}) = 0$ ).  $\square$

## 10.2. ○ Poset product

### Code exercise D.5 (TestFinitePosetConstructionProduct)

Write code to compute the product of finite posets.

Implement the interface in Listing 63.

Test using:

```
act4e-check TestFinitePosetConstructionProduct
```

**Listing 63:** The `FinitePosetConstructionProduct` interface.

```
class FinitePosetConstructionProduct(ABC):
    @abstractmethod
    def product(self, ps: Sequence[FinitePoset[C]]) -> FinitePosetProduct[C, Any]: ...
```

**Listing 64:** The `PosetProduct` interface.

```
class PosetProduct(Generic[C, E], Poset[E], ABC):
    """A poset product is a poset that can be factorized."""

    @abstractmethod
    def carrier(self) -> SetProduct[C, E]:
        """Returns the components of the product"""

    @abstractmethod
    def components(self) -> Sequence[Poset[C]]:
        """Returns the components of the product"""
```

**Listing 65:** The `FinitePosetProduct` interface.

```
class FinitePosetProduct(Generic[C, E], FinitePoset[E], PosetProduct[C, E], ABC):
    """Specialization of PosetProduct where we deal with FinitePosets"""

    @abstractmethod
    def carrier(self) -> FiniteSetProduct[C, E]:
        """Returns the components of the product"""

    @abstractmethod
    def components(self) -> Sequence[FinitePoset[C]]:
        """Returns the components"""
```

## 10.3. Disjoint union of posets

Following the pattern, we can define the disjoint union of poset as a poset with the underlying set being the disjoint union of the underlying sets.

### Definition 10.4 (Disjoint union of posets)

Given posets  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  and  $\mathbf{Q} = \langle \mathbf{Q}, \leq_{\mathbf{Q}} \rangle$ , their *disjoint union*  $\mathbf{P} + \mathbf{Q} = \langle \mathbf{P} + \mathbf{Q}, \leq_{\mathbf{P}+\mathbf{Q}} \rangle$ , is the set  $\mathbf{P} + \mathbf{Q}$  equipped with the order  $\leq_{\mathbf{P}+\mathbf{Q}}$  given by

$$\begin{aligned} \leq_{\mathbf{P}+\mathbf{Q}} : (\mathbf{P} + \mathbf{Q}) \times (\mathbf{P} + \mathbf{Q}) &\rightarrow \mathbf{Bool}, \\ \langle \langle 1, x \rangle, \langle 1, y \rangle \rangle &\mapsto (x \leq_{\mathbf{P}} y), \\ \langle \langle 2, \cdot \rangle, \langle 1, \cdot \rangle \rangle &\mapsto \perp, \\ \langle \langle 1, \cdot \rangle, \langle 2, \cdot \rangle \rangle &\mapsto \perp, \\ \langle \langle 2, x \rangle, \langle 2, y \rangle \rangle &\mapsto (x \leq_{\mathbf{Q}} y). \end{aligned} \quad (10)$$

The expression (10) can be intimidating at first, but all it is saying is that the order relation of the disjoint union is obtained by stitching together the two order relations. No element of  $\mathbf{P}$  is related to an element of  $\mathbf{Q}$ , and vice versa.

**Example 10.5.** Consider the posets  $\mathbf{P}, \mathbf{Q}$ , over the sets  $\mathbf{P} = \langle \{\text{🍷, 🍷}\}, \leq_{\mathbf{P}} \rangle$  with  $\text{🍷} \leq_{\mathbf{P}} \text{🍷}$ , and  $\mathbf{Q} = \langle \{\text{🍷, 🍷}\}, \leq_{\mathbf{Q}} \rangle$ , with  $\text{🍷} \leq_{\mathbf{Q}} \text{🍷}$ . Their disjoint union can be represented as in Fig. 3.

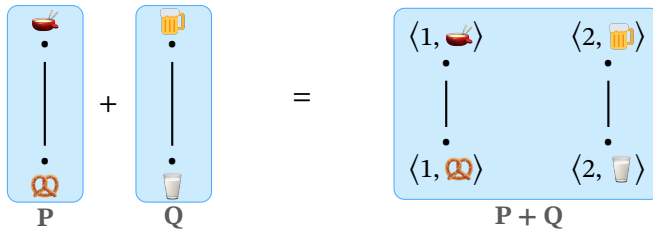
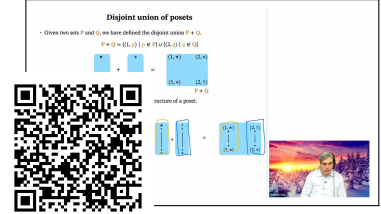


Figure 3.: Disjoint union of posets.

Watch: *Disjoint union of posets* (2 minutes).



### Graded exercise D.4 (MeasurePosetSum)

Prove the following properties:

1. The width of the sum is the sum of the widths:

$$\text{width}(\mathbf{P} + \mathbf{Q}) = \text{width}(\mathbf{P}) + \text{width}(\mathbf{Q}). \quad (11)$$

2. The height of the sum is the maximum of the heights:

$$\text{height}(\mathbf{P} + \mathbf{Q}) = \max(\text{height}(\mathbf{P}), \text{height}(\mathbf{Q})). \quad (12)$$



## 10.4. Disjoint union of posets

### Code exercise D.6 (TestFinitePosetConstructionSum)

Write code to compute the disjoint union of finite posets.

Implement the interface in Listing 66.

Test using:

```
act4e-check TestFinitePosetConstructionSum
```

**Listing 66:** The `FinitePosetConstructionSum` interface.

```
class FinitePosetConstructionSum(ABC):
    @abstractmethod
    def disjoint_union(self, ps: Sequence[FinitePoset[C]]) -> FinitePosetDisjointUnion[C, Any]: ...
```

**Listing 67:** The `PosetDisjointUnion` interface.

```
class PosetDisjointUnion(Generic[C, E], Poset[E], ABC):
    @abstractmethod
    def carrier(self) -> SetDisjointUnion[C, E]: ...

    @abstractmethod
    def components(self) -> Sequence[Poset[C]]: ...
```

**Listing 68:** The `FinitePosetDisjointUnion` interface.

```
class FinitePosetDisjointUnion(Generic[C, E], FinitePoset[E], PosetDisjointUnion[C, E], ABC):
    """Specialization of PosetDisjointUnion where we deal with FiniteSets"""

    @abstractmethod
    def carrier(self) -> FiniteSetDisjointUnion[C, E]: ...

    @abstractmethod
    def components(self) -> Sequence[FinitePoset[C]]: ...
```

## 10.5. Opposite of a poset

### Definition 10.6 (Opposite of a poset)

The *opposite of a poset*  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  is the poset denoted  $\mathbf{P}^{\text{op}} = \langle \mathbf{P}, \leq_{\mathbf{P}}^{\text{op}} \rangle$ . It has the same elements as  $\mathbf{P}$ , but is equipped with the reverse ordering, in the sense that, for all  $x, y \in \mathbf{P}$ ,

$$\frac{x \leq_{\mathbf{P}} y}{y \leq_{\mathbf{P}}^{\text{op}} x}. \quad (13)$$

For a given  $x \in \mathbf{P}$ , we will sometimes write  $x^*$  to denote its corresponding copy in  $\mathbf{P}^{\text{op}}$ , in order to emphasize that  $x$  and  $x^*$  belong to distinct posets. However, often we will not be so pedantic with our notation.

**Example 10.7** (Credit and debt). Let us define the set

$$\mathbf{P} = \{0.00, 0.01, 0.02, \dots\} \subseteq \mathbb{R} \quad (14)$$

of all CHF monetary quantities approximated to the cent. From this set we can define two posets,  $\mathbf{P}^+ = \langle \mathbf{P}, \leq \rangle$  and  $\mathbf{P}^- = \langle \mathbf{P}, \geq \rangle$ , that are the opposite of each other. If the context is that, given two quantities 1 CHF and 2 CHF, we prefer 1 CHF to 2 CHF (for example because it is a cost to pay to acquire a component), then we are working in  $\mathbf{P}^+$ , otherwise we are working in  $\mathbf{P}^-$  (for example because it represents the price at which we are selling our product). Traditionally, in double-entry ledger systems, the numbers were not written with negative signs, but rather in color: red and black. From this convention we get the idioms “being in the black” and “being in the red”.

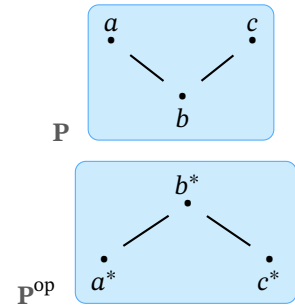


Figure 4.: Opposite of a poset.

## 10.6. ○ Opposite poset

### Code exercise D.7 (TestFinitePosetConstructionOpposite)

Write code to compute the opposite of a poset.

Implement the interface in Listing 69.

Test using:

```
act4e-check TestFinitePosetConstructionOpposite
```

Listing 69: The `FinitePosetConstructionOpposite` interface.

```
class FinitePosetConstructionOpposite(ABC):  
    @abstractmethod  
    def opposite(self, p: FinitePoset[E]) -> FinitePoset[E]: ...
```

*Hint:* You can switch order of the elements passed to the `holds()` mapping.

## 10.7. “Twisted” poset of intervals

An interval of time is defined by two numbers: a lower and upper bound. The notation  $[a, b]$  usually defines an interval on the real line. We can generalize the notion to an interval of a poset.

### Poset intervals

#### Definition 10.8 (Interval)

An *interval* of a poset  $\mathbf{P}$  is a pair of elements  $x, y$  such that  $x \leq_{\mathbf{P}} y$ . We also write  $[x, y]$ , and we identify it with the subset of elements of  $\mathbf{P}$  that are a bounded above and below by the two elements:

$$[x, y] := \{z \in \mathbf{P} : x \leq_{\mathbf{P}} z \leq_{\mathbf{P}} y\}. \quad (15)$$

Note that, following this definition, the empty set is *not* an interval.

### A “twisted” poset of intervals

There are two canonical ways to order poset intervals: a “twisted” version and an “arrow” version. The names are not intuitive at this point: later on, we will see that there exists an “arrow construction” and a “twisted arrow construction” for categories, and they correspond to these constructions when a poset is considered as a category.

#### Definition 10.9 (“Twisted” poset of intervals $\mathbf{Tw} \mathbf{P}$ )

Given a poset  $\mathbf{P}$ , we define a “*twisted*” poset of intervals  $\mathbf{Tw} \mathbf{P}$  by ordering the intervals by inclusion:

$$\frac{[x, y] \leq_{\mathbf{Tw} \mathbf{P}} [z, u]}{[x, y] \subseteq [z, u]}. \quad (16)$$

Equivalently we only need to check the bounds:

$$\frac{[x, y] \leq_{\mathbf{Tw} \mathbf{P}} [z, u]}{(z \leq_{\mathbf{P}} x) \wedge (y \leq_{\mathbf{P}} u)}. \quad (17)$$

**Exercise 23.** Check that the relation defined in Def. 10.9 is indeed a poset.

*See solution on page 199.*

In general,  $\mathbf{Tw} \mathbf{P}$  does not have a top or a bottom.

Watch: *Poset of intervals* (4 minutes).

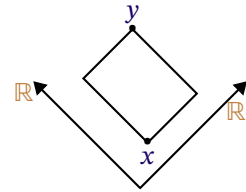
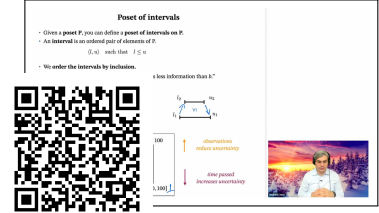


Figure 5.: Poset interval on  $\mathbb{R}^2$ .

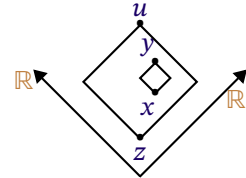


Figure 6.:  $[x, y] \leq_{\mathbf{Tw} \mathbf{P}} [z, u]$

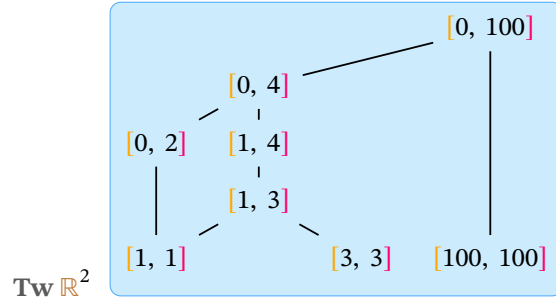


Figure 7.

## Set-based filtering

We now look at an example of **set-based filtering**, where filtering refers to online inference (recursive estimation). Suppose that we want to track the value of a quantity  $x \in [0, 100]$ , without having *a priori* information about  $x$ . We are equipped with sensors, which periodically measure the quantity  $x$  with some variable precision. At time  $t \in \mathbb{R}_{\geq 0}$  they produce an *observation*  $y_t : x_t \in [l_t, u_t]$ . Also, note that the quantity fluctuates randomly, and we bound its “velocity” to be  $\dot{x}_t \in [-1, +1]$  (except at boundaries). At the beginning, our information state  $\bar{i}_0$  could be that  $x \in [0, 100]$ . At time 0, we get an observation  $y_0$ , that says  $x \in [21, 24]$ . The new information state can be obtained by “fusing” the two inputs we have received about  $x$ . This corresponds to the intersection

$$\frac{x \in ([0, 100] \cap [21, 24])}{x \in [21, 24]} \quad (18)$$

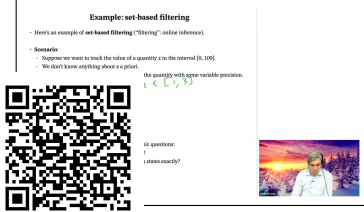
Say we get an observation  $y_1$  which says  $x \in [19, 22]$ . We now need to take into account the evolution/dynamics of the quantity we are tracking. From the interval  $[21, 24]$  we know that the variable could have evolved in  $[20, 25]$  (dynamics are bounded with a unit increase/decrease). Therefore, the new information state is given by

$$\frac{x \in ([20, 25] \cap [19, 22])}{x \in [20, 22]} \quad (19)$$

One of the structures which could sustain this kind of inference, is the poset of twisted intervals (Def. 10.9).

The Hasse diagram representing a situation related to this example could be as reported in Fig. 7.

Watch: *Set-based filtering* (3 minutes).



## 10.8. Twisted poset construction

### Code exercise D.8 (TestFinitePosetConstructionTwisted)

Write code to compute the poset of intervals of a poset according to the twisted interval construction.

Implement the interface in Listing 70.

Test using:

```
act4e-check TestFinitePosetConstructionTwisted
```

**Listing 70:** The `FinitePosetConstructionTwisted` interface.

```
class FinitePosetConstructionTwisted(ABC):
    @abstractmethod
    def twisted(self, s: FinitePoset[C]) -> FinitePosetOfIntervals[C, Any]: ...
```

**Listing 71:** The `PosetOfIntervals` interface.

```
class PosetOfIntervals(Generic[C, E], Poset[E], ABC):
    """A poset of intervals."""

    @abstractmethod
    def construct(self, a: C, b: C) -> E:
        """
        Constructs an interval given the boundaries.

        Must throw ValueError if either a or b is not in the poset,
        or if (a <= b) does not hold.
        """

    @abstractmethod
    def boundaries(self, interval: E) -> Tuple[C, C]:
        """Returns the boundaries of an interval."""
```

**Listing 72:** The `FinitePosetOfIntervals` interface.

```
class FinitePosetOfIntervals(Generic[C, E], PosetOfIntervals[C, E], FinitePoset[E], ABC):
    pass
```

The code requires that you produce a `PosetOfIntervals`. This is a special poset for which we can create elements by naming the two boundary points, and from an element, we can extract the two boundary points.

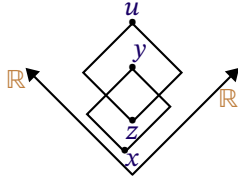


Figure 8.:  $[x, y] \leq_{\text{Arr } P} [z, u]$

## 10.9. Arrow poset of intervals

We can order intervals in a second way, which we call “arrow construction”.

**Definition 10.10** (“Arrow” poset of intervals  $\text{Arr } P$ )

We define an “Arrow” poset of intervals on the poset  $P$  by setting the order:

$$\frac{[x, y] \leq_{\text{Arr } P} [z, u]}{(x \leq_P z) \wedge (y \leq_P u)}. \quad (20)$$

This is similar to taking the product of  $P$  with itself; however, we are only considering intervals, so we obtain a subposet of  $P \times P$ .

**Exercise 24.** Check that the relation defined in Def. 10.10 is indeed a poset.

*See solution on page 199.*

## 10.10. ☉ Arrow poset construction

### Code exercise D.9 (TestFinitePosetConstructionArrow)

Write code to compute the arrow construction (Def. 10.10).

Implement the interface in Listing 73.

Test using:

```
act4e-check TestFinitePosetConstructionArrow
```

**Listing 73:** The `FinitePosetConstructionArrow` interface.

```
class FinitePosetConstructionArrow(ABC):  
    @abstractmethod  
    def arrow(self, s: FinitePoset[C]) -> FinitePosetOfIntervals[C, Any]: ...
```





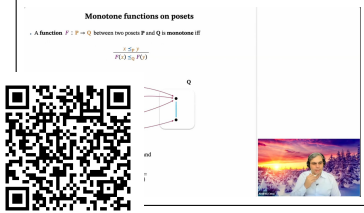


## 11. Monotonicity

Life is hard: to obtain more, you need to work more. Monotonicity is the mathematical concept that captures this principle.

<b>11.1 Monotone maps</b>	<b>190</b>
<b>11.2 Antitone maps</b>	<b>194</b>
<b>11.3 Code exercises - Monotone maps</b>	<b>195</b>
<b>11.4 Compositionality of monotonicity</b>	<b>196</b>
<b>11.5 Order on monotone maps</b>	<b>197</b>

Watch: *Monotone functions on posets* (6 minutes).



## 11.1. Monotone maps

A monotone map is the generalization to posets of a “non-decreasing” function on real numbers. The function  $x \mapsto \max(0, 42x)$  is non-decreasing on the real numbers because

$$\frac{x \leq y}{\max(0, 42x) \leq \max(0, 42y)} . \quad (1)$$

Note that we use “ $\leq$ ” and not “ $<$ ”. “Non-decreasing” is a weaker condition than “increasing”.

The definition of monotone function on a poset is the direct generalization of this concept; the only change is that we use the partial orders at hand, rather than the total order on the reals.

### Definition 11.1 (Monotone map)

A *monotone map* between two posets  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  and  $\mathbf{Q} = \langle \mathbf{Q}, \leq_{\mathbf{Q}} \rangle$  is a function  $f : \mathbf{P} \rightarrow \mathbf{Q}$  that is compatible with the partial-orderings on its source and target in the sense that

$$\frac{x \leq_{\mathbf{P}} y}{f(x) \leq_{\mathbf{Q}} f(y)} . \quad (2)$$

**Example 11.2** (The identity is monotone). Given a poset  $\mathbf{P}$ , the identity function  $\text{id}_{\mathbf{P}} : \mathbf{P} \rightarrow \mathbf{P}$  is a monotone map, since if  $x \leq_{\mathbf{P}} y$ , then  $\text{id}_{\mathbf{P}}(x) = x \leq_{\mathbf{P}} y = \text{id}_{\mathbf{P}}(y)$ .

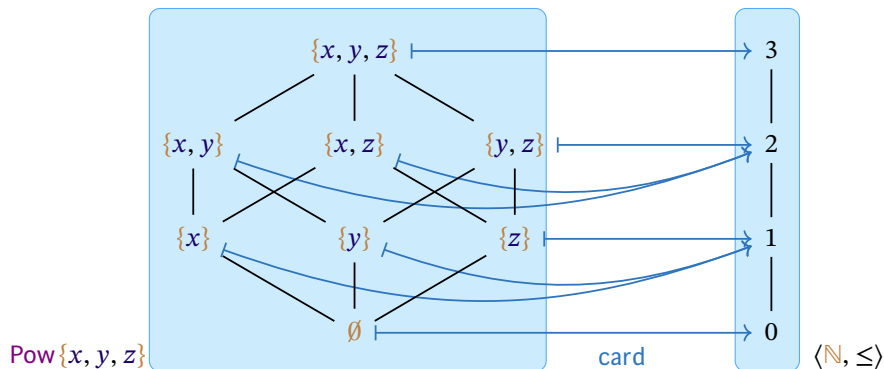
**Example 11.3** (Constant functions). Every constant function is a monotone map.

**Example 11.4** (Cardinality map). Consider the power poset (Def. 9.12)  $\text{Pow } \mathbf{A}$  of a finite set  $\mathbf{A}$ . The cardinality map

$$\text{card} : \text{Pow } \mathbf{A} \rightarrow \mathbb{N} \quad (3)$$

is monotone when considered as a map from the poset  $\text{Pow } \mathbf{A}$  to the poset  $\langle \mathbb{N}, \leq \rangle$ . Figure 1 shows a visualization of this map for the set  $\mathbf{A} = \{x, y, z\}$ . To prove this, recall that in the power poset subsets are ordered by inclusion. Therefore, we need to show that

$$\frac{S \subseteq T}{\text{card}(S) \leq \text{card}(T)} . \quad (4)$$



**Figure 1.:** The cardinality map is a monotone map.

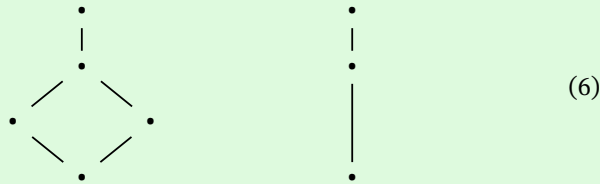
This is easy to see that, because all elements of **S** are also in **T**, the cardinality of **S** cannot be more than the cardinality of **T**. Monotonicity depends on the partial order used on the domain and the codomain. To indicate that a map is monotone, we write it indicating the two posets as the domain/codomain:

$$\text{card} : \langle \text{Pow } \mathbf{A}, \subseteq \rangle \rightarrow \langle \mathbb{N}, \leq \rangle. \quad (5)$$

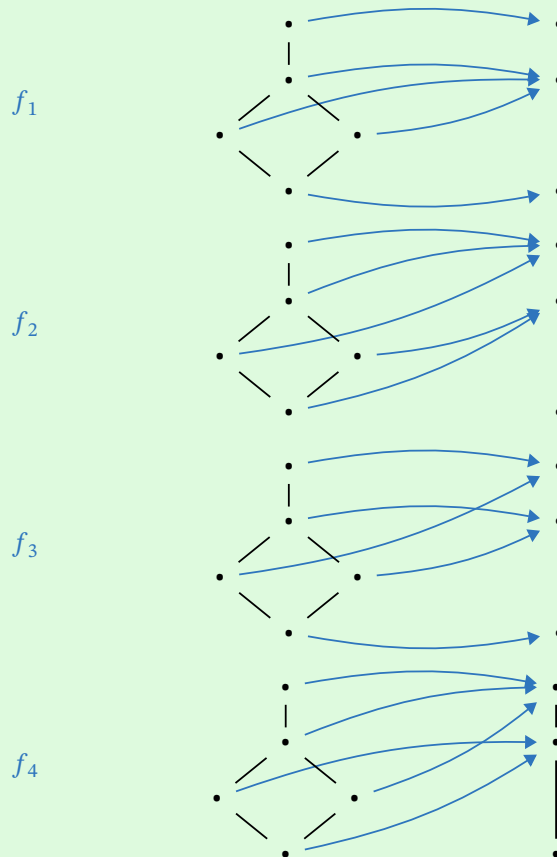
**Graded exercise D.5** (WhichMapsMonotone)

[4 points]

Consider the posets  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  and  $\mathbf{Q} = \langle \mathbf{Q}, \leq_{\mathbf{Q}} \rangle$  described respectively by the following Hasse diagrams.



The following diagrams show functions  $\mathbf{P} \rightarrow \mathbf{Q}$ . We will call them  $f_1, f_2, f_3$  and  $f_4$ , respectively.



Which of the functions  $f_1, f_2, f_3$  and  $f_4$  are monotone maps?

**Lemma 11.5.** Consider a discrete poset  $\mathbf{P}$  and a poset  $\mathbf{Q}$ . Any map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  is monotone.

**Graded exercise D.6** (FromDiscretePosets)  
Prove Lemma 11.5

**Graded exercise D.7** (MonotoneMapCheck)

Prove your answers to the following questions.

1. Is the function

$$\begin{array}{ccc} f : \langle \mathbb{Z}, \leq \rangle & \rightarrow & \langle \mathbb{Z}, \leq \rangle \\ x & \mapsto & x^2 \end{array}$$

monotone?

2. Let  $\mathbf{A} = \{a, b, c\}$  and consider the posets  $\langle \text{Pow } \mathbf{A}, \subseteq \rangle$  and  $\langle \mathbb{N}, \leq \rangle$ . Let

$$\begin{array}{ccc} f : \text{Pow } \mathbf{A} & \rightarrow & \mathbb{N} \\ S & \mapsto & \text{card}(S) \end{array}$$

be the function which calculates the cardinality of any subset of  $\mathbf{A}$ . Is  $f$  monotone?

3. Consider the set of natural numbers which divide the number 36, equipped with the partial order “ $\leq$ ” such that  $x \leq y$  if and only if  $x$  divides  $y$ . Call this poset  $\mathbf{P} = \langle \mathbf{P}, \leq \rangle$ , and let  $f : \mathbf{P} \rightarrow \{\perp, \top\}$  be defined by

$$f(x) = \begin{cases} \top & \text{if } x \text{ is an even number,} \\ \perp & \text{if } x \text{ is an odd number.} \end{cases} \quad (7)$$

Is  $f$  monotone if we equip  $\{\perp, \top\}$  with the usual partial order such that  $\perp \leq \top$ ?

## 11.2. Antitone maps

Dually to monotone functions, we can define antitone maps as order *reversing* functions.

**Definition 11.6** (Antitone map)

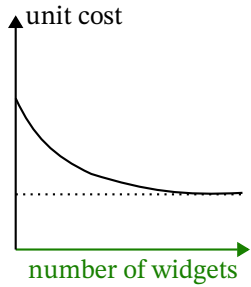
An *antitone map* between two posets  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  and  $\mathbf{Q} = \langle \mathbf{Q}, \leq_{\mathbf{Q}} \rangle$  is a map  $f$  that reverses the ordering, in the sense that

$$\frac{x \leq_{\mathbf{P}} y}{f(x) \geq_{\mathbf{Q}} f(y)} . \quad (8)$$

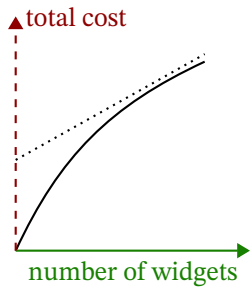
**Example 11.7** (Unit cost, total cost). Assume that you want to produce some widgets, and that the manufacturing cost depends on the number of widgets. The function describing the total cost  $t : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$  is a map between the ordered sets  $\mathbb{N}$  and  $\mathbb{R}_{\geq 0}$ , and maps each quantity of widgets to a total manufacturing cost (Fig. 2b). Clearly,  $t$  is a monotone function. Conversely, the unit cost function  $u : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$  is antitone (Fig. 2a).

It is easy to see that an antitone map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  is the same thing as a monotone map  $f : \mathbf{P}^{\text{op}} \rightarrow \mathbf{Q}$ .

**Lemma 11.8.** An antitone map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  is a monotone map  $f : \mathbf{P}^{\text{op}} \rightarrow \mathbf{Q}$  and a monotone map  $f : \mathbf{P} \rightarrow \mathbf{Q}^{\text{op}}$ .



(a) Unit cost vs number of widgets.



(b) Total cost vs number of widgets.

**Figure 2.:** Unit and total costs vs. number of widgets.

## 11.3. ☉ Monotone maps

### Code exercise D.10 (TestFiniteMonotoneMapProperties)

Check if a map is monotone or antitone.

Implement the interface in Listing 74.

Test using:

```
act4e-check TestFiniteMonotoneMapProperties
```

**Listing 74:** The `FiniteMonotoneMapProperties` interface.

```
class FiniteMonotoneMapProperties(ABC):
    @abstractmethod
    def is_monotone(self, p1: FinitePoset[E1], p2: FinitePoset[E2], m: FiniteMap[E1, E2]) -> bool:
        """Check if a map is monotone."""

    @abstractmethod
    def is_antitone(self, p1: FinitePoset[E1], p2: FinitePoset[E2], m: FiniteMap[E1, E2]) -> bool:
        """Check if a map is antitone."""
```

*Hint:* You can reduce `is_antitone()` to a call of `is_monotone()` by using opposite posets.



## 11.4. Compositionality of monotonicity

Monotonicity is a compositional property: the series composition of two monotone maps is monotone.

**Lemma 11.9.** Given posets  $\mathbf{P}, \mathbf{Q}, \mathbf{R}$  and two monotone maps  $f : \mathbf{P} \rightarrow \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow \mathbf{R}$ , the composite map  $f \circ g : \mathbf{P} \rightarrow \mathbf{R}$  is monotone as well.

*Proof.* Consider  $p_1, p_2 \in \mathbf{P}$ ,  $q_1, q_2 \in \mathbf{Q}$ . By assuming that  $f$  and  $g$  are monotone, we have

$$\frac{p_1 \leq_{\mathbf{P}} p_2}{f(p_1) \leq_{\mathbf{Q}} f(p_2)} \quad (9)$$

and

$$\frac{q_1 \leq_{\mathbf{Q}} q_2}{g(q_1) \leq_{\mathbf{R}} g(q_2)}. \quad (10)$$

By substituting the above in the map composition formula, we have

$$\frac{p_1 \leq_{\mathbf{P}} p_2}{(f \circ g)(p_1) \leq_{\mathbf{R}} (f \circ g)(p_2)}, \quad (11)$$

which is the monotonicity condition for the composite map  $(f \circ g)$ .  $\square$

## 11.5. Order on monotone maps

Fixed two posets  $\mathbf{P}$  and  $\mathbf{Q}$ , the set of monotone maps  $\mathbf{P} \rightarrow \mathbf{Q}$  form a poset themselves. We can order them point wise.

**Definition 11.10** (Order on monotone maps)

Consider two monotone maps  $f, g : \mathbf{P} \rightarrow \mathbf{Q}$ . We say that  $f$  precedes  $g$  if, point wise, the output of  $f$  precedes the output of  $g$  when presented with the same input:

$$\frac{f \leq_{\mathbf{P} \rightarrow \mathbf{Q}} g}{\forall x \in \mathbf{P} : f(x) \leq_{\mathbf{Q}} g(x)} . \quad (12)$$

**Example 11.11** (Rounding functions). In this example we look at “rounding functions”: these are functions that truncate a real number to an integer. You might already know the ceiling function `ceil` (Fig. 4a) and the floor function `floor` (Fig. 4b), which are formally defined as

$$\begin{aligned} \text{ceil} : \langle \mathbb{R}, \leq \rangle &\rightarrow \langle \mathbb{N}, \leq \rangle, \\ x &\mapsto \min \{y \in \mathbb{N} : y \geq x\}, \end{aligned} \quad (13)$$

and

$$\begin{aligned} \text{floor} : \langle \mathbb{R}, \leq \rangle &\rightarrow \langle \mathbb{N}, \leq \rangle, \\ x &\mapsto \max \{y \in \mathbb{N} : y \leq x\}. \end{aligned} \quad (14)$$

The functions `ceil` and `floor` are monotone, since  $x \leq z$  implies both  $\text{ceil}(x) \leq \text{ceil}(z)$  and  $\text{floor}(x) \leq \text{floor}(z)$ .

There exist many other rounding functions, commonly used by computers. For example, the map “round to nearest, ties to even” [12] rounds a number to the closest integer, and in case of ties it rounds to the even one (Fig. 4c). For example, 3.2 is mapped to 3, 1.5 is mapped to 2, and 4.5 is mapped to 4. This is the formal definition:

$$\begin{aligned} \text{rnttte} : \langle \mathbb{R}, \leq \rangle &\rightarrow \langle \mathbb{N}, \leq \rangle, \\ x &\mapsto \begin{cases} \text{floor}(x), & x < (\text{floor}(x) + \text{ceil}(x))/2 \\ \text{ceil}(x), & x > (\text{floor}(x) + \text{ceil}(x))/2 \\ \text{ceil}(x), & (x = (\text{floor}(x) + \text{ceil}(x))/2) \wedge (\text{ceil}(x) \text{ is even}) \\ \text{floor}(x), & (x = (\text{floor}(x) + \text{ceil}(x))/2) \wedge (\text{floor}(x) \text{ is even}) \end{cases} . \end{aligned} \quad (15)$$

In this example, note that

$$\text{floor} \leq \text{id} \leq \text{ceil} \quad (16)$$

and

$$\text{floor} \leq \text{rnttte} \leq \text{ceil}, \quad (17)$$

and `id` and `rnttte` are not comparable (see Fig. 3).

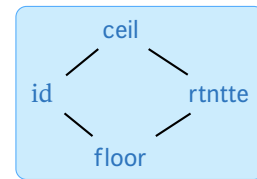
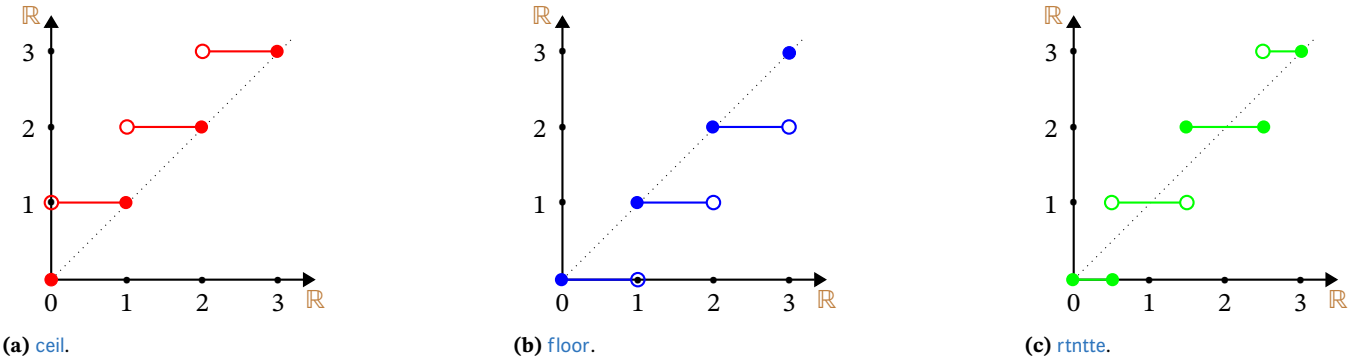


Figure 3.



**Figure 4.:** Comparison of three rounding methods.

## Solutions to selected exercises

**Solution of Exercise 20.**  $\mathbf{P}$  is a pre-order, because it satisfies reflexivity and transitivity.  $\mathbf{Q}$  violates both reflexivity (e.g.,  $\langle x, x \rangle$  is missing), and transitivity (e.g.,  $\langle x, z \rangle$  is missing).

**Solution of Exercise 21.** The reachability relationship in any directed graph does not define a poset. As a simple counterexample, take a graph with nodes  $\{x, y, z\}$  and paths  $x$  to  $y$ ,  $y$  to  $z$ , and  $z$  to  $x$ . From transitivity, one has  $x \leq z$ , but from reachability we also have  $z \leq x$ . Therefore, per antisymmetry one should have  $x = z$ , but these are actually distinct nodes. To make things work, one needs to consider only acyclic graphs.

**Solution of Exercise 22.** Consider a set  $\mathbf{A}$ . Clearly, given  $\mathbf{S} \in \mathbf{Pow} \mathbf{A}$ , we have  $\mathbf{S} \subseteq \mathbf{S}$ . Furthermore, given also  $\mathbf{T} \in \mathbf{Pow} \mathbf{A}$ , we have

$$\frac{\mathbf{S} \subseteq \mathbf{T} \quad \mathbf{T} \subseteq \mathbf{S}}{\mathbf{S} = \mathbf{T}}. \quad (18)$$

Finally, given also  $\mathbf{U} \in \mathbf{Pow} \mathbf{A}$ , we have

$$\frac{\mathbf{S} \subseteq \mathbf{T} \quad \mathbf{T} \subseteq \mathbf{U}}{\mathbf{S} \subseteq \mathbf{U}}. \quad (19)$$

**Solution of Exercise 23.** We check the three conditions.

- ▷ First, we know that  $[p_1, q_1] \leq_{\mathbf{Tw} \mathbf{P}} [p_1, q_1]$ , since  $p_1 \leq_{\mathbf{P}} p_1$  and  $q_1 \leq_{\mathbf{P}} q_1$ .
- ▷ Second,  $[p_1, q_1] \leq_{\mathbf{Tw} \mathbf{P}} [p_2, q_2]$  and  $[p_2, q_2] \leq_{\mathbf{Tw} \mathbf{P}} [p_3, q_3]$  imply

$$[p_1, q_1] \leq_{\mathbf{Tw} \mathbf{P}} [p_3, q_3]. \quad (20)$$

- ▷ Third, if  $[p_1, q_1] \leq_{\mathbf{Tw} \mathbf{P}} [p_2, q_2]$  and  $[p_2, q_2] \leq_{\mathbf{Tw} \mathbf{P}} [p_1, q_1]$ , then  $p_1 = p_2$  and  $q_1 = q_2$ .

**Solution of Exercise 24.** We check the three conditions.

- ▷ First, we know that  $[p_1, q_1] \leq_{\mathbf{Arr} \mathbf{P}} [p_1, q_1]$ , since  $p_1 \leq_{\mathbf{P}} p_1$  and  $q_1 \leq_{\mathbf{P}} q_1$ .
- ▷ Second,  $[p_1, q_1] \leq_{\mathbf{Arr} \mathbf{P}} [p_2, q_2]$  and  $[p_2, q_2] \leq_{\mathbf{Arr} \mathbf{P}} [p_3, q_3]$  imply

$$[p_1, q_1] \leq_{\mathbf{Arr} \mathbf{P}} [p_3, q_3]. \quad (21)$$

- ▷ Third, if  $[p_1, q_1] \leq_{\mathbf{Arr} \mathbf{P}} [p_2, q_2]$  and  $[p_2, q_2] \leq_{\mathbf{Arr} \mathbf{P}} [p_1, q_1]$ , then  $p_1 = p_2$  and  $q_1 = q_2$ .



# PART E. CATEGORIES



---

<b>12. Graphs</b>	<b>203</b>
<b>13. (Semi)categories</b>	<b>209</b>
<b>14. Categories and structures</b>	<b>231</b>
<b>15. Modeling with categories</b>	<b>241</b>
<b>16. Semicategory actions</b>	<b>259</b>
<b>17. Constructing categories</b>	<b>285</b>
<b>18. Culture</b>	<b>293</b>

---

Hiking (“wandern” in german) is one of the main sport activities in Switzerland, often referred to as the “national sport”. On average, 520 million kilometres (in 130 million hours) are travelled every year by the Swiss. In Switzerland, the total hiking trail network is about 65,000 kilometres.







# 12. Graphs

In this chapter we give a formal description of graphs. Graphs are data structures with “points” and “arrows”: it is the first time we encounter a data structure with two kinds of elements.

12.1 Graphs . . . . .	204
12.2 Graph homomorphisms . . . . .	206

*Röstli* is a Swiss dish, consisting of fried potatoes. Originally a breakfast dish eaten by farmers in the canton Bern, it is now eaten all over Switzerland at any meal. It is typically served to accompany other dishes, and therefore considered as a side dish.



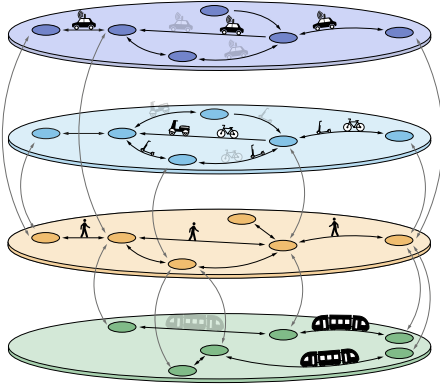


Figure 1.: Intermodal mobility network of a city.

## 12.1. Graphs

A graph is a data structure with “points” and “arrows”, usually called *nodes/vertices* and *arcs/edges*.

Graphs are widely used in many engineering disciplines, to represent, formulate, and solve complex problems. For instance, we can represent the intermodal mobility network in a city as a directed graph (Fig. 1). The word “intermodal” means that one can jump from a mobility option to another. For instance, in the figure, you can spot the graphs for an autonomous vehicle mobility service, a micromobility service, the subway service, and roads on which you can walk. The vertices represent locations, and the edges represent different travel routes connecting the locations.

### Defining graphs

The usual definition of directed graph in engineering, which we will *not* use, is as follows:

**Definition 12.1** (Directed Graph)

A *directed graph* is a pair  $\mathcal{G} = \langle \mathbf{V}, \mathbf{E} \rangle$ , where  $\mathbf{V}$  is a set of vertices and  $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$  is a set of edges.

In this definition, an edge is a pair of vertices  $\langle x, y \rangle$  where  $x$  is the source and  $y$  is the target. One limitation of this notion of graph is that we can only have *one* edge between two vertices in either direction.

The following definition is more expressive, though a bit more abstract.

**Definition 12.2** (Directed Multigraph)

A *directed multigraph*  $\mathcal{G} = \langle \mathbf{V}, \mathbf{E}, \text{src}, \text{tgt} \rangle$  consists of a set of vertices  $\mathbf{V}$ , a set of edges  $\mathbf{E}$ , and two functions  $\text{src}, \text{tgt} : \mathbf{E} \rightarrow \mathbf{V}$ , called the *source* and *target* functions, respectively. Given  $a \in \mathbf{E}$  with  $\text{src}(a) = v$  and  $\text{tgt}(a) = w$ , we say that  $a$  is an edge (or arrow) from  $v$  to  $w$ .

Both directed graphs and undirected graphs play a prominent role in many kinds of mathematics. In this text, we work primarily with directed multigraphs and so, from now on, we drop the “directed” and the “multi”: unless indicated otherwise, the word “graph” will mean “directed multigraph”.

### Paths

**Definition 12.3** (Paths)

A *path* in a graph  $\mathcal{G} = \langle \mathbf{V}, \mathbf{E}, \text{src}, \text{tgt} \rangle$  is:

Constituents

- ▷ a list of edges  $[e_1, \dots, e_n]_{\mathbf{E}}$ , with  $n \in \mathbb{N}$ .
  - If  $n \neq 0$ , the source of a path  $[e_1, \dots, e_n]_{\mathbf{E}}$  is defined as  $\text{src}(e_1)$  and its target is  $\text{tgt}(e_n)$ .
  - If  $n = 0$ , we speak of a “trivial path” or an “empty path” and we must additionally specify an element  $x \in \mathbf{V}$  which is designated as both the source and target of the path. If paths describe a journey, then trivial paths correspond to “not going anywhere”.

Conditions

- ▷ if  $n \geq 2$ , we require that, for any two subsequent edges  $e_i$  and  $e_{i+1}$  in  $[e_1,$

$\dots, e_n]_{\mathbf{E}}$ ,

$$\text{tgt}(e_i) = \text{src}(e_{i+1}). \quad (1)$$

The length of  $[e_1, \dots, e_n]_{\mathbf{E}}$  is called the *length* of the path.

## 12.2. Graph homomorphisms

**Definition 12.4** (Graph homomorphism)

Given graphs  $\mathcal{G}_1 = \langle \mathbf{V}_1, \mathbf{E}_1, \text{src}_1, \text{tgt}_1 \rangle$  and  $\mathcal{G}_2 = \langle \mathbf{V}_2, \mathbf{E}_2, \text{src}_2, \text{tgt}_2 \rangle$ , a *graph homomorphism*  $F : \mathcal{G}_1 \rightarrow \mathcal{G}_2$  is given by two maps

$$F_{\bullet} : \mathbf{V}_1 \rightarrow \mathbf{V}_2, \quad (2)$$

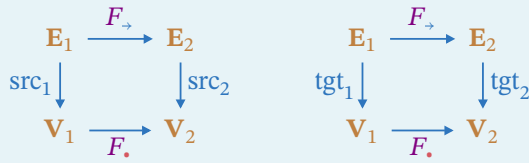
$$F_{\rightarrow} : \mathbf{E}_1 \rightarrow \mathbf{E}_2, \quad (3)$$

such that

$$F_{\rightarrow} \circ \text{src}_2 = \text{src}_1 \circ F_{\bullet}, \quad (4)$$

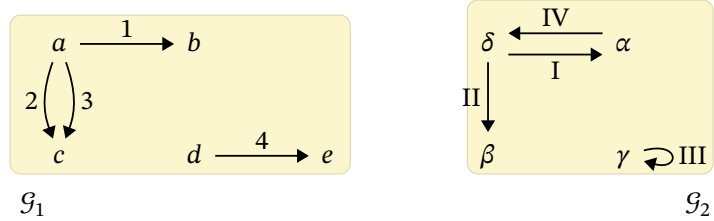
$$F_{\rightarrow} \circ \text{tgt}_2 = \text{tgt}_1 \circ F_{\bullet}, \quad (5)$$

or, in other words, that the following diagrams commute:

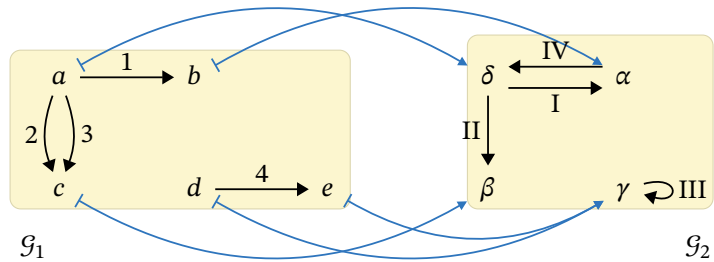


**Remark 12.5.** Intuitively, all this is saying is that “arrows are bound to their vertices”, meaning that if a vertex  $v_1$  is connected to  $v_2$  via an arrow  $a$ , the vertices resulting from the application of the maps on nodes  $F_{\bullet}(v_1)$  and  $F_{\bullet}(v_2)$  have to be connected via the arrow resulting from the application of the map on arrows  $F_{\rightarrow}(a)$ .

**Example 12.6.** Consider the two graphs,  $\mathcal{G}_1$  and  $\mathcal{G}_2$  depicted in Fig. 2.



**Figure 2.:** Example of graphs for graph homomorphism.

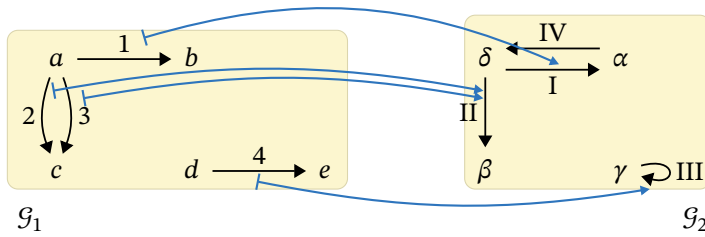


**Figure 3.:**  $F_{\bullet}$  for the presented graph homomorphism.

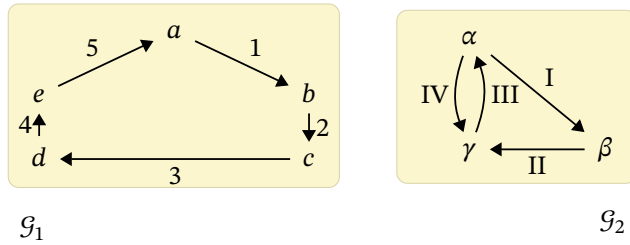
A possible graph homomorphism between the two is given by  $F_{\bullet}, F_{\rightarrow}$  graphically defined as in Fig. 3 and Fig. 4, respectively.

**Example 12.7** (Counterexample). By considering the graphs in Example 12.6, we could define  $F_{\bullet}, F_{\rightarrow}$  in the same way, exception made for  $F_{\bullet}(e) = x$ . Clearly, this would violate the commuting diagrams condition.

**Exercise 25.** Consider the two graphs depicted in Fig. 5.



**Figure 4.:**  $F_+$  for the presented graph homomorphism.

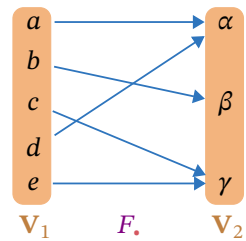


**Figure 5.**

Furthermore, consider the map  $F_+$  depicted in Fig. 6.

Find a map  $F_+$  such that  $F_+, F_+$  describe a graph homomorphism between  $\mathcal{G}_1$  and  $\mathcal{G}_2$ .

See solution on page 299.



**Figure 6.**





## 13. (Semi)categories

In this chapter we look at the fundamental notion of a category, and also its cousin, the notion of semicategory. We will see that categories generalize many of mathematical structures that we have studied in this book so far.

Although categories are more of a protagonist in this book, we introduce semicategories first because they are more rudimentary. The step from semicategories to categories will then be very similar to the step from semigroups to monoids.

<b>13.1 Interfaces</b> . . . . .	<b>210</b>
<b>13.2 Semicategories</b> . . . . .	<b>212</b>
<b>13.3 Categories</b> . . . . .	<b>216</b>
<b>13.4 Diagrams</b> . . . . .	<b>219</b>
<b>13.5 Categories vs graphs</b> . . . . .	<b>221</b>
<b>13.6 Categories from graphs</b> . . . . .	<b>222</b>
<b>13.7 Code exercises - Semicategories</b>	<b>224</b>

### 13.1. Interfaces

One way to understand semicategories is to see them as a generalization of semigroups. In semigroups, monoids, and groups we could take any two elements and compose them: the elements always had a “compatible” interface.

To motivate the need for interfaces, consider the ropes of Chapter 2, which had this composition rule:

$$\begin{array}{c}
 \text{---} a \text{---} \quad \text{---} b \text{---} \\
 \hline
 \text{---} a + b \text{---}
 \end{array}
 \quad (1)$$

Two chapters later, we can recognize that we were describing the monoid  $\langle \mathbb{R}_{\geq 0}, +, 0 \rangle$ . Being a monoid, all pieces of rope are compatible and can be composed.

A first step towards discussing interfaces is to think of things that have a direction. For example, consider extension cords. Let  $\text{---} c \text{---}$  be an extension cord of length  $c$ . If you have an extension cord of length  $c$  and another of length  $d$ , you can plug them together to get an extension cord of length  $c + d$ :

$$\begin{array}{c}
 \text{---} c \text{---} \quad \text{---} d \text{---} \\
 \hline
 \text{---} c + d \text{---}
 \end{array}
 \quad (2)$$

In this form, this is still the same monoid.

But suppose now that, reading this book, you fall in love with Switzerland and want to visit. As you start to plan your trip, at some point you need to think about electrical adapters. Switzerland uses the connector of type N (Fig. 1). If you come from Ireland, your appliances use type G. Now when we think of extension cords, we might allow either end to have a plug type. These would be Irish and Swiss extension cords of length  $\ell$ :

$$\begin{array}{c}
 \text{---} \ell \text{---} \quad \text{---} \ell \text{---} \\
 \hline
 \text{---} \ell \text{---}
 \end{array}
 \quad (3)$$

You might want a cord that has a Swiss male end and an Irish female end:

$$\begin{array}{c}
 \text{---} \ell \text{---} \\
 \hline
 \text{---} \ell \text{---}
 \end{array}
 \quad (4)$$

Unfortunately these devices don’t exist. What you can buy are adapters, which we can think of extension cord of length zeros:

$$\begin{array}{c}
 \text{---} 0 \text{---} \\
 \hline
 \text{---} 0 \text{---}
 \end{array}
 \quad (5)$$

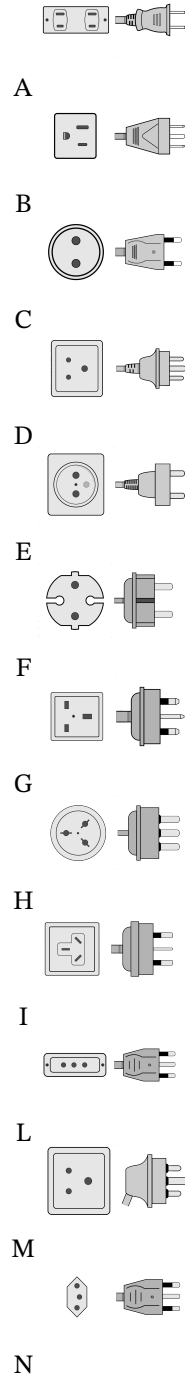
If you have an adapter, then you can attach an extension cord to it to obtain (4):

$$\begin{array}{c}
 \text{---} \ell \text{---} \quad \text{---} 0 \text{---} \\
 \hline
 \text{---} \ell \text{---}
 \end{array}
 \quad (6)$$

The general formula to compose cords with generic types  $X, Y, Z$  is

$$\frac{X \text{ --- } a \text{ --- } Y \quad Y \text{ --- } b \text{ --- } Z}{X \text{ --- } a + b \text{ --- } Z} \quad (7)$$

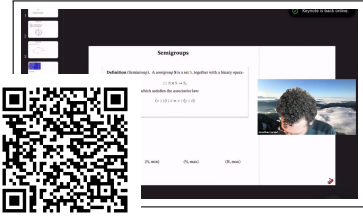
This kind of composition of things that have an input and an output interface, like cords, can be modeled by the notions of semicategory and category.



**Figure 1.:** Plug/socket types used in the world



Watch: *Introduction for Semicategories and Categories* (14 minutes).



## 13.2. Semicategories

We begin to introduce the concept of a semicategory by listing key aspects of the electrical cords example in the previous section which each correspond to a part of the formal definition of semicategory.

### Objects, morphisms, composition

Firstly, in the cords example, we have a set of types of electrical interfaces. In a semicategory, the things that play the role of interfaces are called *objects*. (In this text, we often denote generic objects by letters  $X, Y, Z$ , etc.)

Secondly, in the electrical cords example, the cords themselves connect two interfaces and have a directionality: one end has a socket, the other end has a plug. In a semicategory, the things that play the role of cords are called *morphisms*. (This is a word we've already gotten to know, and the connection here is not accidental.) The object that denotes the type of the “socket end” of the morphism is called the *domain* or *source* of the morphism. The object that denotes the “plug end” of a morphism is called its *codomain* or *target*. So morphisms are directed: they go from their source to their target.

To visualize morphisms, we often draw arrows. For example, if  $X$  and  $Y$  are objects in some semicategory, then we draw a morphism from  $X$  to  $Y$  (call it  $f$ ) like this:

$$f : X \rightarrow Y. \quad (8)$$

So here  $X$  is the source of  $f$ , and  $Y$  is its target.

Thirdly, a key feature of the cords example is that we can compose two cords and the result is again a chord. However, for this to work, the plug-end of the first chord must match the socket-end of the other. Similarly, in a semicategory we specify a way to compose two morphisms, provided that the target object of the first morphism matches the source object of the second morphism. (When this is the case, the morphisms are said to be *composable*.)

In other words, if  $f : X \rightarrow Y$  and  $g : Y \rightarrow Z$  are morphisms in some semicategory (note: they are composable), then the semicategory has an operation for composing them, and the result is another morphism. Our notation for composition of  $f$  and  $g$  is  $f \circ g$ , read “ $f$  then  $g$ ”. (Once again: this is in contrast to the more traditional notation  $g \circ f$ .) Thus, we have

$$\frac{f : X \rightarrow Y \quad g : Y \rightarrow Z}{(f \circ g) : X \rightarrow Z}, \quad (9)$$

which is analogous to (7) in the cords example.

### Associativity

So far we have described the building blocks, or constituents, of a semicategory: objects, morphisms, and composition operations. We also want these to obey a certain condition called the *associativity law*. This condition says that if we are given a string of three composable morphisms, then it doesn't matter in which order we choose to compose them:

$$\frac{f : X \rightarrow Y \quad g : Y \rightarrow Z \quad h : Z \rightarrow U}{(f \circ g) \circ h = f \circ (g \circ h)}. \quad (10)$$

This is analogous to the fact, in the electrical cords example, that if we look at

three cords connected together, we cannot tell if the first two were connected together first and then the result was connected to the third, or if the connecting happened the other way around.

## The definition of a semicategory

Here is the full formal definition of a semicategory. Have a read through, then we illustrate it further with examples.

### Definition 13.1 (Semicategory)

A *semicategory*  $\mathbf{C}$  is specified by:

#### Constituents

1. Objects: A collection\*  $\mathbf{Ob}_{\mathbf{C}}$  whose elements are called *objects*.
2. Morphisms: For every pair of objects  $X, Y$  in  $\mathbf{Ob}_{\mathbf{C}}$ , there is a set called a “hom-set” and indicated as  $\mathbf{Hom}_{\mathbf{C}}(X; Y)$ , elements of which are called *morphisms* and denoted  $f : X \rightarrow Y$ .

For such an  $f$ , we call  $X$  its *source* and  $Y$  its *target*.

3. Composition operations: For every three objects  $X, Y, Z$  in  $\mathbf{Ob}_{\mathbf{C}}$  there is a composition map

$$\circ_{X,Y,Z} : \mathbf{Hom}_{\mathbf{C}}(X; Y) \times \mathbf{Hom}_{\mathbf{C}}(Y; Z) \rightarrow \mathbf{Hom}_{\mathbf{C}}(X; Z). \quad (11)$$

We usually just write  $\circ$  instead of  $\circ_{X,Y,Z}$ :

$$\frac{f : X \rightarrow Y \quad g : Y \rightarrow Z}{(f \circ g) : X \rightarrow Z}. \quad (12)$$

The morphism  $f \circ g$  is called the *composition* of  $f$  and  $g$ .

#### Conditions

1. Associativity: it holds that

$$\frac{f : X \rightarrow Y \quad g : Y \rightarrow Z \quad h : Z \rightarrow U}{(f \circ g) \circ h = f \circ (g \circ h)}. \quad (13)$$

**Remark 13.2.** We denote composition of morphisms using the symbol “ $\circ$ ” (pronounced “then”), as already introduced for functions in Section 4.10. This is in contrast to the more common notation for composition, namely  $g \circ f$ , or simply  $gf$ , which reads as “ $g$  after  $f$ ”. As usual,  $f^2$  denotes  $f \circ f$ ,  $f^3$  denotes  $f \circ f \circ f$ , and so on.

**Remark 13.3.** 1. When we want to emphasize which semicategory we are working with, we will sometimes write

$$f : X \rightarrow_{\mathbf{C}} Y \quad (14)$$

to indicate

$$f \in \mathbf{Hom}_{\mathbf{C}}(X; Y). \quad (15)$$

2. Sometimes we will use the notation  $\mathbf{Mor}_{\mathbf{C}}$  to denote the collection of *all* morphisms in a semicategory  $\mathbf{C}$ , not just a certain hom-set.

**Remark 13.4.** We will often visualize objects and morphisms using diagrams where symbols or dots indicate objects and arrows indicate morphisms. For

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

Watch: *Semicategories* (8 minutes).



$$X \xrightarrow{f} Y \xrightarrow{g} Z$$

Figure 2.

$$\begin{array}{c} X \xrightarrow{f} Y \xrightarrow{g} Z \\ \quad \quad \quad \curvearrowright \\ \quad \quad \quad f \circ g \end{array}$$

Figure 3.

instance, if  $X, Y, Z$  are objects in a semicategory  $\mathbf{C}$ , and  $f : X \rightarrow Y$  and  $g : Y \rightarrow Z$  are morphisms, we draw this as in Fig. 2, for example. The composition  $f \circ g$  of  $f$  with  $g$  corresponds to yet another arrow, as in Fig. 3.

**Example 13.5.** There is a semicategory whose objects are all sets, morphisms are functions between sets, and the composition operations are the usual composition of functions.

Let us also check that composition of functions does indeed satisfy the associativity law. Suppose  $f : X \rightarrow Y$ ,  $g : Y \rightarrow Z$ , and  $h : Z \rightarrow U$  are three composable functions, and let  $x \in X$  be an arbitrary element. On the one hand

$$((f \circ g) \circ h)(x) = h((f \circ g)(x)) = h(g(f(x))), \quad (16)$$

while on the other hand

$$(f \circ (g \circ h))(x) = (g \circ h)(f(x)) = h(g(f(x))). \quad (17)$$

So

$$(f \circ g) \circ h = f \circ (g \circ h) \quad (18)$$

holds.

**Example 13.6.** This example is an extension of Example 6.12. We will describe a semicategory  $\mathbf{C}$  with two objects,  $\text{Ob}_{\mathbf{C}} = \{\mathbf{A}, \mathbf{B}\}$ . Each of the objects is a set which describes possible states of a plant. Let

$$\mathbf{A} = \{\text{sprout, young, mature, old, dead}\}, \quad (19)$$

and

$$\mathbf{B} = \{\text{alive, dead}\}. \quad (20)$$

Also, let  $f : \mathbf{A} \rightarrow \mathbf{A}$  be the function with

$$\begin{aligned} f(\text{sprout}) &= \text{young}, \\ f(\text{young}) &= \text{mature}, \\ f(\text{mature}) &= \text{old}, \\ f(\text{old}) &= \text{dead}, \\ f(\text{dead}) &= \text{dead}; \end{aligned} \quad (21)$$

let  $g : \mathbf{B} \rightarrow \mathbf{B}$  be the function with

$$\begin{aligned} g(\text{alive}) &= \text{dead}, \\ g(\text{dead}) &= \text{dead}; \end{aligned} \quad (22)$$

and let  $h : \mathbf{A} \rightarrow \mathbf{B}$  be the function with

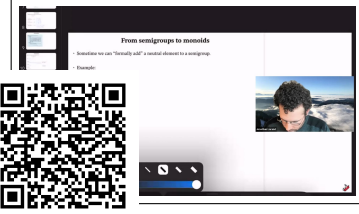
$$\begin{aligned} h(\text{sprout}) &= \text{alive}, \\ h(\text{young}) &= \text{alive}, \\ h(\text{mature}) &= \text{alive}, \\ h(\text{old}) &= \text{alive}, \end{aligned} \quad (23)$$

For our sets of morphisms, we let

$$\begin{aligned}
 \text{Hom}_{\mathbf{C}}(\mathbf{A}; \mathbf{A}) &= \{f, f^2, f^3, f^4\}, \\
 \text{Hom}_{\mathbf{C}}(\mathbf{B}; \mathbf{B}) &= \{g\}, \\
 \text{Hom}_{\mathbf{C}}(\mathbf{A}; \mathbf{B}) &= \{h, f \circ h, f^2 \circ h, f^3 \circ h, f^4 \circ h\}, \\
 \text{Hom}_{\mathbf{C}}(\mathbf{B}; \mathbf{A}) &= \emptyset;
 \end{aligned} \tag{24}$$

and for the composition operations, we define these to be the usual composition of functions, which we know obeys the associative law. Thus, we have a semicategory.

Watch: *Categories* (24 minutes).



### 13.3. Categories

Now we come to one of the central protagonists of this book: the concept of a category. Categories are like semicategories, but with one more ingredient added: identity morphisms. A good analogy is that categories are to semicategories as monoids are to semigroups. A monoid is a semigroup that additionally has an identity element, and similarly a category is a semicategory that additionally has identity morphisms.

#### Identity morphisms

One might say that identity morphisms are morphisms that “do nothing”: they do not have any effect when we compose with them. This is analogous to how the identity element of a monoid “does nothing” when we multiply it with other elements of the monoid.

**Definition 13.7** (Identity morphisms)

Let  $\mathbf{C}$  be a semicategory. An *identity morphism*, or just *identity*, for an object  $X$  of  $\mathbf{C}$  is a morphism

$$\text{id}_X : X \rightarrow X \quad (25)$$

in  $\mathbf{C}$  that acts neutrally with respect to composition with any morphism in the category with which it is composable:

$$\frac{f : W \rightarrow X}{f \circ \text{id}_X = f}, \quad (26)$$

and

$$\frac{g : X \rightarrow Y}{\text{id}_X \circ g = g}. \quad (27)$$

**Remark 13.8.** If an identity morphism  $\text{id}_X$  for an object  $X$  exists, then it is unique. To see this, observe that  $\text{Hom}_{\mathbf{C}}(X; X)$  is a semigroup, and  $\text{id}_X$  is a neutral element for this semigroup, making  $\text{Hom}_{\mathbf{C}}(X; X)$  a monoid. We have seen earlier that neutral elements for semigroups are necessarily unique.

#### Categories

**Definition 13.9** (Category)

A *category*  $\mathbf{C}$  is a semicategory in which there is an identity morphism for every object.

**Example 13.10.** The semicategory of sets and functions described above in Example 13.5 is in fact a category. Given a set  $X$ , the identity morphism for this set is the identity function

$$\begin{aligned} \text{id}_X : X &\rightarrow X, \\ x &\mapsto x. \end{aligned} \quad (28)$$

Let us check that the conditions (26) and (27) are satisfied. Given a function  $f : W \rightarrow X$ , the function composition  $f \circ \text{id}_X$  is the same function as just  $f$  on its own:

$$(f \circ \text{id}_X)(x) = \text{id}_X(f(x)) = f(x). \quad (29)$$

Given a function  $g : X \rightarrow Y$ , we can show similarly that  $\text{id}_X \circ g = g$ .

**Definition 13.11** (Category of sets)

The category **Set** of sets is defined by:

1. *Objects*: all sets.
2. *Morphisms*: given sets  $X$  and  $Y$ , the hom-set  $\text{Hom}_{\text{Set}}(X; Y)$  is the set of all functions from  $X$  to  $Y$ .
3. *Composition*: the usual composition of functions.
4. *Identity morphisms*: given a set  $X$ , its identity morphism  $\text{id}_X$  is the identity function  $X \rightarrow X$ ,  $\text{id}_X(x) = x$ .

**Graded exercise E.1** (LinearMaps)

Morphisms between real vector spaces are called *linear maps*. Given real vector spaces  $U = \langle U, +_U, 0_U, \cdot_U \rangle$  and  $V = \langle V, +_V, 0_V, \cdot_V \rangle$ , a linear map  $f : U \rightarrow V$  is a function

$$f : U \rightarrow V \quad (30)$$

which satisfies the following conditions:

1.  $f(x +_U y) = f(x) +_V f(y) \quad \forall x, y \in U$ ;
2.  $f(\lambda \cdot_U x) = \lambda \cdot_V f(x) \quad \forall x \in U, \forall \lambda \in \mathbb{R}$ .

Your task in this exercise is to prove that the composition of linear maps is again a linear map. Concretely, suppose that you are given linear maps  $f : U \rightarrow V$  and  $g : V \rightarrow W$  and check that the composition of functions  $f \circ g : U \rightarrow W$  is again a linear map.

**Graded exercise E.2** (CategoryRealVect)

Show that real vector spaces and linear maps between them form a category. This means:

1. State what are the objects, the morphisms, and the composition operations (and check that the latter are well-defined).
2. Check that the associative law holds.
3. State what the identity morphisms are, and prove that they are neutral for composition.

**Graded exercise E.3** (PointedEuclideanSpaces)

In this exercise we will define a category **Euc**<sub>\*</sub> of “pointed Euclidean spaces” and your task is to check that it is in fact a category.

The objects of **Euc**<sub>\*</sub> are pairs  $\langle \mathbb{R}^n, x \rangle$  where  $x \in \mathbb{R}^n$  (and  $n \in \mathbb{N}$  varies).

A morphism  $f : \langle \mathbb{R}^n, x \rangle \rightarrow_{\text{Euc}_*} \langle \mathbb{R}^m, y \rangle$  is a smooth (infinitely differentiable) function  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  such that  $f(x) = y$ .

Composition is given by composition of smooth functions, and identity morphisms are identity functions.

**Graded exercise E.4** (Endofunctions)

In this exercise we will define a category **EndSet** of “endofunctions” and it is your task to check that it really is a category.

The objects of **EndSet** are pairs  $\langle A, \varphi \rangle$  where  $A$  is a set and  $\varphi : A \rightarrow A$  is a function.

A morphism  $f : \langle A, \varphi \rangle \rightarrow_{\text{EndSet}} \langle B, \psi \rangle$  is a function  $f : A \rightarrow_{\text{Set}} B$  with the property that  $f \circ \psi = \varphi \circ f$ .

Composition is given by composition of functions, and identity morphisms are identity functions.

**Graded exercise E.5** (HwkGSet)

Consider the definition below of the category **GSet**. Your task: check/justify that this does indeed define a category.

**Definition 13.12** (GC)

An object of **GSet** is a tuple

$$\langle Q, A, R \rangle, \quad (31)$$

where  $Q$  and  $A$  are sets, and  $R : Q \rightarrow_{\mathbf{Rel}} A$  is a relation.

A morphism  $\mathbf{r} : \langle Q_1, A_1, R_1 \rangle \rightarrow_{\mathbf{GSet}} \langle Q_2, A_2, R_2 \rangle$  is a pair of maps

$$\mathbf{r} = \langle r_{\flat}, r^{\sharp} \rangle, \quad (32)$$

$$r_{\flat} : Q_1 \leftarrow_{\mathbf{Set}} Q_2, \quad (33)$$

$$r^{\sharp} : A_1 \rightarrow_{\mathbf{Set}} A_2, \quad (34)$$

that satisfy the property

$$\forall q_2 : Q_2 \quad \forall a_1 : A_1 \quad r_{\flat}(q_2) R_1 a_1 \Rightarrow q_2 R_2 r^{\sharp}(a_1). \quad (35)$$

Morphism composition is defined component-wise

$$(\mathbf{r} \circ \mathbf{s})_{\flat} = s_{\flat} \circ r_{\flat}, \quad (36)$$

$$(\mathbf{r} \circ \mathbf{s})^{\sharp} = r^{\sharp} \circ s^{\sharp}. \quad (37)$$

The identity at  $\langle Q, A, R \rangle$  is  $\text{id}_{\langle Q, A, R \rangle} = \langle \text{id}_Q, \text{id}_A \rangle$ .

## Isomorphisms

What are identity morphisms good for? One thing we can do with them is define, for any category, the important notion of isomorphism. This concept describes a way of saying when two objects are “the same”, even if they are not equal.

**Definition 13.13**

Let  $\mathbf{C}$  be a category. A morphism  $f : X \rightarrow Y$  in  $\mathbf{C}$  is an *isomorphism* if there exists a morphism  $g : Y \rightarrow X$  in  $\mathbf{C}$  such that

$$f \circ g = \text{id}_X \quad (38)$$

and

$$g \circ f = \text{id}_Y. \quad (39)$$

**Remark 13.14.** Note that the above definition coincides, for the category **Set** of sets and functions, with Def. 4.19. We saw in Exercise 8 that an isomorphism in the category **Set** is the same thing as a bijective function.

## 13.4. Diagrams

When working with (semi)categories, it is typical to use diagrams that look like directed graphs, with nodes representing objects and directed arrows representing morphisms. Diagrams are usually used as a tool to speak and think about specific situations, where one is focusing on certain objects and morphisms of a given (semi)category. We typically don't draw every single object and morphism in the (semi)category, we just draw the ones that we want to refer to.

For example, we might draw a diagram as in Fig. 4 because we are considering the morphisms  $f$ ,  $g$ , and  $h$ . The diagram encodes their sources and targets, and is suggestive of how they may be composed. Composition corresponds to following paths in the diagram. Based on Fig. 4, we could build all the morphisms depicted in Fig. 5. By the associative law for semicategories, however, we know that  $(f \circ g) \circ h = f \circ (g \circ h)$ ; this morphism corresponds to the path along  $f$ ,  $g$ , and  $h$ .

### Commutative diagrams

Often we will be interested in knowing whether two given morphisms in a hom-set are *equal* or not. For instance, in Fig. 5 we have  $(f \circ g) \circ h = f \circ (g \circ h)$ . Or consider for example the situation in Fig. 6. It could be that the morphisms  $f \circ g$  and  $h$  are two distinct elements of  $\text{Hom}_C(X; Z)$ , or it could be that they are in fact equal,  $f \circ g = h$ . When the latter is the case, we encode this information compactly by drawing just the diagram in Fig. 7 and saying that it is *commutative*. This is a shorthand way of saying that both possible paths from  $X$  to  $Z$  in the diagram (namely via  $h$  or via  $f$  then  $g$ ) give rise to the same morphism.

#### Definition 13.15 (Commutative diagram)

A diagram in a (semi)category is commutative if, for any two objects  $X$  and  $Y$  in the diagram, all morphisms that arise from following paths in the diagram from  $X$  to  $Y$  are in fact equal.

As a further example, if we say that the diagram in Fig. 8 is commutative, then this means in particular that the morphisms  $f \circ g$ ,  $h \circ i$ , and  $j$  are all equal. By Def. 13.15 it also means for instance that all paths between  $Y$  and  $U$  give rise to the same morphism, but since there is only one path, namely via  $g$ , this doesn't give us any new information.

**Remark 13.16.** When drawing a diagram in a category (as opposed to just a semicategory), we usually do not draw the identity morphisms. By the definition of a category, we know the identity morphisms are there, and because they act neutrally for composition, they do not alter the computation of morphisms via following paths in a diagram.

Sometimes, however, it is more clear or simply needed to draw identity morphisms in a diagram. For instance, saying that the diagram in Fig. 9 is commutative is a way of saying that  $f \circ g = \text{id}_X$ .

#### Graded exercise E.6 (InventingCommDiagrams)

Come up with three different examples of a commutative diagram in the category of sets and functions. In each example, be sure to define clearly all of the functions involved.

### Graphical calculi

Later in the book we will get to know another kind of diagram: *string diagrams*. These are a different visual tool for reasoning “diagrammatically” about situations



Figure 4.

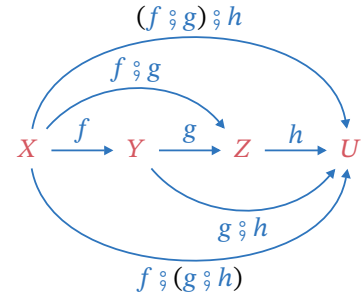


Figure 5.

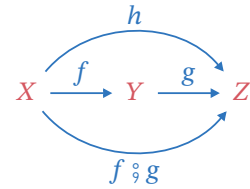


Figure 6.

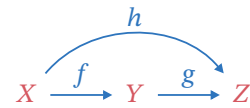


Figure 7.

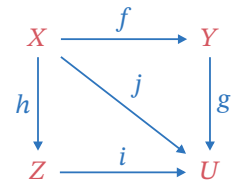


Figure 8.

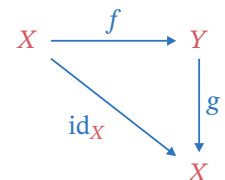


Figure 9.



in category theory. In fact, the string diagrams we will introduce are just one example in a whole zoo of visual tools, “graphical calculi”, that combine visual intuition with formal rigor. For now though, the word “diagram” will refer to the kind discussed above.

## 13.5. Semicategories vs directed graphs

One might now be led to ask: what is actually the difference between a semicategory and a directed graph?

With directed graphs we have nodes and directed edges, and with semicategories we have objects and morphisms. These are the same ingredients, apart from name differences.

A further essential ingredient in the definition of a semicategory is the composition operation: for any two morphisms where the target of one is the source of the other, we can compose them to obtain a further morphism.

One might thus say: a semicategory corresponds to a special kind of directed graph, where for any two adjacent directed edges there must exist a third edge corresponding to the “composite” of those edges. This is technically a true statement, however it does not emphasize the key point that, in category theory, we are often interested in comparing composite morphisms which might turn out to be equal, or not.

In Section 33.6, we will spell out an elegant and formally detailed way of thinking about the relationship between directed graphs and (semi)categories.

### 13.6. Generating categories from graphs

We can turn any graph into a (semi)category.

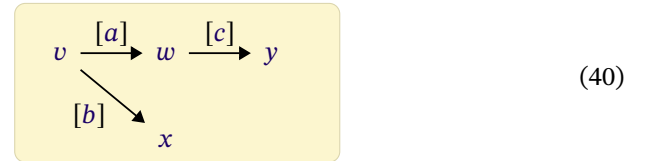
**Definition 13.17** (Semicategory generated by a graph)

Let  $\mathcal{G} = \langle \mathbf{V}, \mathbf{E}, \text{src}, \text{tgt} \rangle$  be a graph. The *free semicategory on  $\mathcal{G}$* , denoted  $\mathbf{SC}(\mathcal{G})$ , has as objects the vertices  $\mathbf{V}$  of  $\mathcal{G}$ , and given vertices  $v \in \mathbf{V}$  and  $w \in \mathbf{V}$ , morphisms  $\text{Hom}_{\mathbf{SC}(\mathcal{G})}(v; w)$  are the non-trivial paths from  $v$  to  $w$ . The composition of morphisms is given by concatenation of paths.

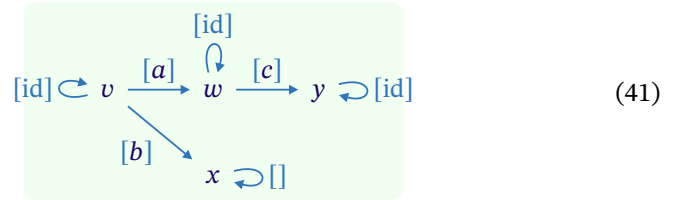
**Definition 13.18** (Category generated by a graph)

Given a graph  $\mathcal{G} = \langle \mathbf{V}, \mathbf{E}, \text{src}, \text{tgt} \rangle$ , the *free category on  $\mathcal{G}$* , denoted  $\mathbf{C}(\mathcal{G})$ , is defined analogously to  $\mathbf{SC}(\mathcal{G})$  but with the modification that  $\text{Hom}_{\mathbf{C}(\mathcal{G})}(v; w)$  is equal to *all* paths from  $v$  to  $w$ . Identity morphisms are the trivial paths.

For instance, consider the graph



The free category on this graph is given by

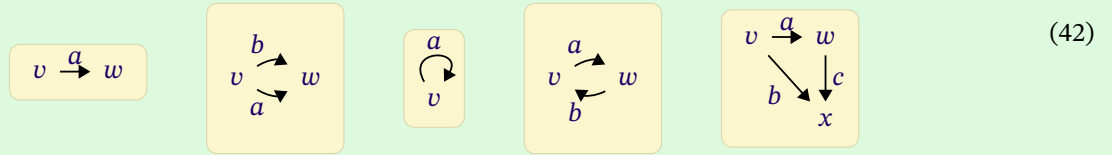


and has 8 morphisms (each vertex/object has identity morphisms,  $a, b, c$  give rise to three morphisms, and the composition of  $a, c$  gives rise to a morphism, omitted from the drawing).

Does Def. 13.18 define a category? We can check it ourselves. For it to define a category, unitality and associativity need to be satisfied. Given our definition of path, this is easy. The concatenation of paths is just list concatenation (which we already proved to be associative). Furthermore, a trivial path can be expressed via an empty list, which acts as an identity when composed to any other path.

**Graded exercise E.7** (HowManyMorphisms)

Consider the following five graphs. For each graph  $\mathcal{G}$ , how many morphisms in total are there in the associated category  $\mathbf{SC}(\mathcal{G})$ ?

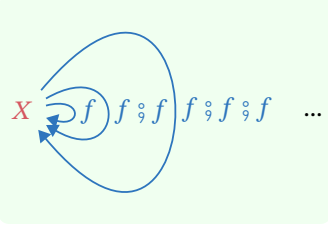


## 13.7. Semicategories

### Computable semicategories

We need to make a couple of modifications to the theory to make sure we have implementations of categories that we can easily manipulate.

We must take into account that a category is not a “static” data structure, like a graph, but rather a “dynamic” object that can grow. For example, consider a semicategory with one object  $X$ , and one morphism  $f : X \rightarrow X$ , which is not the identity. Then, even if the semicategory can be specified with a finite amount of data (one object and one morphism) there are actually an infinite amount of morphisms:



**Figure 10.:** Category with one object and one morphism.

$$\text{Hom}(X; X) = \{f, (f \circ f), (f \circ f \circ f), (f \circ f \circ f \circ f), \dots\}. \quad (43)$$

So far, we have not seen a mechanism that, given two objects, allows to produce another object, but we will see several later.

Our solution for software modeling is to distinguish among different “generations” of morphisms. If a morphism is given explicitly, like  $f$ , we say it is “of level 0”. The levels increase every time we use the composition operator.

Let  $\text{Hom}_i(X; Y)$  be the set of morphisms of *up to* level  $i \geq 0$ . Note that the morphisms at level  $i + 1$  include all the morphism of level  $i$ :

$$\frac{i \leq j}{\text{Hom}_i(X; Y) \subseteq \text{Hom}_j(X; Y)}. \quad (44)$$

The regular hom-set is recovered as the union of all the levels:

$$\text{Hom}(X; Y) = \bigcup_{i \geq 0} \text{Hom}_i(X; Y). \quad (45)$$

We need a rule to update the levels of morphisms. We say that if  $f$  has level  $m$  and  $g$  has level  $n$ , then the composition  $h = f \circ g$  is in level  $\max(m, n) + 1$ . We can update the signature of the composition rule as follows:

$$\frac{f \in \text{Hom}_m(X; Y) \quad g \in \text{Hom}_n(Y; Z)}{(f \circ g) \in \text{Hom}_{\max(m, n) + 1}(X; Z)}. \quad (46)$$

A morphism could be obtained in different ways, which could give different levels. We define “the level” of a morphism as the *minimum* level at which it appears.

Take again the example of the category with one morphism  $f : X \rightarrow X$  in Fig. 10. Table 13.1 shows how each generation looks like. For each row after the first, we look in all the previous rows and find the result of all compositions, and we note the new ones that we have not obtained before. For clarity, we write  $f^n$  for the composition of  $n$  copies of  $f$ .

There is one important property: if there is a finite number of morphisms at level 0, then there is a finite number of morphisms at each level. There might still be an overall infinite number of morphisms, but this assumption allows us to look at finite slices of the infinity.

**Table 13.1.:** Generations of morphisms

level	new morphisms at level
0	$f$
1	$f^2$
2	$f^3, f^4$
3	$f^5, f^6, f^7, f^8$
4	$f^9, f^{10}, f^{11}, f^{12}, f^{13}, f^{14}, f^{15}, f^{16}$

## Interface

Listing 75 below shows the Python interface we chose to model a category.

Compared to the previous exercises about sets, relations, posets, there is one relevant difference: we are not assuming anymore that the collections are finite sets. However, we still assume that the set of objects and the hom-sets are instances of `EnumerableSet`.

The class `SemiCategory` is a generic class with the following type variables:

1. The type variable `Ob` (object data) is the type of the objects;
2. The type variable `Mor` (morphism data) is the type of the morphisms.

The class `SemiCategory` has the following methods:

1. The method `objects()` returns an `EnumerableSet` of elements with type `Ob`. The optional parameter is the maximum level of the objects. If the value is `None`, the method must return all objects. So far we have not seen any way to create new objects from old; so far this argument will be unused.
2. The method `hom()`, given two objects, returns the `EnumerableSet` for the morphisms in the hom-set (of type `Mor`). The optional parameter is the maximum level of the morphisms. If the value is `None`, it must return all objects.
3. The method `identity()` returns the identity for the given object, if it exists; otherwise, it raises `InvalidValue`.
4. The method `compose()` represents the composition operation. Recall from (11) that the signature of  $\circ$  is

$$\circ_{X,Y,Z} : \text{Hom}_{\mathbf{C}}(X; Y) \times \text{Hom}_{\mathbf{C}}(Y; Z) \rightarrow \text{Hom}_{\mathbf{C}}(X; Z). \quad (47)$$

That is, we need to specify 3 objects, and then the 2 morphisms. In total, 5 parameters.

Listing 75: The `SemiCategory` interface.

```
class SemiCategory(Generic[Ob, Mor], ABC):
    @abstractmethod
    def objects(self, uptolevel: Optional[int] = None) -> EnumerableSet[Ob]:
        """Returns the set of objects up to given level (if given)."""

    @abstractmethod
    def hom(self, ob1: Ob, ob2: Ob, uptolevel: Optional[int] = None) -> EnumerableSet[Mor]:
        """Returns the homset of up to given level (if given)."""

    @abstractmethod
    def compose(self, ob1: Ob, ob2: Ob, ob3: Ob, m1: Mor, m2: Mor) -> Mor:
        """Equivalent to the ; operator."""
    ...

    @abstractmethod
    def identity(self, ob: Ob) -> Mor:
        """Identity for the object. Raises I.InvalidValue if there is no identity for ob."""
```

```
objects:
  X:
    obdata: 1
    identity:
      mordata: ''
morphisms:
  f:
    mordata: df
    source: X
    target: X
equations:
  eq1: f;f;f=f
```

Figure 11.: cat\_example1



Figure 12.: Category with one object and one morphism and equation.

Representation

We now introduce a file format for representing categories. A dictionary representing a category (Fig. 11) has three fields:

- 1. **objects** describes the objects;
- 2. **morphisms** describes the morphisms;
- 3. **equations** describes known equations between morphisms.

Each object and morphism has a string label. In this example, there is only 1 object called x and one morphism called f.

Objects and morphisms have data attached to them. For an object, it is given by the field **obdata**. For a morphism, it is given by the field **mordata**.

Each object has also a field **identity**. If the value is none, it means that there is no identity for that object. Otherwise, there is an identity called id\_OBJECT. The field **mordata** gives the additional data for the identity morphism.

In this example, the object x has the integer 1 associated to it, the morphism f has the data "py" associated to it, and the morphism id\_x has the data "" (empty string) associated to it.

For now, they are just objects without semantics; later we introduce a way to interpret the data.

The field **equations** describes the assumptions about morphisms composition. Each equation content is a string, consisting of two substrings separated by equality. Each substring is the concatenation of morphism names with ; as a separator. In this example, the string f;f;f=f is to be interpreted as the equation

$f \circ f \circ f = f.$  (48)

Therefore, in this example we are describing a category with 1 object and 3 morphisms:  $id_x$ ,  $f$ , and  $f \circ f \circ f$ . Note that  $f \circ f \circ f \circ f$  does not appear in this list because it is equivalent to  $f$  given (48).

In the exercises, when you are given a category description like in Fig. 11, you will not be given *all* morphisms *explicitly*, but you need to generate them, based on the composition rules.

More examples

A category with zero objects.



<pre>objects: {}</pre>	<pre>morphisms: {}</pre>	<pre>equations: {}</pre>
------------------------	--------------------------	--------------------------

A category with one object, no morphisms.



<pre>objects:   X:     obdata: 1     identity:</pre>	<pre>morphisms: {}</pre>	<pre>equations: {}</pre>
------------------------------------------------------	--------------------------	--------------------------

A category with one object and its identity.

```
objects:
X:
  obdata: 1
  identity:
  mordata: ''
```

```
morphisms: {}
```

```
equations: {}
```

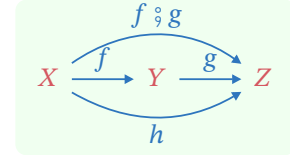
$$X \xrightarrow{\text{id}_X} X$$

A category with three objects, three explicit morphisms  $f, g, h$ , and four morphisms in total:  $f, g, h, f \circ g$ .

```
objects:
X:
  obdata: 1
  identity:
  mordata: ''
Y:
  obdata: 2
  identity:
  mordata: ''
Z:
  obdata: 3
  identity:
  mordata: ''
```

```
morphisms:
f:
  mordata: df
  source: X
  target: Y
g:
  mordata: dg
  source: Y
  target: Z
h:
  mordata: dh
  source: X
  target: Z
```

```
equations: {}
```

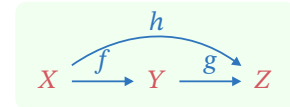


A category with three objects, three explicit morphisms  $f, g, h$  and the equation  $f \circ g = h$ . There are now only 3 morphisms in total:  $f, g, h$ .

```
objects:
X:
  obdata: 1
  identity:
  mordata: ''
Y:
  obdata: 2
  identity:
  mordata: ''
Z:
  obdata: 3
  identity:
  mordata: ''
```

```
morphisms:
f:
  mordata: df
  source: X
  target: Y
g:
  mordata: dg
  source: Y
  target: Z
h:
  mordata: dh
  source: X
  target: Z
```

```
equations:
eq1: f;g=h
```



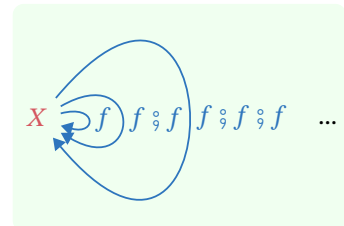
A category with one object, no identity, but a morphism. This is the first case in which there is an infinity of morphisms, starting with

$f, f \circ f, f \circ f \circ f, \dots$  (49)

```
objects:
X:
  obdata: 1
  identity:
```

```
morphisms:
f:
  mordata: df
  source: X
  target: X
```

```
equations: {}
```

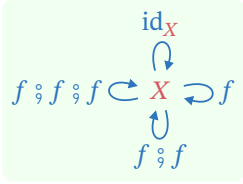




A category with one object, the identity, one morphism and one equation  $f \circ f \circ f \circ f = f$ . There are 4 morphisms:

$$\text{id}_X, \quad f, \quad f \circ f, \quad f \circ f \circ f.$$

It's similar to counting in base 4.



objects:

```
X:
  obdata: 1
  identity:
    mordata: ''
```

morphisms:

```
f:
  mordata: df
  source: X
  target: X
```

equations:

```
eq1: f;f;f;f=id_X
```

## Reading categories

In the file format that we have introduced, you should treat the `obdata` and `mordata` as opaque objects that are to be interpreted by two setoids that will be passed to you. The interface that you need to implement is in Listing 76. The function `load()` has the following parameters:

1. `h` is the usual `IOHelper`;
2. `data` is the Python data structure corresponding to the YAML.
3. `ObData` is a setoid that is in charge of interpreting the `obdata`;
4. `MorData` is a setoid that is in charge of interpreting the `mordata`;
5. `compose` is the compose operation that will allow you to compose the morphisms.

What you need to return is a semicategory with object type `RichObject` (Listing 77) and morphism type `RichMorphism` (Listing 78). These are just wrappers that group together the label and the object data.

Listing 76: The `SemiCategoryRepresentation` interface.

```
class SemiCategoryRepresentation(ABC):
    @abstractmethod
    def load(
        self,
        h: IOHelper,
        data: FiniteSemiCategory_desc,
        ObData: Setoid[OD],
        MorData: Setoid[MD],
        compose: Callable[[OD, OD, OD, MD, MD], MD],
    ) -> SemiCategory[RichObject[OD], RichMorphism[MD]]:
        pass
```

Listing 77: The `RichObject` interface.

```
@dataclass(frozen=True)
class RichObject(Generic[OD]):
    label: str
    obdata: OD
```

Listing 78: The `RichMorphism` interface.

```
@dataclass(frozen=True)
class RichMorphism(Generic[MD]):
    label: str
    mordata: MD
```

For example, in the case of Fig. 11, you would return the object

```
RichObject("X", 1)
```

and the morphism

```
RichObject("f", "df")
```

To give you a more concrete idea of how the method will be called during the unit tests, here's an example usage. We set `ObData` and `MorData` to be the setoids of integers (Listing 79) and strings (Listing 80). The `compose()` function is just concatenation of strings.

```

ObData = IntegerSetoid()
MorData = StringSetoid()
def compose(od1: int, od2: int, od3: int, md1: str, md2: str) -> str:
    return md1 + md2
yourcode: SemiCategoryRepresentation = ...
data = ... # one of the examples
h = ...
sc = yourcode.load(
    h,
    data,
    ObData,
    MorData,
    compose
)

```

Once the semicategory is created, it will be queried. For example, we will check if it has the correct objects and morphisms.

```

# for the first example, we expect this object to exist
obx = GenericObject("x", 1)
objects = sc.objects()
assert objects.contains(obx)
# ask for the homset of X
homset = sc.hom(obx, obx)

f = GenericMorphism('f', 'df')
# let's ask for all morphisms in the homset until we find f
for e in homset.elements():
    if homset.equal(morphism, f):
        break
else:
    raise ValueError("cannot find morphism f ")

```

Listing 79: The `IntegerSetoid` interface.

```

class IntegerSetoid(Setoid[int]):
    """Represents the setoids of integers."""

    def contains(self, x: int) -> bool:
        return isinstance(x, int)

    def save(self, _: IOHelper, x: int) -> int:
        if not isinstance(x, int):
            raise InvalidValue()
        return x

    def load(self, _: IOHelper, o: int) -> int:
        if not isinstance(o, int):
            raise InvalidFormat()
        return o

```

Listing 80: The `StringSetoid` interface.

```

class StringSetoid(Setoid[str]):
    """Represents the setoids of strings."""

    def contains(self, x: str) -> bool:
        return isinstance(x, str)

    def save(self, _: IOHelper, x: str) -> str:
        if not isinstance(x, str):
            raise InvalidValue()
        return x

    def load(self, _: IOHelper, o: str) -> str:
        if not isinstance(o, str):
            raise InvalidFormat()
        return o

```

### Code exercise E.1 (TestSemiCategoryRepresentation)

Write code to load a semicategory from its representation. Please read in details the rules below.

Implement the interface in Listing 76.

Test using:

```
act4e-check TestSemiCategoryRepresentation
```

Some rules to keep in mind:

1. When you compose two `RichMorphisms` with labels "label1" and "label2", their

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

new label should be "label1;label2". Their `mordata` will be given by the `compose()` function.

2. Two morphisms are equal if any of the two conditions are met:
  - a) Their labels are related by an equation.
  - b) Their `mordata` is equal based on what the setoid `MorData` thinks.
3. Always return the shortest name for the morphisms. For example, if you can derive  $f;g$  and the equation " $f;g=h$ " holds, you have to return the morphisms  $f$ ,  $g$ , and  $h$ , because  $h$  is shorter than  $f;g$  even though they are equivalent.

Some simplifications you can use:

1. **Simplification 1:** You have to return an `EnumerableSet` from `hom()`. The unit tests will not test the functions `load()`, `save()`, `contains()` of this enumerable set, so you don't strictly need to implement them. However, it is important you implement `equals()` and `elements()`.
2. **Simplification 2:** If the equation is of the form " $\text{first}=\text{second}$ ", it is guaranteed that you only need to substitute the second term in place of the first to guarantee that you find all equalities, and you never need to substitute in the other directions. For example, if the equation is " $a;b;c=d;e$ ", you will have to substitute the sequence  $[d, e]$  to  $[a, b, c]$ ; that is, simplify the list of morphisms  $[a, a, b, c, a]$  to  $[a, d, e, a]$ . But you will never need to substitute  $[a, b, c]$  to  $[d, e]$ . As a result, you are guaranteed to have a *confluent* system, in the sense that if you run all the substitutions you will eventually converge.



## 14. Categories and structures

In this chapter we will take a tour of various categories whose objects are sets equipped with additional structures, and whose morphisms are maps between those sets that preserve the given structures.

<b>14.1 Categories of sets and functions</b>	<b>232</b>
<b>14.2 Categories of relations</b>	<b>233</b>
<b>14.3 Categories of semigroups, monoids, groups</b>	<b>234</b>
<b>14.4 Categories from linear algebra</b>	<b>235</b>
<b>14.5 Categories of posets</b>	<b>236</b>
<b>14.6 Sets with data</b>	<b>237</b>
<b>14.7 Categories of graphs</b>	<b>238</b>
<b>14.8 Preorders as categories</b>	<b>239</b>
<b>14.9 Monoids as categories</b>	<b>240</b>

### 14.1. Categories of sets and functions

We have already introduced in Example 13.10 the category of all sets and functions (it is a very large category). A close relative of this category is the category **FinSet**, where we only consider *finite* sets as objects, but otherwise, everything is the same as in the category **Set**.

Other categories of sets and functions can be obtained by restricting what type of functions we consider. For example, there is a category **InjSet** where the objects are all sets and where morphisms are injective functions. Similarly, there is a category **Surj** of surjective functions, and also a category **Bij** of bijective functions.

**Exercise 26.** Spell out a definition of the category **InjSet** of injective functions, and check that it is indeed a category. In particular:

1. Specify what the composition operations are and check if the composition of two composable injective functions is again injective;
2. Specify what the identity morphisms are and check that they are indeed injective functions;
3. Argue why the associativity condition is satisfied.

*See solution on page 299.*

## 14.2. Categories of relations

Recall that a (binary) relation from a set  $\mathbf{A}$  to a  $\mathbf{B}$  is a subset  $R \subseteq \mathbf{A} \times \mathbf{B}$ . We have already seen that relations can be composed, so it is natural now to think of a relation  $R \subseteq \mathbf{A} \times \mathbf{B}$  as a *morphism* from  $\mathbf{A}$  to  $\mathbf{B}$ .

### Definition 14.1 (Category $\mathbf{Rel}$ )

The category  $\mathbf{Rel}$  of relations is defined by:

1. *Objects*: all sets.
2. *Morphisms*: for sets  $X, Y$ ,  $\mathbf{Hom}_{\mathbf{Rel}}(X; Y)$  is the set of all relations  $R \subseteq X \times Y$ .
3. *Composition*: for relations  $R : X \rightarrow Y$  and  $S : Y \rightarrow Z$ , their composition is

$$R \circ S := \{\langle x, z \rangle \in X \times Z \mid \exists y \in Y : (x R y) \wedge (y S z)\}. \quad (1)$$

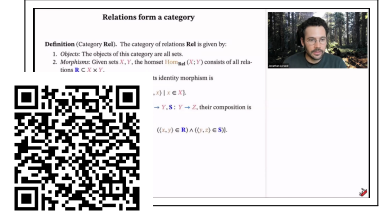
4. *Identity morphisms*: for a set  $X$ , its identity morphism is

$$\text{id}_X := \{\langle x, y \rangle \in X \times X \mid x = y\}. \quad (2)$$

### Graded exercise E.8 (IsosInRel)

Prove that isomorphisms in the category  $\mathbf{Rel}$  are precisely those relations which correspond to bijective functions.

Watch: *Category of relations* (5 minutes).



### 14.3. Categories of semigroups, monoids, groups

**Definition 14.2** (Category of semigroups)

The category **SGrp** of semigroups is:

1. *Objects*: all semigroups.
2. *Morphisms*: for semigroups  $X$  and  $Y$ , the hom-set  $\text{Hom}_{\text{SGrp}}(X; Y)$  is the set of all semigroup morphisms from  $X$  to  $Y$ .
3. *Composition*: composition of semigroup morphisms (composition of the underlying functions).
4. *Identity morphisms*: for a semigroup  $X = \langle S, \cdot_X \rangle$ , its identity morphism  $\text{id}_X$  is given by the identity function  $S \rightarrow S$ .

**Graded exercise E.9** (CategorySemigroups)

Check explicitly that the above definition does indeed define a category. Is the composition of composable morphisms again a morphism? Does the associative law hold? Are the conditions for identity morphisms satisfied?

**Remark 14.3.** An isomorphism (in the sense of category theory) in the category **SGrp** of semigroups is the same thing as an isomorphism of semigroups (in the sense of algebra).

**Definition 14.4** (Category of monoids)

The category **Mon** of monoids is:

1. *Objects*: all monoids.
2. *Morphisms*: for monoids  $X$  and  $Y$ , the hom-set  $\text{Hom}_{\text{Mon}}(X; Y)$  is the set of all monoid morphisms from  $X$  to  $Y$ .
3. *Composition*: composition of monoid morphisms.
4. *Identity morphisms*: for a monoid  $X = \langle M, \cdot, \text{id} \rangle$ , its identity morphism  $\text{id}_X$  is given by the identity function  $M \rightarrow M$ .

**Definition 14.5** (Category of groups)

The category **Grp** of groups is:

1. *Objects*: all groups.
2. *Morphisms*: for groups  $X, Y$ , the hom-set  $\text{Hom}_{\text{Grp}}(X; Y)$  is the set of all group morphisms from  $X$  to  $Y$ .
3. *Composition*: composition of group morphisms.
4. *Identity morphisms*: for a group  $X$ , its identity morphism  $\text{id}_X$  is given by the identity function  $G \rightarrow G$ .

## 14.4. Categories from linear algebra

### Definition 14.6 (Category of real matrices)

The category  $\mathbf{Mat}_{\mathbb{R}}$  of real matrices is:

1. *Objects*: natural numbers  $\mathbb{N}$ .
2. *Morphisms*: for any  $m, n \in \mathbb{N}$ ,  $\mathbf{Hom}_{\mathbf{Mat}_{\mathbb{R}}}(m; n)$  is the set of  $n \times m$  real matrices.
3. *Composition*: matrix multiplication.
4. *Identity morphisms*: identity matrices.

### Definition 14.7 (Category of real vector spaces)

The category  $\mathbf{Vect}_{\mathbb{R}}$  of real vector spaces is:

1. *Objects*: all real vector spaces.
2. *Morphisms*:  $\mathbf{Hom}_{\mathbf{Vect}_{\mathbb{R}}}(X; Y)$  is the set of real linear maps  $X \rightarrow Y$ .
3. *Composition*: the usual composition of linear maps.
4. *Identity morphisms*: for any real vector space  $X = \langle X, +, \cdot \rangle$ , the identity morphism  $\text{id}_X$  is given by the identity function  $X \rightarrow X$ .



## 14.5. Categories of posets

### Definition 14.8 (Category **Pos**)

The category **Pos** is:

1. *Objects*: all posets.
2. *Morphisms*: for posets  $X = \langle P, \leq_X \rangle$  and  $Y = \langle P, \leq_Y \rangle$ ,  $\text{Hom}_{\mathbf{Pos}}(X; Y)$  is the set of all monotone maps from  $X$  to  $Y$ .
3. *Composition*: composition of monotone maps.
4. *Identity morphisms*: for a poset  $X = \langle P, \leq \rangle$ , its identity morphism  $\text{id}_X$  is given by the identity function  $P \rightarrow P$ .

Occasionally we will write  $f : X \rightarrow_{\mathbf{Pos}} Y$  to emphasize that a monotone map between posets is a morphism in **Pos**.

## 14.6. Sets with data

There are various simple constructions where we can build categories whose objects are not just sets, but sets together with some extra data. Morphisms are then functions which are compatible with the extra data. Below we give a few examples and we encourage the reader to imagine further variations.

### Definition 14.9 (Pointed sets)

The category  $\mathbf{Set}_*$  of pointed sets is:

1. *Objects*: pairs  $\langle \mathbf{A}, x \rangle$  where  $\mathbf{A}$  is a set and  $x \in \mathbf{A}$  is an element of  $\mathbf{A}$ .
2. *Morphisms*: a morphism  $f : \langle \mathbf{A}, x \rangle \rightarrow_{\mathbf{Set}_*} \langle \mathbf{B}, y \rangle$  is a function  $f : \mathbf{A} \rightarrow_{\mathbf{Set}} \mathbf{B}$  such that  $f(x) = y$ .
3. *Composition*: the usual composition of functions.
4. *Identity morphisms*: identity functions.

**Exercise 27.** Prove that Def. 14.9 really is a category.

*See solution on page 299.*

### Definition 14.10 (Endofunctions)

The category  $\mathbf{EndSet}$  of endofunctions is:

1. *Objects*: pairs  $\langle \mathbf{A}, \varphi \rangle$  where  $\mathbf{A}$  is a set and  $\varphi : \mathbf{A} \rightarrow \mathbf{A}$  is a function.
2. *Morphisms*: a morphism  $f : \langle \mathbf{A}, \varphi \rangle \rightarrow_{\mathbf{EndSet}} \langle \mathbf{B}, \psi \rangle$  is a function  $f : \mathbf{A} \rightarrow_{\mathbf{Set}} \mathbf{B}$  with the property that  $f \circ \varphi = \psi \circ f$ .
3. *Composition*: the usual composition of functions.
4. *Identity morphisms*: identity functions.

**Exercise 28.** Prove that Def. 14.10 is indeed a category.

*See solution on page 299.*

### Definition 14.11 (Equivalence relations)

The category  $\mathbf{EquivRel}$  of equivalence relations is:

1. *Objects*: pairs  $\langle \mathbf{A}, \sim_{\mathbf{A}} \rangle$  where  $\mathbf{A}$  is a set and  $\sim_{\mathbf{A}} : \mathbf{A} \rightarrow_{\mathbf{Rel}} \mathbf{A}$  is an equivalence relation.
2. *Morphisms*: a morphism  $f : \langle \mathbf{A}, \sim_{\mathbf{A}} \rangle \rightarrow_{\mathbf{EquivRel}} \langle \mathbf{B}, \sim_{\mathbf{B}} \rangle$  is a function  $f : \mathbf{A} \rightarrow_{\mathbf{Set}} \mathbf{B}$  such that

$$x \sim_{\mathbf{A}} y \implies f(x) \sim_{\mathbf{B}} f(y). \quad (3)$$

3. *Composition*: the usual composition of functions.
4. *Identity morphisms*: identity functions.

**Exercise 29.** We can visualize an equivalence relation on a set  $\mathbf{A}$  as a partition of  $\mathbf{A}$ . Can you visualize the condition (3) in terms of sets and partitions of them?

*See solution on page 299.*

**Remark 14.12.** The above example with equivalence relations is very similar to the category of posets and monotone maps; we are simply considering equivalence relations instead of relations which are partial orders. The category of posets, and as well as most of our other examples of categories of algebraic structures (semigroups, monoids, groups, etc.), can all be thought of as categories built from “sets with extra data”.

## 14.7. Categories of graphs

Before introducing the category of graphs **Grph**, we show how we can compose graph homomorphisms. Given graph homomorphisms  $F : \mathcal{G}_1 \rightarrow \mathcal{G}_2$  and  $G : \mathcal{G}_2 \rightarrow \mathcal{G}_3$ , their composition  $F \circ G : \mathcal{G}_1 \rightarrow \mathcal{G}_3$  acts on vertices with  $F \circ G$ , and on edges with  $F \rightarrow \circ G \rightarrow$ .

**Lemma 14.13.** The composition of graph homomorphisms is a graph homomorphism.

*Proof.* Consider graphs  $\mathcal{G}_1 = \langle \mathbf{V}_1, \mathbf{E}_1, \text{src}_1, \text{tgt}_1 \rangle$ ,  $\mathcal{G}_2 = \langle \mathbf{V}_2, \mathbf{E}_2, \text{src}_2, \text{tgt}_2 \rangle$ , and  $\mathcal{G}_3 = \langle \mathbf{V}_3, \mathbf{E}_3, \text{src}_3, \text{tgt}_3 \rangle$  and graph homomorphisms  $F : \mathcal{G}_1 \rightarrow \mathcal{G}_2$  and  $G : \mathcal{G}_2 \rightarrow \mathcal{G}_3$ . The composition of the graph homomorphisms is  $F \circ G : \mathcal{G}_1 \rightarrow \mathcal{G}_3$ , and acts on vertices with  $F \circ G$ , and on edges with  $F \rightarrow \circ G \rightarrow$ . Then,  $F \circ G$  is a graph homomorphism, since we have:

$$\begin{aligned} (F \rightarrow \circ G \rightarrow) \circ \text{src}_3 &= F \rightarrow \circ G \rightarrow \circ \text{src}_3 && \text{associativity in } \mathbf{Set} \\ &= F \rightarrow \circ \text{src}_2 \circ G. && G \text{ is a graph homom.} \\ &= \text{src}_1 \circ F \circ G. && F \text{ is a graph homom.} \\ &= \text{src}_1 \circ (F \circ G) && \text{associativity in } \mathbf{Set}, \end{aligned} \tag{4}$$

and

$$\begin{aligned} (F \rightarrow \circ G \rightarrow) \circ \text{tgt}_3 &= F \rightarrow \circ G \rightarrow \circ \text{tgt}_3 && \text{associativity in } \mathbf{Set} \\ &= F \rightarrow \circ \text{tgt}_2 \circ G. && G \text{ is a graph homom.} \\ &= \text{tgt}_1 \circ F \circ G. && F \text{ is a graph homom.} \\ &= \text{tgt}_1 \circ (F \circ G) && \text{associativity in } \mathbf{Set}. \end{aligned} \tag{5}$$

These are precisely the conditions for a graph homomorphism  $\mathcal{G}_1 \rightarrow \mathcal{G}_3$   $\square$

### Definition 14.14 (Category **Grph**)

The category **Grph** is defined by:

1. *Objects*: all graphs.
2. *Morphisms*: for graphs  $X$  and  $Y$ ,  $\text{Hom}_{\mathbf{Grph}}(X; Y)$  is the set of graph homomorphisms (Def. 12.4) from  $X$  to  $Y$ .
3. *Composition*: composition of graph homomorphisms.
4. *Identity morphisms*: for a graph  $X = \langle \mathbf{V}, \mathbf{E}, \text{src}, \text{tgt} \rangle$ , its identity morphism  $\text{id}_X$  is the graph homomorphism  $F$  with  $F \circ = \text{id}_{\mathbf{V}}$  and  $F \rightarrow = \text{id}_{\mathbf{E}}$ .

**Lemma 14.15.** **Grph** is indeed a category.

*Proof.* First, from Lemma 14.13 we know that the composition of graph homomorphisms is a graph homomorphism. We check unitality and associativity. We start with unitality. Consider graphs  $\mathcal{G}_1, \mathcal{G}_2$  and a graph homomorphism  $F : \mathcal{G}_1 \rightarrow \mathcal{G}_2$ . For  $\text{id}_{\mathcal{G}_1} \circ F$  we have, for every  $a \in \mathbf{E}_1$  that  $\text{id} \circ F \circ = F \circ$  and  $\text{id} \circ F \rightarrow = F \rightarrow$ . Similarly, for  $F \circ \text{id}_{\mathcal{G}_2}$  we have, for every  $a \in \mathbf{E}_2$  that  $F \circ \text{id} = F \circ$  and  $F \rightarrow \circ \text{id} = F \rightarrow$ . (In short, unitality follows from the unitality in **Set**). Similarly, associativity follows from associativity in **Set**.  $\square$

## 14.8. Preorders as categories

**Definition 14.16** (Categorification of a preorder)

Given a preorder  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$ , its *categorification*  $\mathbf{C}(\mathbf{P})$  is a category with

1. *Objects*: the elements of  $\mathbf{P}$ ;
2. *Morphisms*: given  $X, Y \in \mathbf{P}$ , we define the homset  $\text{Hom}_{\mathbf{C}(\mathbf{P})}(X; Y)$  to be

$$\text{Hom}_{\mathbf{C}(\mathbf{P})}(X; Y) = \begin{cases} \mathbf{1} & \text{if } X \leq_{\mathbf{P}} Y \\ \emptyset & \text{else.} \end{cases} \quad (6)$$

3. *Composition*: should be given by functions of the type

$$\text{Hom}_{\mathbf{C}(\mathbf{P})}(X; Y) \times \text{Hom}_{\mathbf{C}(\mathbf{P})}(Y; Z) \rightarrow \text{Hom}_{\mathbf{C}(\mathbf{P})}(X; Z). \quad (7)$$

When either of the factors in the source set are the empty set, then there is a unique function of the desired type. When both factors are equal to the set  $\mathbf{1}$ , then thanks to the *transitivity* of the preorder  $\leq_{\mathbf{P}}$  the target set  $\text{Hom}_{\mathbf{C}(\mathbf{P})}(X; Z)$  must also be the set  $\mathbf{1}$ , and there is a unique function of the type  $\mathbf{1} \times \mathbf{1} \rightarrow \mathbf{1}$ .

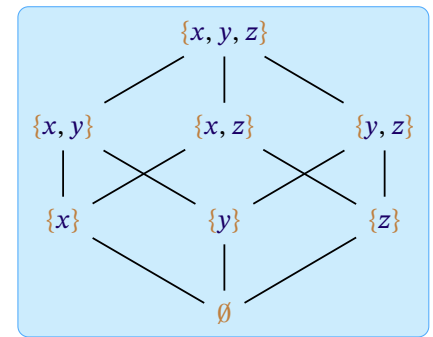
4. *Identities*: given any  $X \in \text{Ob}_{\mathbf{C}(\mathbf{P})}$ , we always have  $X \leq_{\mathbf{P}} X$ , by the *reflexivity* of the preorder  $\leq_{\mathbf{P}}$ . Hence  $\text{Hom}_{\mathbf{C}(\mathbf{P})}(X; X) = \mathbf{1}$  always. We define the single element of  $\text{Hom}_{\mathbf{C}(\mathbf{P})}(X; X) = \mathbf{1}$  to be the identity morphism of  $X$ .

**Remark 14.17.** A *thin* category is one in which there is at most one morphism in any homset. Categorifications of preorders are examples of thin categories. Conversely, every thin category can be interpreted as defining a preorder.

**Remark 14.18.** If we consider a preorder which is actually a poset, then its categorification is an example of a *skeletal* category. These are categories where, if any two objects are isomorphic, then they are necessarily equal.

**Example 14.19.** We revisit Def. 9.12, in which we had a poset  $\mathbf{P}$  on  $\text{Pow}\{a, b, c\}$  with order given by inclusion (Fig. 1). Its categorification  $\mathbf{C}(\mathbf{P})$  is a category, with  $\text{Ob}_{\mathbf{C}(\mathbf{P})} = \text{Pow}(\{a, b, c\})$ , and morphisms given by the inclusions. Note that we omit self-arrows for the identity morphisms, taking these to be tacitly implied. Composition is given by the transitivity law of posets. For instance, since  $\{a\} \subseteq \{a, b\}$  and  $\{a, b\} \subseteq \{a, b, c\}$ , we can say that  $\{a\} \subseteq \{a, b, c\}$ .

$\text{Pow}\{x, y, z\}$



**Figure 1.:** Power set  $\text{Pow}\{a, b, c\}$  as a poset.

## 14.9. Monoids as categories

**Definition 14.20** (Categorification of a monoid)

Given a monoid  $\mathbf{M} = \langle \mathbf{M}, \cdot, \text{id} \rangle$ , its *categorification*  $\mathbf{C}(\mathbf{M})$  is the category with

1. *Objects*: a single object, which we denote by  $\star$  (this is arbitrarily chosen);
2. *Morphisms*:  $\text{Hom}_{\mathbf{C}(\mathbf{M})}(\star, \star) = \mathbf{M}$ ;
3. *Composition*: is defined by the composition operation  $\cdot$  of the monoid;
4. *Identity morphism*: is defined by the neutral element  $\text{id}$  of the monoid.



# 15. Modeling with categories

In this chapter we provide some examples of modeling real-world phenomena using categories.

15.1 Mobility . . . . .	242
15.2 Trekking in the Swiss Mountains	244
15.3 Currency categories . . . . .	246
15.4 Resources dependencies . . . . .	249
15.5 Procedures . . . . .	253
15.6 Software dependencies . . . . .	256
15.7 Code exercises - Currency categories . . . . .	257

*Fondue* is a Swiss dish consisting of melted cheese, served in a *caquelon* (communal pot) over a *réchaud* (portable stove). It is best enjoyed with boiled potatoes, bread, and pickled vegetables.

## 15.1. Mobility

For a specific mode of transportation, say a car, we can define a graph

$$\mathcal{G}_c = \langle \mathbf{V}_c, \mathbf{E}_c, \text{src}_c, \text{tgt}_c \rangle, \quad (1)$$

where  $\mathbf{V}_c$  represents geographical locations which the car can reach and  $\mathbf{E}_c$  represents the paths it can take, such as roads. Similarly, we consider a graph  $\mathcal{G}_s = \langle \mathbf{V}_s, \mathbf{E}_s, \text{src}_s, \text{tgt}_s \rangle$ , representing the subway system of a city, with stations  $\mathbf{V}_s$  and subway lines going through paths  $\mathbf{E}_s$ , and a graph  $\mathcal{G}_b = \langle \mathbf{V}_b, \mathbf{E}_b, \text{src}_b, \text{tgt}_b \rangle$ , representing onboarding and off boarding at airports. In the following, we want to express intermodality: the phenomenon that someone might travel to a certain intermediate location in a car and then take the subway to reach their final destination. By considering the graph  $\mathcal{G} = (\mathbf{V}, \mathbf{E}, \text{src}, \text{tgt})$  with  $\mathbf{V} = \mathbf{V}_c \cup \mathbf{V}_s \cup \mathbf{V}_b$  and  $\mathbf{E} = \mathbf{E}_c \cup \mathbf{E}_s \cup \mathbf{E}_b$ , we obtain the desired intermodality graph. Graph  $\mathcal{G}$  can be seen as a new category, with objects  $\mathbf{V}$  and morphisms  $\mathbf{E}$ .

**Example 15.1.** Consider the **Car** category, describing your road trip through Italy and Switzerland, with

$$\mathbf{V}_c = \{\text{FCO}_c, \text{Florence}, \text{Bologna}, \text{MPX}_c, \text{Gotthard}, \text{ZRH}_c\}, \quad (2)$$

and arrows as in Fig. 1. The nodes represent typical touristic road-trip checkpoints in Italy and Switzerland and the arrows represent highways connecting them.

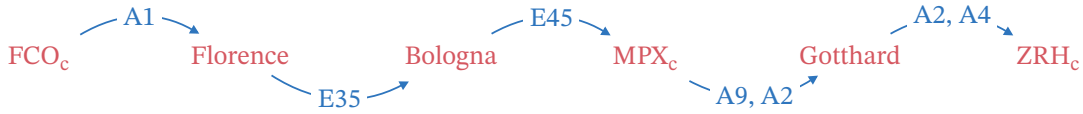


Figure 1.: The **Car** category.

Furthermore, consider the **Flight** category with

$$\mathbf{V}_f = \{\text{FCO}_f, \text{LIN}, \text{MPX}_f, \text{ZRH}_f\} \quad (3)$$

and arrows as in Fig. 2. The nodes represent airports in Italy and Switzerland and the arrows represent connections, offered by specific flight companies.

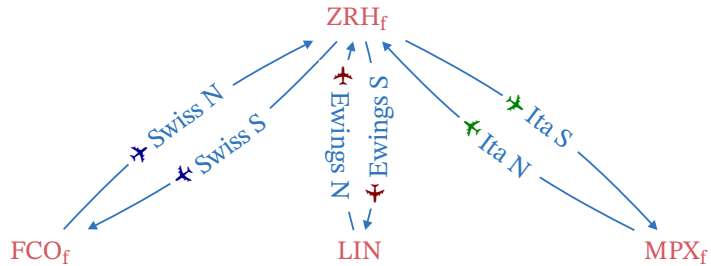


Figure 2.: The **Flight** category.

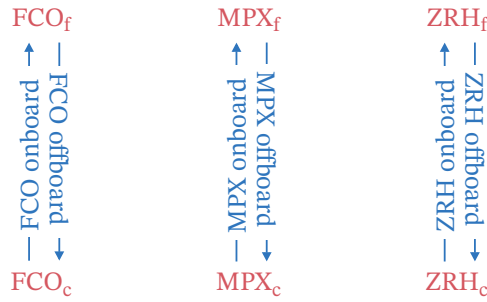
We then consider the **Board** category, with nodes

$$\mathbf{V}_b = \{\text{FCO}_f, \text{FCO}_c, \text{MPX}_f, \text{MPX}_c, \text{ZRH}_f, \text{ZRH}_c\} \quad (4)$$

and arrows as in Fig. 3. Nodes represent airports and airport parking lots, and arrows represent the onboarding and off boarding paths we have to walk to get from the parking lot to the airport and vice-versa.

The combination of the three, which we call the *intermodal graph*, can be represented as a graph, in which we use dashed arrows for intermodal morphisms,



Figure 3.: The **Board** category.

arising from composition of morphisms involving multiple modes (Fig. 4). Imagine that you are in the parking lot of **ZRH** airport and you want to reach **Florence**. From there, you can onboard to a **Swiss** flight to **FCO\_f**, will then offboard reaching the parking lot **FCO\_c**, and drive on highway **A1** reaching **Florence**. This is intermodality.

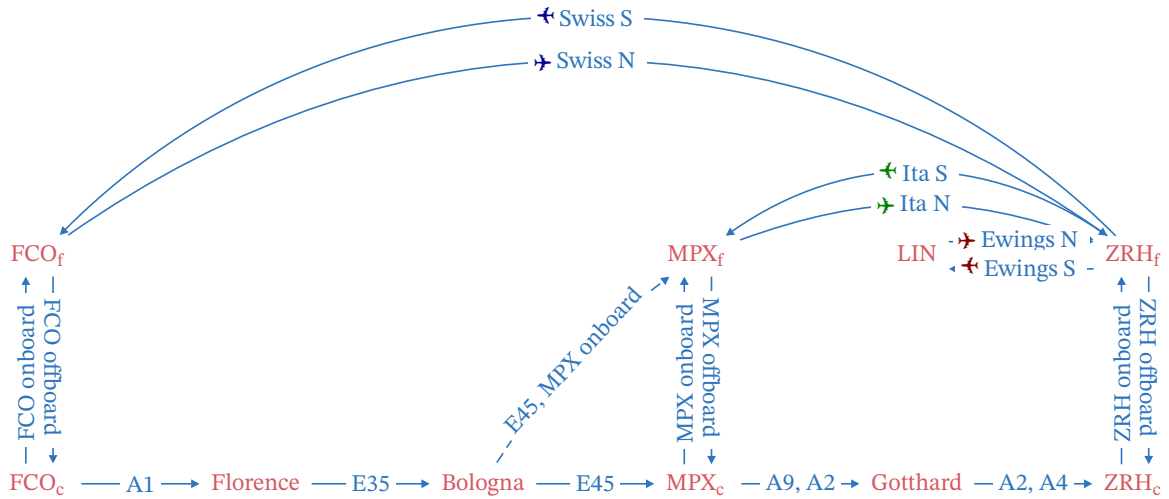


Figure 4.: Intermodal graph. The dashed arrows represent intermodal morphisms, and we depict just one of them for simplicity.

The intermodal network category **Intermodal** is the free category on the graph illustrated in Fig. 4.



## 15.2. Trekking in the Swiss Mountains

In the section we discuss a more “continuum-flavored” (as opposed to “discrete-flavored”) example of how one might describe “connectedness” using a category.

Suppose we are planning a hiking tour in the Swiss Alps. In particular, we wish to consider various routes for hikes. We have a map of the relevant region which uses coordinates  $\langle x, y, z \rangle$ . We assume the  $z$ -th coordinate is given by an “elevation function”,  $z = h(x, y)$ , and that  $h$  is  $C^1$  (a continuously differentiable function). We think of the graph of  $h$  as describing the surface of the landscape; call this surface  $L$ .

We will now define a category where the morphisms are built from  $C^1$  paths through the landscape, and such that these paths can be composed, essentially, by concatenation. We take paths which are  $C^1$  so that we can speak of the slope (steepness) of a path in any given point, as given by its derivative.

To set things up, we need to have a way to compose  $C^1$  paths such that their composition is again  $C^1$ . For this, the derivative (velocity) at the end of one path must match the starting velocity of the subsequent path.

### Definition 15.2 (Berg)

Let **Berg** be the category defined as follows:

▷ *Objects*: Objects are tuples  $\langle p, v \rangle$ , where

- $p \in L$ ,
- $v \in \mathbb{R}^3$  (we think of this as a tangent vector to  $L$  at  $p$ ).

▷ *Morphisms*: A morphism  $\langle p_1, v_1 \rangle \rightarrow \langle p_2, v_2 \rangle$  is  $\langle \gamma, T \rangle$ , where

- $T \in \mathbb{R}_{\geq 0}$ ,
- $\gamma : [0, T] \rightarrow L$  is a  $C^1$  function with  $\gamma(0) = p_1$  and  $\gamma(T) = p_2$ , as well as  $\dot{\gamma}(0) = v_1$  and  $\dot{\gamma}(T) = v_2$  (we take one-sided derivatives at the boundaries).

▷ *Identity morphism*: For any object  $\langle p, v \rangle$ , we define its identity morphism

$$\text{id}_{\langle p, v \rangle} = \langle \gamma, 0 \rangle \quad (5)$$

formally: its path  $\gamma$  is defined on the closed interval  $[0, 0]$ , (with  $T = 0$  and  $\gamma(0) = p$ ). We declare this path to be  $C^1$  by convention, and declare its derivative at 0 to be  $v$ .

▷ *Composition of morphisms*: Given morphisms

$$\langle \gamma_1, T_1 \rangle : \langle p_1, v_1 \rangle \rightarrow \langle p_2, v_2 \rangle \quad (6)$$

and

$$\langle \gamma_2, T_2 \rangle : \langle p_2, v_2 \rangle \rightarrow \langle p_3, v_3 \rangle, \quad (7)$$

their composition is  $\langle \gamma, T \rangle$  with  $T = T_1 + T_2$  and

$$\gamma(t) = \begin{cases} \gamma_1(t) & 0 \leq t \leq T_1 \\ \gamma_2(t - T_1) & T_1 \leq t \leq T_1 + T_2. \end{cases} \quad (8)$$

Since we are only amateurs, we don’t feel comfortable hiking on paths that are too steep in some places. We want to only consider paths that have a certain maximum inclination. Mathematically speaking, for any path – as described by a morphism  $\langle \gamma, T \rangle$  in the category **Berg** – we can compute its vertical inclination (vertical slope) and renormalize it to give a number in the interval  $[-1, +1]$ , say. (Here  $-1$  represents vertical descent, and  $+1$  represents vertical ascent.) Taking

absolute values of inclinations – call the resulting quantity “steepness” – we can compute the maximum steepness that a path  $\gamma$  obtains over its domain  $[0, T]$ . This gives, for every hom-set  $\text{Hom}_{\mathbf{Berg}}(\langle p_1, v_1 \rangle; \langle p_2, v_2 \rangle)$ , a function

$$\text{MaxSteepness} : \text{Hom}_{\mathbf{Berg}}(\langle p_1, v_1 \rangle; \langle p_2, v_2 \rangle) \longrightarrow [0, 1]. \quad (9)$$

Now, suppose we decide that we don’t want to traverse paths which have a maximal steepness greater than  $1/2$ . Paths which satisfy this condition we call *feasible*. Consider only the feasible paths in  $\mathbf{Berg}$ . If we keep the same objects as  $\mathbf{Berg}$ , but only consider feasible path, will the resulting structure still form a category? Should we restrict the set of objects for this to be true? We’ll let you ponder here; this type of question leads to the notion of a *subcategory*, which we’ll introduce soon in a subsequent chapter.

### 15.3. Currency categories

In this section, we introduce a kind of category for describing currency exchangers. Our idea is to model currencies as objects of a category, and morphisms will describe ways of exchanging between those currencies. As an example, currency exchangers offer this service.

We start with a set  $\mathbf{C}$  of labels for all the currencies we wish to consider:

$$\mathbf{C} = \{\text{EUR}, \text{USD}, \text{CHF}, \text{SGD}, \dots\}. \quad (10)$$

Now consider two currencies, say  $\text{USD}$  and  $\text{EUR}$ . How can we describe the process of changing an amount of  $\text{USD}$  to an amount of  $\text{EUR}$ ? We model this using two numbers: an exchange rate  $a$  and a commission  $d$  for the transaction. A morphism from one currency to another is given by this pair of numbers  $\langle a, b \rangle$ . Now, for each morphism, there is a function which takes an amount of the source currency and transforms it into an amount of the target currency. Given an amount  $x \in \mathbb{R}$  of  $\text{USD}$ , this function is:

$$\begin{aligned} f_{\langle a, d \rangle} : \mathbb{R} &\rightarrow \mathbb{R}, \\ x &\mapsto ax - d. \end{aligned} \quad (11)$$

Note that the commission  $d$  is to be intended in the target currency. Of course, for changing  $\text{USD}$  to  $\text{EUR}$ , there may be various different banks or agencies which each offer different exchange rates and/or different commissions. Each of these corresponds to a different morphism from  $\text{USD}$  to  $\text{EUR}$ .

To build our category, we also need to specify how currency exchangers compose. Given currencies  $X, Y, Z$ , and given currency exchangers  $\langle a, d \rangle$  from  $X$  to  $Y$  and  $\langle b, e \rangle$  from  $Y$  to  $Z$ , we define the composition  $\langle a, d \rangle \circ \langle b, e \rangle$  to be the currency exchanger from  $X$  to  $Z$  given by the pair of numbers

$$\langle ab, bd + e \rangle. \quad (12)$$

In other words, we compose currency exchangers as one would expect: we multiply the first and the second exchange rates together, and we add the commissions (paying attention to first transform the first commission into the units of the final target currency).

Finally, we also need to specify identity morphisms for our category. These are currency exchangers which “do nothing”. For any object  $X$ , its identity morphism is

$$\langle 1, 0 \rangle, \quad (13)$$

the currency exchanger from  $X$  to  $X$  with exchange rate “1” and commission “0”.

We now want to check that the composition of currency exchangers as defined above obeys unitality and associativity.

Given  $\langle 1, 0 \rangle$  from  $X$  to  $X$ ,  $\langle 1, 0 \rangle$  from  $Y$  to  $Y$ , and  $\langle a, d \rangle$  from  $X$  to  $Y$  we have:

$$\begin{aligned} \langle 1, 0 \rangle \circ \langle a, d \rangle &= \langle 1 \cdot a, 1 \cdot 0 + d \rangle \\ &= \langle a, d \rangle, \end{aligned} \quad (14)$$

and

$$\begin{aligned} \langle a, d \rangle \circ \langle 1, 0 \rangle &= \langle a \cdot 1, a \cdot 0 + d \rangle \\ &= \langle a, d \rangle, \end{aligned} \quad (15)$$

This is unitality. Furthermore, given  $\langle b, e \rangle$  from  $Y$  to  $Z$ , and  $\langle c, f \rangle$  from  $Z$  to  $U$

we have:

$$\begin{aligned}
 \langle \langle a, d \rangle \circ \langle b, e \rangle \rangle \circ \langle c, f \rangle &= \langle ab, bd + e \rangle \circ \langle c, f \rangle \\
 &= \langle abc, c(bd + e) + f \rangle \\
 &= \langle a, d \rangle \circ \langle bc, ce + f \rangle \\
 &= \langle a, d \rangle \circ (\langle b, e \rangle \circ \langle c, f \rangle).
 \end{aligned} \tag{16}$$

This is associativity. Thus, we indeed have a category.

We can formally define the category of currencies **Curr**.

**Definition 15.3** (Category **Curr**)

The *category of currencies* **Curr** is specified by:

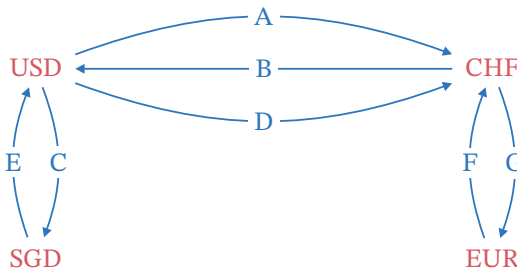
1. *Objects*: a collection of currencies.
2. *Morphisms*: given two currencies  $X, Y$ , morphisms between them are currency exchangers  $\langle a, d \rangle$  from  $X$  to  $Y$ .
3. *Identity morphism*: given a currency  $X$ , its identity morphism is the currency exchanger  $\langle 1, 0 \rangle$ . We also call such morphisms “trivial currency exchangers”.
4. *Composition of morphisms*: the composition of morphisms is given by the formula (12), describing a composed currency exchanger.

As an illustration, consider three currency exchange companies ExchATM, MoneyLah, and Frankurrencies, which operate on several currencies (Table 15.1).

**Table 15.1.** Three currency exchange companies operating different currencies.

Company name	Exchanger label	Direction	Exchange rate $a$	Fixed commission $b$
ExchATM	A	USD $\rightarrow$ CHF	0.95 (in CHF/USD)	2.0 (in CHF)
ExchATM	B	CHF $\rightarrow$ USD	1.05 (in USD/CHF)	1.5 (in USD)
ExchATM	C	USD $\rightarrow$ SGD	1.40 (in SGD/USD)	1.0 (in SGD)
MoneyLah	D	USD $\rightarrow$ CHF	1.00 (in CHF/USD)	1.0 (in CHF)
MoneyLah	E	SGD $\rightarrow$ USD	0.72 (in USD/SGD)	3.0 (in USD)
Frankurrencies	F	EUR $\rightarrow$ CHF	1.20 (in CHF/EUR)	0.0 (in CHF)
Frankurrencies	G	CHF $\rightarrow$ EUR	1.00 (in EUR/CHF)	1.0 (in EUR)

We can represent this information as a graph, where the nodes are the currencies and the edges are particular exchange operations (Fig. 5).



**Figure 5.** Three currency exchange companies operating different currencies as a graph.

There is a currency category built from the information in Table 15.1 and the graph in Fig. 5. Its collection of objects is the set  $\{\text{EUR}, \text{USD}, \text{CHF}, \text{SGD}\}$ , and its morphisms are, in total:

- ▷ the trivial currency exchanger (identity morphism)  $\langle 1, 0 \rangle$  for each of the four currencies (which are the objects),
- ▷ the currency exchangers corresponding to each item in Table 15.1,
- ▷ all possible compositions of the currency exchangers listed in Table 15.1.

The phrase “all possible compositions” is a bit vague. What we mean here can be made more precise. It corresponds to a general recipe for starting with a graph  $G$ , such as in Fig. 5, and obtaining from it an associated category, called the *free category on  $G$* .

We introduce this concept in Section 13.6.

**Exercise 30.** [Temperatures] Define a category of temperature converters, where the objects are **Celsius**, **Kelvin**, **Fahrenheit**, and the morphisms are the rules to transform a measurement from one unit to another.

Prove that this forms a category.

*See solution on page 299.*

## 15.4. Resources dependencies

In engineering design, one creates *systems* out of *components*. Each component has a reason to be in there. We will show how category theory can help in formalizing the chains of causality that underlie a certain design.

We will need to reason at the level of abstraction where we consider the “function”, or “functionality”, which each component provides, and the “requirements” that are needed to provide the function.

We will start with a simple example of the functioning principle of an electric car.

In an electric car, there is a battery, a store of the electric energy resource. We can see the production of motion as the series of two transformations:

- ▷ The **motor** transmutes the **electricity** into **rotation**.
- ▷ The **rotation** is converted into **translation** by the **wheels** and their friction with the road.

We see that there are two types of things in this example:

1. The “transmuted”: the **electricity**, the **rotation**, the **translation**: these are objects in a category of transmuted resources.
2. The “transmuters”: the **motor** and **wheels**: these are morphisms in a category of transmuted resources.

For a first qualitative description of the scenario, we might choose to just keep track of what is transmuted into what. We can draw a diagram in which each resource is a point (Fig. 6).



Figure 6.: Resources in the electric car example.

Now, we can draw arrows between two points if there is a transmuter between them.

We choose the direction of the arrow such that

$$X \xrightarrow{\text{transmuter}} Y \quad (17)$$

means that “using **transmuter**, having **Y** is sufficient to produce **X**”.

**Remark 15.4** (Are we going the wrong direction?). The chosen direction for the arrows is completely the opposite of what you would expect if you thought about “input and outputs”. There is a good reason to use this convention, though it will be apparent only a few chapters later. In the meantime, it is a good exercise to liberate your mind about the preconception of what an arrow means; in category theory there will be categories where the arrows represent much more abstract concepts than input/output.

Another way to write (17) would be as follows:

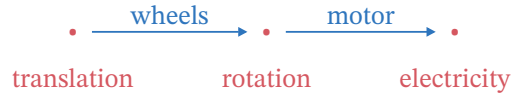
$$\text{transmuter} : X \rightarrow Y. \quad (18)$$

With these conventions, we can describe the two transmuters as these arrows:

$$\begin{aligned} \text{motor} &: \text{rotation} \rightarrow \text{electricity}, \\ \text{wheels} &: \text{translation} \rightarrow \text{rotation}. \end{aligned} \quad (19)$$

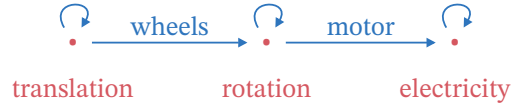
We can put these arrows in the diagrams, and obtain the following (Fig. 7).

**Figure 7.:** Transmuters are arrows between resources.



In this representation, the arrows are the components of the system. The basic rule is *composition*. If we use the semantics that an arrow from resource  $X$  to resource  $Y$  means “having  $Y$  is enough to obtain  $X$ ”, then, since  $Y$  is enough for  $Y$  per definition, we can add a self-loop for each resource. We will call the self-loops *identities* (Fig. 8).

**Figure 8.:** System components and identities.

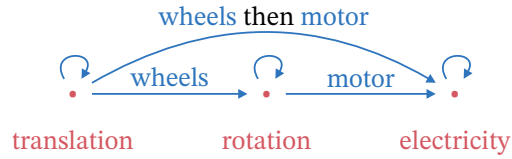


Furthermore, we might consider the idea of composition of arrows. Suppose that we know that

$$X \xrightarrow{f} Y \quad \text{and} \quad Y \xrightarrow{g} Z, \quad (20)$$

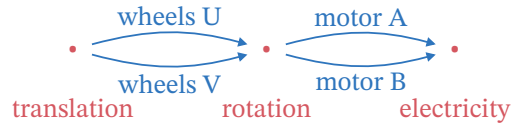
that is, using a  $g$  we can get a  $Y$  from a  $Z$ , and using a  $f$  we can get a  $X$  from a  $Y$ , then we conclude that using a  $f$  and a  $g$  we can get an  $X$  from a  $Z$ .

In our example, if the arrows **wheels** and **motor** exist, then also the arrow “**wheels then motor**” exists (Fig. 9).



**Figure 9.:** Composition of system components.

So far, we have drawn only one arrow between two points, but we can draw as many as we want. If we want to distinguish between different brands of motors, we would just draw one arrow for each model. For example, Fig. 10 shows two models of motors (**motor A**, and **motor B**) and two models of wheels (**wheels U** and **wheels V**).



**Figure 10.:** Multiple models for wheels and motors.

The figure implies now the existence of *four* composed arrows: “**wheels U then motor A**”, “**wheels U then motor B**”, “**wheels V then motor A**”, and “**wheels V then motor B**”, all going from **translation** to **electricity**;

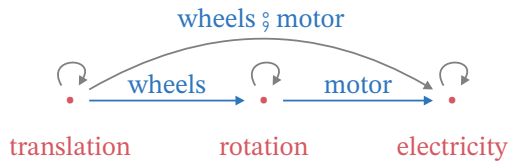
Note that we may save some ink when drawing diagrams of morphisms:

- ▷ We do not need to draw the identity arrows from one object to itself, because, by Def. 13.9, they always exist.
- ▷ Given arrows  $X \rightarrow Y$  and  $Y \rightarrow Z$ , we do not need to draw their composition because, by Def. 13.9, this composition is guaranteed to exist.

With these conventions, we can just draw the arrows **motor** and **wheels** in the diagram, and the rest of the diagram is implied (Fig. 11).

In particular, the electric car example corresponds to the category  $\mathbf{C}$  specified by

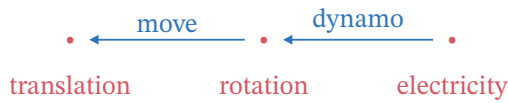
- ▷ *Objects:*  $\text{Ob}_{\mathbf{C}} = \{\text{electricity}, \text{rotation}, \text{translation}\}$ .



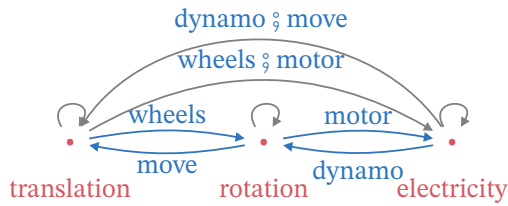
**Figure 11.:** Electric car example. The gray arrows are implied by the properties of a category.

▷ *Morphisms:* The system components are the morphisms. For instance, we have **motor**, **wheels**, and the morphism **wheels ; motor**, implied by the properties of the category.

We can slightly expand this example by noting the reverse transformations. In an electric car it is possible to regenerate power; that is, we can obtain **rotation** of the **wheels** from **translation** (via the morphism **move**), and then convert the **rotation** into **electricity** (via the morphism **dynamo**) (Fig. 12, Fig. 13).



**Figure 12.:** Electric power can be produced from motion.

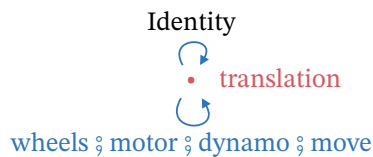


**Figure 13.:** Electric car example: forward and backward transformations.

Given the semantics of the arrows in a category, all compositions of arrows exist, even if they are not drawn explicitly. For example, we can consider the composition

$$\text{wheels ; motor ; dynamo ; move}, \quad (21)$$

which converts **translation** into **rotation**, into **electricity**, then back to **rotation** and **translation**. Note that this is an arrow that has the same head and tail as the identity arrow on **translation** (Fig. 14). However, these two arrows are not necessarily the same. In this example we are representing physical systems, so we would in fact not expect them to be the same, since there will be some losses during the many conversions.



**Figure 14.:** There can be multiple morphisms from an object to itself.

The directionality of the arrows is also important. While the convention of which resource is the tail and which the head is just a typographic convention, it might be the case that we know how to convert one resource into another, but not vice versa. Figure 15 shows an example of a diagram that describes a process which is definitely not invertible.

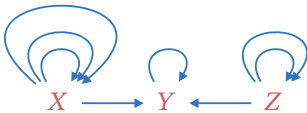
**Example 15.5.** Given any category  $\mathbf{C}$ , and any object  $X \in \mathbf{C}$ , the set of *endomorphisms*  $\text{Hom}_{\mathbf{C}}(X; X)$  is a monoid. The category depicted in Fig. 16 has three objects  $X, Y, Z$  and several morphisms.  $X$  has four endomorphisms,  $Y$  two, and  $Z$





**Figure 15.:** An example of a process which is not invertible.

three (including identity morphisms). Take the binary operation  $\circ$  to be the composition  $\circ$  in  $\mathbf{C}$ , and the neutral element to be the identity  $\text{id}_X$ . The associativity and unitality laws of the category  $\mathbf{C}$  coincide with the ones of the monoid's definition, and are satisfied. Therefore, we can identify a monoid as a one-object category.



**Figure 16.**

## 15.5. Procedures

In programming, it is common to use the name *function*; we keep that word to denote mathematical functions: morphisms of the category **Set**. We use the word *procedure* to refer to pieces of code that run on a computer or virtual machine.

Procedures are richer than functions:

- ▷ They might be non-deterministic: not always return the same value.
- ▷ They might have side effects: change the world in some way.
- ▷ They have resource consumption associated to them: they need memory and computation to produce the result.

The category **Set** is not sufficient to describe these properties, but we can easily invent categories that are built on top of **Set** to add these properties.

We are going to do this in stages. First, we are going to define a category of procedures that keeps track of running time. Then we are going to keep track of time depending on the size of the input. Finally, we are going to keep track of memory usage.

### Modeling execution time

We can model execution time by keeping track of an additional real number in the morphism.

#### Definition 15.6 (Semicategory **ProcTime**)

The semicategory **ProcTime** consists of the following constituents:

1. *Objects*: The objects of **ProcTime** are the objects of **Set**.
2. *Morphisms*: A morphism

$$f : X \rightarrow_{\mathbf{ProcTime}} Y \quad (22)$$

is a pair  $\langle f_e, t \rangle$ , where  $f_e : X \rightarrow_{\mathbf{Set}} Y$  is a regular function that describes what the procedure computes and  $t > 0$  is a real number representing “execution time”.

3. *Composition operation*: Given two morphisms

$$f : X \rightarrow_{\mathbf{ProcTime}} Y \quad \text{and} \quad g : Y \rightarrow_{\mathbf{ProcTime}} Z, \quad (23)$$

represented by  $\langle f_e, t_1 \rangle$  and  $\langle g_e, t_2 \rangle$ , their composition is given by  $\langle f_e \circ_{\mathbf{Set}} g_e, t_1 + t_2 \rangle$ .

This category allows distinguishing between different implementations of the same functions with different computational requirements. For example, there could be two morphisms  $\langle f_e, 1 \rangle$  and  $\langle f_e, 10 \rangle$  which compute the same function, but the second takes 10 times as much.

### Modeling sized data

The category **ProcTime** models execution time, but it does not model how the execution time depends on the size of the input.

If we wanted to capture such ideas, we need to have an explicit way to talk about the size of the input. For this we introduce “sized sets”.

**Definition 15.7** (Sized set)

A sized set is a pair  $\langle \mathbf{A}, \text{size} \rangle$ , where  $\mathbf{A}$  is a set, and  $\text{size} : \mathbf{A} \rightarrow \mathbb{N} \cup \{\infty\}$  is the size function.

For example, we could have  $\mathbf{A} = \text{lists of integers}$  and  $\text{size} = \text{length of list}$ .

Now we can create a category whose objects are sized sets.

The procedures are morphisms that keep track of how the size of the input influences the size of the output. For example, a sorting function produces an output that is the same size as the input, while a function that chooses an element of a list produces an output of size 1, no matter how large the input is.

**Definition 15.8** (Semicategory **ProcSize**)

The semicategory **ProcSize** consists of the following constituents:

1. *Objects*: The objects are sized sets.
2. *Morphisms*: A morphism

$$f : X \rightarrow_{\text{ProcSize}} X \quad (24)$$

between the two objects

$$X = \langle \mathbf{A}, \text{size}_{\mathbf{A}} \rangle \quad \text{and} \quad Y = \langle \mathbf{B}, \text{size}_{\mathbf{B}} \rangle \quad (25)$$

is a pair

$$\langle f_e, \sigma \rangle, \quad (26)$$

where:

- a)  $f_e : \mathbf{A} \rightarrow \mathbf{B}$  is the function computed;
  - b)  $\sigma : \mathbb{N} \rightarrow \mathbb{N}$  keeps track of how the size changes.
3. *Composition*: The composition of

$$\langle f_e, \sigma_f \rangle \quad \text{and} \quad \langle g_e, \sigma_g \rangle \quad (27)$$

is given by

$$\langle f_e \circ g_e, \sigma_f \circ \sigma_g \rangle. \quad (28)$$

**Modeling data-size-dependent running times**

Now we can create a category that keeps track of both data size and execution time, possibly dependent on data size.

**Definition 15.9** (Semicategory **ProcSizeTime**)

The semicategory **ProcSizeTime** consists of the following constituents:

1. *Objects*: The objects are sized sets.
2. *Morphisms*: A morphism

$$f : X \rightarrow_{\text{ProcSizeTime}} X \quad (29)$$

between the two objects

$$X = \langle \mathbf{A}, \text{size}_{\mathbf{A}} \rangle \quad \text{and} \quad Y = \langle \mathbf{B}, \text{size}_{\mathbf{B}} \rangle \quad (30)$$

is a tuple

$$\langle f_e, \sigma, \text{dur} \rangle, \quad (31)$$

where:

- a)  $f_e : \mathbf{A} \rightarrow \mathbf{B}$  is the function computed;
- b)  $\sigma : \mathbb{N} \rightarrow \mathbb{N}$  keeps track of how the size changes.
- c)  $\text{dur} : \mathbb{N} \rightarrow \mathbb{R}$  says what is the computation time as a function of instance size;

3. *Composition*: The composition of

$$\langle f_1, \sigma_1, \text{dur}_1 \rangle \quad \text{and} \quad \langle g_2, \sigma_2, \text{dur}_2 \rangle \quad (32)$$

is given by

$$\langle f_1 \circ g_2, \sigma_1 \circ \sigma_2, \text{dur}_{1,2} \rangle, \quad (33)$$

where  $\text{dur}_{1,2}$  is defined as

$$\begin{aligned} \text{dur}_{1,2} : \mathbb{N} &\rightarrow \mathbb{R}, \\ n &\mapsto \text{dur}_1(n) + \text{dur}_2(\sigma_1(n)). \end{aligned} \quad (34)$$

**Exercise 31.** Check that associativity holds for the composition in **ProcSizeTime**.

*See solution on page 300.*

#### Graded exercise E.10 (Asymptotics)

The category **ProcSizeTime** as defined thinks of time as real numbers, and size of the data as integers. In computer science, it is convenient to use *asymptotic analysis*. For example, we know that, in the general case, ordering a list of  $n$  elements takes  $O(n \log n)$  steps.

Discuss how you can extend or modify **ProcSizeTime** to be able to capture asymptotic analysis.

## 15.6. Software dependencies



We will discuss two examples:

- ▷ Makefiles.
- ▷ Python packages dependencies.

## 15.7. ☉ Currency categories

This exercise is based on the currency category described in Section 15.3.

Using the same format introduced in Code Exercise E.1, we describe a currency category as in Fig. 17. In this example, there are two objects: CHF and USD. There are two identity morphisms, as well as a morphism from CHF and USD.

Recall that a morphism in **Curr** is given by two numbers: a rate and a commission. The two identity morphisms have rate 1.0 and commission 0.0. The morphism  $f$  has rate 0.95 and commission 2.0.

To parse this semicategory, we need a series of pieces that are already implemented and that you can find in the package `act4e_interfaces`.

We already give you the implementation of `CurrencyExchanger` (Listing 81), which represents the currency exchanger.

Listing 81: The `CurrencyExchanger` interface.

```
@dataclass
class CurrencyExchanger:
    rate: float
    commission: float
```

```
objects:
  USD:
    obdata: USD
    identity:
      modata:
        rate: 1.0
        commission: 0.0
  CHF:
    obdata: CHF
    identity:
      modata:
        rate: 1.0
        commission: 0.0
morphisms:
  f:
    modata:
      rate: 0.95
      commission: 2.0
    source: USD
    target: CHF
    equations: {}
```

Figure 17.: `cat_example1`

The composition function is given by the function `currency_exchange_compose()` in Listing 82.

Listing 82: The `currency_exchange_compose` interface.

```
def currency_exchange_compose(
    _ob1: None, _ob2: None, _ob3: None, t1: CurrencyExchanger, t2:
    ← CurrencyExchanger
) -> CurrencyExchanger:
    rate = t1.rate * t2.rate
    commission = t1.commission * t2.rate + t2.commission
    return CurrencyExchanger(rate, commission)
```

Finally, the setoid `AllCurrencyExchangers` is already implemented (Listing 83).

Listing 83: The `AllCurrencyExchangers` interface.

```
class AllCurrencyExchangers(Setoid[CurrencyExchanger]):
    def contains(self, x: CurrencyExchanger) -> bool:
        return isinstance(x, CurrencyExchanger) and x.rate > 0 and x.commission
        ← >= 0

    def save(self, h: IOHelper, x: CurrencyExchanger) -> ConcreteRepr:
        return dict(rate=x.rate, commission=x.commission)

    def load(self, h: IOHelper, o: CurrencyExchanger_desc) ->
    ← CurrencyExchanger:
        return CurrencyExchanger(o["rate"], o["commission"])
```

In the tests, the category will be loaded using the code you wrote for Code Exercise E.1 and the pieces above. In the end, we will have created an object of type

```
SemiCategory[RichObject[str], RichMorphism[CurrencyExchanger]]
```

What you have to implement is the interface in Listing 84.

The function `compute_optimal_conversion()` has the following parameters:

1. `available` is the semicategory already loaded.

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

2. source is the name of the source currency.
3. amount is the amount to be converted.
4. target is the target currency.

The function must return an object of type `OptimalSolution` (Listing 85), which has two fields:

1. `optimal_path` is the sequence of exchangers to use.
2. `final_amount` is the final amount in the target categories.

**Listing 84:** The `CurrencyOptimization` interface.

```
class CurrencyOptimization:
    @abstractmethod
    def compute_optimal_conversion(
        self,
        available: SemiCategory[RichObject[str],
        ← RichMorphism[CurrencyExchanger]],
        source: str,
        amount: float,
        target: str,
    ) -> OptimalSolution:
        """Returns the optimal path for converting a certain amount from source
        ← to target.
        Raises InvalidValue if there is no path."""
```

**Listing 85:** The `OptimalSolution` interface.

```
@dataclass
class OptimalSolution:
    optimal_path: List[str]
    final_amount: float
```

### Code exercise E.2 (TestCurrencyOptimization)

Implement the interface in Listing 84.

Test using:

```
act4e-check TestCurrencyOptimization
```

**Table 15.2.**

queries		results	
currencies	val	result	path
USD → USD	10	10	[id_USD]
CHF → CHF	10	10	[id_CHF]
USD → CHF	10	7.5	[f]
CHF → USD	10	no path exists	

**Example queries and result** For the example in Fig. 17, Table 15.2 shows some examples queries and the expected result. In this example, the problem are trivial: there is only one morphism to go from **USD** to **CHF**. Therefore, it does not matter what is the initial amount. In the unit tests, there are more interesting cases, in which the optimal path depends on the amount.

**Hints** Here are some hints:

- ▷ The difficulty in this exercise lies in efficiently exploring the (possibly infinite) set of morphisms. Recall that the interface of `SemiCategory` has the function `hom()` that allows to specify up to what level you want the morphisms.
- ▷ One thing to notice is that no optimal path will ever employ a currency converter twice.
- ▷ You might be needing the functionality of the `networkx` Python package. Note that there are various graph types to choose from. You likely need to use `MultiDiGraph` rather than `DiGraph`.





# 16. Semicategory actions

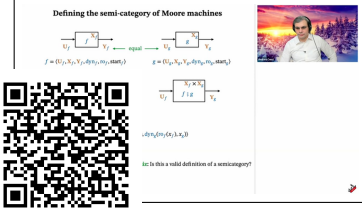
Categories are well suited to represent processes: both physical processes that move and transform matter, and software processes that communicate and compute.

16.1 Moore machines, first version	260
16.2 The category $\langle \text{Set} \rangle$	264
16.3 Moore machines, $\langle \text{Set} \rangle$ version	266
16.4 Standard action of Moore machines	269
16.5 Semicategory actions	272
16.6 Different actions of Moore machines	274
16.7 More machines	277
16.8 LTI systems	279

Stadler Rail is a Swiss manufacturer of railway rolling stock. Among others, it produces Swiss regional trains and trams. The company, founded in 1942 by Ernst Stadler, counts over 8,500 employees, and has 12 factories.



Watch: *Moore machines* (19 minutes).



## 16.1. Moore machines, first version

In the following we consider **Moore machines**<sup>\*</sup>, which are a type of basic model for describing certain dynamical systems. Its characteristic features are:

- ▷ a **state space** which describes all possible states that the system can possibly be in;
- ▷ an **input space** and an **output space**;
- ▷ a **dynamics** which describes how, given an input, the system's state changes according to that input;
- ▷ a **read-out** which relates the current state of the system to the output space.

We are in particular interested in the ways that such a machine can transform a sequence of inputs into a sequence of outputs.

### A first mathematical model

Our first version for formalizing the idea of a Moore machine looks as follows. We model the input space, state space, and output space as sets  $\mathbf{U}$ ,  $\mathbf{X}$ , and  $\mathbf{Y}$  respectively, and we model the dynamics and read-out as functions

$$\begin{cases} \text{dyn} : \mathbf{U} \times \mathbf{X} \rightarrow \mathbf{X}, \\ \text{ro} : \mathbf{X} \rightarrow \mathbf{Y}. \end{cases} \quad (1)$$

We will also choose an element  $\text{st} \in \mathbf{X}$  of the state space as an initial state; we will use this when specifying how a Moore machine acts on sequences of inputs.

Thus, in total, a Moore machine is specified by a tuple of the following type:

$$\langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle. \quad (2)$$

**Example 16.1** (Robot on a grid). We consider a robot moving on a two-dimensional unit grid, represented as  $\mathbb{N}_{[0,100]}^2$  (Fig. 1).

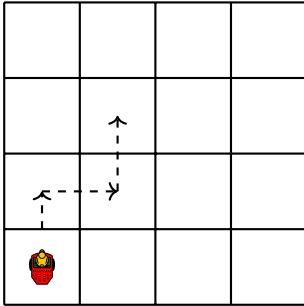


Figure 1.: Robot on a grid example.

The state of the robot is described by its position coordinates, as  $\mathbf{x} \in \mathbb{N}_{[0,100]}^2$  (for convenience, consider the first element of the state to be the horizontal coordinate and the second element to be the vertical one). The robot can choose from a set of input actions  $\mathbf{U} = \{\uparrow, \downarrow, \rightarrow, \leftarrow\}$ , each representing a movement of 1 unit in the corresponding direction on the grid. We can write the dynamics as

$$\begin{aligned} \text{dyn} : \mathbf{U} \times \mathbf{X} &\rightarrow \mathbf{X} \\ \langle \uparrow, \langle x_1, x_2 \rangle \rangle &\mapsto \langle x_1, \min(x_2 + 1, 100) \rangle, \\ \langle \downarrow, \langle x_1, x_2 \rangle \rangle &\mapsto \langle x_1, \max(x_2 - 1, 0) \rangle, \\ \langle \rightarrow, \langle x_1, x_2 \rangle \rangle &\mapsto \langle \min(x_1 + 1, 100), x_2 \rangle, \\ \langle \leftarrow, \langle x_1, x_2 \rangle \rangle &\mapsto \langle \max(x_1 - 1, 0), x_2 \rangle, \end{aligned} \quad (3)$$

where the “min” and “max” ensure that the robot stays in within the boundaries of the grid. The readout can be thought of as a sensor measuring the position of the robot (the state). Assuming a perfect sensor, we have:

$$\begin{aligned} \text{ro} : \mathbf{X} &\rightarrow \mathbf{Y}, \\ \langle x_1, x_2 \rangle &\mapsto \langle x_1, x_2 \rangle. \end{aligned}$$

The starting position of the robot can be specified as any  $\text{st} \in \mathbb{N}_{[0,100]}^2$ .

<sup>\*</sup> They are named after Edward F. Moore.

## Moore machines acting on sequences

Given a Moore machine  $f = \langle \mathbf{U}, \mathbf{Y}, \mathbf{X}, \text{dyn}, \text{ro}, \text{st} \rangle$ , the following is a standard way to think of it as acting on a sequence of inputs, transforming it into a sequence of outputs.

Given an infinite sequence of inputs  $u_0, u_1, u_2, \dots$ , we use the following recipe

$$\begin{cases} x_{k+1} = \text{dyn}(u_k, x_k) \\ y_k = \text{ro}(x_k), \end{cases} \quad (4)$$

to produce an infinite sequence  $y_0, y_1, y_2, \dots$  of outputs. For the very first step, when  $k = 0$ , we need the initial state  $x_0 = \text{st}$  in order to compute  $x_1 = \text{dyn}(u_0, x_0)$  and  $y_0 = \text{ro}(x_0)$ .

**Example 16.2.** To continue Example 16.1, we can consider a robot with starting position  $\langle 0, 0 \rangle$ . We consider a sequence of inputs  $\rightarrow, \uparrow, \rightarrow, \downarrow, \leftarrow$ . Using the recipe given by the Moore machine in Example 16.1, one can find the sequences of states and outputs, which in this case coincide, given the identity readout:

$$\begin{aligned} x_0 = \text{st} = y_0 &= \langle 0, 0 \rangle \\ x_1 = y_1 &= \langle 1, 0 \rangle, \\ x_2 = y_2 &= \langle 1, 1 \rangle, \\ x_3 = y_3 &= \langle 2, 1 \rangle, \\ x_4 = y_4 &= \langle 2, 0 \rangle, \\ x_5 = y_5 &= \langle 1, 0 \rangle, \end{aligned}$$

An infinite sequence  $u_0, u_1, u_2, \dots$  of elements of  $\mathbf{U}$  can be also thought as a function  $u : \mathbb{N} \rightarrow \mathbf{U}$  with  $u(0) = u_0, u(1) = u_1, u(2) = u_2, \dots$  etc. To denote an infinite sequence of elements of  $\mathbf{U}$  we use the two notations  $\mathbf{U}^{\mathbb{N}}$  and **Stream**  $\mathbf{U}$ :

$$\text{Stream } \mathbf{A} := (\mathbb{N} \rightarrow \mathbf{A}) = \mathbf{A}^{\mathbb{N}}. \quad (5)$$

For a fixed Moore machine  $f$ , the recipe (4) defines a function  $\text{act}_f$  which maps any given sequence  $u$  of elements of  $\mathbf{U}$  to a corresponding sequence  $y = \text{act}_f(u)$  of elements of  $\mathbf{Y}$ . In other words, from  $\langle \mathbf{U}, \mathbf{Y}, \mathbf{X}, \text{dyn}, \text{ro}, \text{st} \rangle$  and (4) we obtain

$$\text{act}_f : \text{Stream } \mathbf{U} \rightarrow \text{Stream } \mathbf{Y}. \quad (6)$$

We think of this function  $\text{act}_f$  as describing the external behavior of the Moore machine  $f$ , because it encodes what is externally observable in terms of how the Moore machine is used to relate inputs to outputs.

## Composing Moore machines

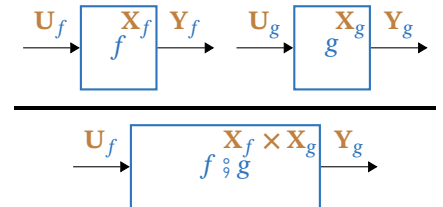
Let us consider composing Moore machines serially by letting the output of one machine be the input of the next. We'd like the result to again be a Moore machine.

In other words, given Moore machines

$$f = \langle \mathbf{U}_f, \mathbf{X}_f, \mathbf{Y}_f, \text{dyn}_f, \text{ro}_f, \text{st}_f \rangle \quad (7)$$

and

$$g = \langle \mathbf{U}_g, \mathbf{X}_g, \mathbf{Y}_g, \text{dyn}_g, \text{ro}_g, \text{st}_g \rangle, \quad (8)$$



**Figure 2.:** Composition of Moore machines (first version).

with  $\mathbf{Y}_f = \mathbf{U}_g$ , we'd like to define their composition as a Moore machine

$$f \circ g = \langle \mathbf{U}_{f \circ g}, \mathbf{X}_{f \circ g}, \mathbf{Y}_{f \circ g}, \text{dyn}_{f \circ g}, \text{ro}_{f \circ g}, \text{st}_{f \circ g} \rangle. \quad (9)$$

The situation is illustrated in Fig. 2.

Here is one way to do it: we set

$$\begin{aligned} \mathbf{U}_{f \circ g} &:= \mathbf{U}_f, \\ \mathbf{X}_{f \circ g} &:= \mathbf{X}_f \times \mathbf{X}_g, \\ \mathbf{Y}_{f \circ g} &:= \mathbf{Y}_g, \\ \text{st}_{f \circ g} &:= \langle \text{st}_f, \text{st}_g \rangle, \end{aligned} \quad (10)$$

we define the composite dynamics to be

$$\begin{aligned} \text{dyn}_{f \circ g} : \mathbf{U}_f \times (\mathbf{X}_f \times \mathbf{X}_g) &\rightarrow (\mathbf{X}_f \times \mathbf{X}_g), \\ \langle u, \langle x_f, x_g \rangle \rangle &\mapsto \langle \text{dyn}_f(u, x_f), \text{dyn}_g(\text{ro}_f(x_f), x_g) \rangle, \end{aligned} \quad (11)$$

and we define the composite readout to be

$$\begin{aligned} \text{ro}_{f \circ g} : (\mathbf{X}_f \times \mathbf{X}_g) &\rightarrow \mathbf{Y}_g, \\ \langle x_f, x_g \rangle &\mapsto \text{ro}_g(x_g). \end{aligned} \quad (12)$$

This formalization works very nicely – it models composition using the output of one machine as the input of the next, and result of composition is again a Moore machine.

However, there is one aspect which we wish were different: this composition operation is not associative. It *almost* is, but not quite. We explain why below.

The reason we wish it *were* associative is that Moore machines would then form a semicategory, and we could integrate them nicely into the mathematics we have been developing thus far.

The culprit for associativity failing is the cartesian product that we use to define the state space  $\mathbf{X}_f \times \mathbf{X}_g$  of a composite Moore machine  $f \circ g$ . Consider three composable systems  $f$ ,  $g$ , and  $h$ . If we compose them in the two ways  $(f \circ g) \circ h$  and  $f \circ (g \circ h)$ , then their respective state spaces are  $(\mathbf{X}_f \times \mathbf{X}_g) \times \mathbf{X}_h$  and  $\mathbf{X}_f \times (\mathbf{X}_g \times \mathbf{X}_h)$ . These sets are *isomorphic*, but they are not equal on the nose, and hence also the Moore machines  $(f \circ g) \circ h$  and  $f \circ (g \circ h)$  are very close to being equal, but are not quite.

To see why

$$(\mathbf{X}_f \times \mathbf{X}_g) \times \mathbf{X}_h \neq \mathbf{X}_f \times (\mathbf{X}_g \times \mathbf{X}_h), \quad (13)$$

recall that the elements of  $(\mathbf{X}_f \times \mathbf{X}_g) \times \mathbf{X}_h$  are nested tuples of the form

$$\langle \langle x_f, x_g \rangle, x_h \rangle \quad (14)$$

while the elements of  $\mathbf{X}_f \times (\mathbf{X}_g \times \mathbf{X}_h)$  are nested tuples of the form

$$\langle x_f, \langle x_g, x_h \rangle \rangle. \quad (15)$$

An isomorphism between the two sets is given by the following function

$$\begin{aligned} (\mathbf{X}_f \times \mathbf{X}_g) \times \mathbf{X}_h &\rightarrow \mathbf{X}_f \times (\mathbf{X}_g \times \mathbf{X}_h) \\ \langle \langle x_f, x_g \rangle, x_h \rangle &\mapsto \langle x_f, \langle x_g, x_h \rangle \rangle \end{aligned} \quad (16)$$

which simply re-brackets the tuples.

In the next section we will introduce a technical construction for defining a product similar to the cartesian product, but which is associative on the nose - not just “up to an isomorphism”. This construction will allow us to make a new, modified formalization of Moore machines which form a bona fide semicategory, and it will prove useful in other respects further down the road.

## 16.2. The category $\langle \mathbf{Set} \rangle$

We will define a category  $\langle \mathbf{Set} \rangle$  whose objects are “sets of tuples”.

Given sets  $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_n$ , where  $n \in \mathbb{N}$ , we define

$$\langle \mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_n \rangle := \{ \langle x_1, x_2, \dots, x_n \rangle \mid x_1 \in \mathbf{A}_1, x_2 \in \mathbf{A}_2, \dots, x_n \in \mathbf{A}_n \}. \quad (17)$$

So for example,

$$\langle \mathbb{Z}, \mathbb{N}, \mathbb{R} \rangle = \{ \langle x, y, z \rangle \mid x \in \mathbb{Z}, y \in \mathbb{N}, z \in \mathbb{R} \}. \quad (18)$$

Note the case  $n = 0$ , where

$$\langle \rangle = \{ \langle \rangle \} \quad (19)$$

is a singleton set whose element is the empty tuple.

All the objects of  $\langle \mathbf{Set} \rangle$  are in particular just sets, so we can package them into a category whose morphisms are simply functions between such sets.

### Definition 16.3 ( $\langle \mathbf{Set} \rangle$ )

The category  $\langle \mathbf{Set} \rangle$  is the category whose objects are those sets which are of the form  $\langle \mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_n \rangle$ , for any  $n \in \mathbb{N}$  and any sets  $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_n$ . Morphisms in  $\langle \mathbf{Set} \rangle$  are any functions between such sets, and the composition operations and identities are the usual ones for functions.

### From Set to $\langle \mathbf{Set} \rangle$

Observe that we can turn any set into an object of  $\langle \mathbf{Set} \rangle$ : given a set  $\mathbf{A}$ , there is the associated set

$$\langle \mathbf{A} \rangle = \{ \langle x \rangle \mid x \in \mathbf{A} \} \quad (20)$$

whose elements are precisely all the tuples of length one whose entry is an element of  $\mathbf{A}$ . This defines a function

$$\begin{aligned} \mathbf{Ob}_{\mathbf{Set}} &\rightarrow \mathbf{Ob}_{\langle \mathbf{Set} \rangle} \\ \mathbf{A} &\mapsto \langle \mathbf{A} \rangle \end{aligned} \quad (21)$$

which one might call “bracket”.

Similarly, given a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  between sets, there is an associated function  $\langle f \rangle : \langle \mathbf{A} \rangle \rightarrow \langle \mathbf{B} \rangle$  given by

$$\langle f \rangle(\langle x \rangle) := \langle f(x) \rangle. \quad (22)$$

This defines a function

$$\mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}, \mathbf{B}) \rightarrow \mathbf{Hom}_{\langle \mathbf{Set} \rangle}(\langle \mathbf{A} \rangle, \langle \mathbf{B} \rangle) \quad (23)$$

for any sets  $\mathbf{A}$  and  $\mathbf{B}$ ; one might also call it “bracket”.

### Concatenation of tuples

Given tuples  $\langle x, y, z \rangle$  and  $\langle u, v \rangle$  we can stick them together to make the longer tuple  $\langle x, y, z, u, v \rangle$ , using the already defined concatenation of tuples.

## Multiplication in $\langle \mathbf{Set} \rangle$

We define a multiplication for objects of  $\langle \mathbf{Set} \rangle$ . The symbol  $\circ_{\langle \rangle}$  will denote this multiplication in infix notation.

Given  $\langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle$  and  $\langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle$  we define the operation

$$\circ_{\langle \rangle} : \mathbf{Ob}_{\langle \mathbf{Set} \rangle} \times \mathbf{Ob}_{\langle \mathbf{Set} \rangle} \rightarrow \mathbf{Ob}_{\langle \mathbf{Set} \rangle} \quad (24)$$

by

$$\langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle \circ_{\langle \rangle} \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle := \langle \mathbf{A}_1, \dots, \mathbf{A}_m, \mathbf{B}_1, \dots, \mathbf{B}_n \rangle \quad (25)$$

for any  $n, m \in \mathbb{N}$ .

The elements of  $\langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle \circ_{\langle \rangle} \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle$  are all possible concatenations of elements of  $\langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle$  with elements of  $\langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle$ .

Equation (25) holds in particular also in the cases when  $m = 0$  or  $n = 0$ :

$$\langle \rangle \circ_{\langle \rangle} \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle = \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle \quad (26)$$

and

$$\langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle \circ_{\langle \rangle} \langle \rangle = \langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle. \quad (27)$$

The multiplication of objects in  $\langle \mathbf{Set} \rangle$  is associative (which was our goal with making this construction). Indeed, both

$$(\langle \mathbf{A}_1, \dots, \mathbf{A}_l \rangle \circ_{\langle \rangle} \langle \mathbf{B}_1, \dots, \mathbf{B}_m \rangle) \circ_{\langle \rangle} \langle \mathbf{C}_1, \dots, \mathbf{C}_n \rangle \quad (28)$$

and

$$\langle \mathbf{A}_1, \dots, \mathbf{A}_l \rangle \circ_{\langle \rangle} (\langle \mathbf{B}_1, \dots, \mathbf{B}_m \rangle \circ_{\langle \rangle} \langle \mathbf{C}_1, \dots, \mathbf{C}_n \rangle) \quad (29)$$

are equal to

$$\langle \mathbf{A}_1, \dots, \mathbf{A}_l, \mathbf{B}_1, \dots, \mathbf{B}_m, \mathbf{C}_1, \dots, \mathbf{C}_n \rangle. \quad (30)$$

**Remark 16.4.** Note that, for any sets  $\mathbf{A}, \mathbf{B}$ , we have

$$\langle \mathbf{A}, \mathbf{B} \rangle = \mathbf{A} \times \mathbf{B}. \quad (31)$$

In other words, the multiplication operation in  $\langle \mathbf{Set} \rangle$  “coincides” with the cartesian product operation in the special case when we are multiplying two sets together.

Because of this, the operation of cartesian product is well-defined for  $\langle \mathbf{Set} \rangle$ . Namely, given tuple sets  $\langle \mathbf{A}_1, \dots, \mathbf{A}_l \rangle$  and  $\langle \mathbf{B}_1, \dots, \mathbf{B}_m \rangle$ , their cartesian product is again a tuple set:

$$\langle \mathbf{A}_1, \dots, \mathbf{A}_l \rangle \times \langle \mathbf{B}_1, \dots, \mathbf{B}_m \rangle = \langle \langle \mathbf{A}_1, \dots, \mathbf{A}_l \rangle \langle \mathbf{B}_1, \dots, \mathbf{B}_m \rangle \rangle, \quad (32)$$

(which is a “nested” tuple set).

### 16.3. Moore machines, $\langle \mathbf{Set} \rangle$ version

We can now slightly rework the definition of Moore machines using morphisms in  $\langle \mathbf{Set} \rangle$  rather than  $\mathbf{Set}$ .

**Definition 16.5** (Moore machine,  $\langle \mathbf{Set} \rangle$  version)

A *Moore machine* is a tuple

$$\langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle, \quad (33)$$

where  $\mathbf{U}$ ,  $\mathbf{X}$ , and  $\mathbf{Y}$  are objects of  $\langle \mathbf{Set} \rangle$ ,

$$\text{dyn} : \mathbf{U} \circlearrowleft \mathbf{X} \rightarrow \mathbf{X}, \quad (34)$$

and

$$\text{ro} : \mathbf{X} \rightarrow \mathbf{Y}, \quad (35)$$

are morphisms in  $\langle \mathbf{Set} \rangle$  (so they are functions), and  $\text{st} \in \mathbf{X}$ .

#### Composition

Consider two Moore machines

$$f = \langle \mathbf{U}_f, \mathbf{X}_f, \mathbf{Y}_f, \text{dyn}_f, \text{ro}_f, \text{st}_f \rangle \quad (36)$$

$$g = \langle \mathbf{U}_g, \mathbf{X}_g, \mathbf{Y}_g, \text{dyn}_g, \text{ro}_g, \text{st}_g \rangle, \quad (37)$$

that are compatible for composition, in the sense that  $\mathbf{Y}_f = \mathbf{U}_g$ . Their composition is given by

$$f \circ g = \langle \mathbf{U}_{f \circ g}, \mathbf{X}_{f \circ g}, \mathbf{Y}_{f \circ g}, \text{dyn}_{f \circ g}, \text{ro}_{f \circ g}, \text{st}_{f \circ g} \rangle, \quad (38)$$

where

$$\begin{aligned} \mathbf{U}_{f \circ g} &:= \mathbf{U}_f, \\ \mathbf{X}_{f \circ g} &:= \mathbf{X}_f \circlearrowleft \mathbf{X}_g, \\ \mathbf{Y}_{f \circ g} &:= \mathbf{Y}_g, \\ \text{st}_{f \circ g} &:= \text{st}_f \circlearrowleft \text{st}_g, \end{aligned} \quad (39)$$

and  $\text{dyn}_{f \circ g}$  and  $\text{ro}_{f \circ g}$  are defined as

$$\begin{aligned} \text{dyn}_{f \circ g} : \mathbf{U}_f \circlearrowleft \mathbf{X}_f \circlearrowleft \mathbf{X}_g &\rightarrow \mathbf{X}_f \circlearrowleft \mathbf{X}_g, \\ u \circlearrowleft x_f \circlearrowleft x_g &\mapsto \text{dyn}_f(u \circlearrowleft x_f) \circlearrowleft \text{dyn}_g(\text{ro}_f(x_f) \circlearrowleft x_g), \end{aligned} \quad (40)$$

and

$$\begin{aligned} \text{ro}_{f \circ g} : \mathbf{X}_f \circlearrowleft \mathbf{X}_g &\rightarrow \mathbf{Y}_g, \\ x_f \circlearrowleft x_g &\mapsto \text{ro}_g(x_g). \end{aligned} \quad (41)$$

#### Composition is associative

Let three composable Moore machines  $f$ ,  $g$ , and  $h$  be given. We check that each of the six entries in the definition Def. 16.5 coincide for  $(f \circ g) \circ h$  and  $f \circ (g \circ h)$ .

Clearly,

$$\mathbf{U}_{(f \circ g) \circ h} = \mathbf{U}_f = \mathbf{U}_{f \circ (g \circ h)} \quad (42)$$

and

$$\mathbf{Y}_{(f \circ g) \circ h} = \mathbf{Y}_h = \mathbf{Y}_{f \circ (g \circ h)}. \quad (43)$$

Furthermore,

$$\mathbf{X}_{(f \circ g) \circ h} = (\mathbf{X}_f \circ \mathbf{X}_g) \circ \mathbf{X}_h = \mathbf{X}_f \circ (\mathbf{X}_g \circ \mathbf{X}_h) = \mathbf{X}_{f \circ (g \circ h)} \quad (44)$$

since concatenation of lists is associative. Similarly,

$$\begin{aligned} \mathbf{st}_{(f \circ g) \circ h} &= \mathbf{st}_{f \circ g} \circ \mathbf{st}_h \\ &= (\mathbf{st}_f \circ \mathbf{st}_g) \circ \mathbf{st}_h \\ &= \mathbf{st}_f \circ (\mathbf{st}_g \circ \mathbf{st}_h) \\ &= \mathbf{st}_f \circ \mathbf{st}_{g \circ h} \\ &= \mathbf{st}_{f \circ (g \circ h)}. \end{aligned} \quad (45)$$

Next we show that  $\mathbf{dyn}_{(f \circ g) \circ h} = \mathbf{dyn}_{f \circ (g \circ h)}$ .

On the one hand,

$$\begin{aligned} \mathbf{dyn}_{(f \circ g) \circ h} : \mathbf{U}_f \circ \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h &\rightarrow \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h \\ u \circ x_f \circ x_g \circ x_h &\mapsto \mathbf{dyn}_{f \circ g}(u \circ x_f \circ x_g) \circ \mathbf{dyn}_h(\mathbf{ro}_{f \circ g}(x_f \circ x_g) \circ x_h) \\ &= \mathbf{dyn}_f(u \circ x_f) \circ \mathbf{dyn}_g(\mathbf{ro}_f(x_f) \circ x_g) \circ \mathbf{dyn}_h(\mathbf{ro}_g(x_g) \circ x_h), \end{aligned} \quad (46)$$

while on the other hand

$$\begin{aligned} \mathbf{dyn}_{f \circ (g \circ h)} : \mathbf{U}_f \circ \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h &\rightarrow \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h \\ u \circ x_f \circ x_g \circ x_h &\mapsto \mathbf{dyn}_f(u \circ x_f) \circ \mathbf{dyn}_{g \circ h}(\mathbf{ro}_f(x_f) \circ x_g \circ x_h) \\ &= \mathbf{dyn}_f(u \circ x_f) \circ \mathbf{dyn}_g(\mathbf{ro}_f(x_f) \circ x_g) \circ \mathbf{dyn}_h(\mathbf{ro}_g(x_g) \circ x_h). \end{aligned} \quad (47)$$

So, these two functions are indeed the same.

Finally, we verify that  $\mathbf{ro}_{(f \circ g) \circ h} = \mathbf{ro}_{f \circ (g \circ h)}$ :

$$\begin{aligned} \mathbf{ro}_{(f \circ g) \circ h} : \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h &\rightarrow \mathbf{Y}_h \\ x_f \circ x_g \circ x_h &\mapsto \mathbf{ro}_h(x_h) \end{aligned} \quad (48)$$

while

$$\begin{aligned} \mathbf{ro}_{f \circ (g \circ h)} : \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h &\rightarrow \mathbf{Y}_h, \\ x_f \circ x_g \circ x_h &\mapsto \mathbf{ro}_{g \circ h}(x_g \circ x_h) = \mathbf{ro}_h(x_h). \end{aligned} \quad (49)$$

## The semicategory of Moore machines

Now that we have shown that composition of Moore machines is associative (with our new definition), we can organize Moore machines as a semicategory.

### Definition 16.6 (Moo)

The *semicategory of Moore machines* **Moo** is given by:

1. *Objects*: objects of  $\langle \mathbf{Set} \rangle$ .
2. *Morphisms*: A morphism from **U** to **Y** is a Moore machine

$$\langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \mathbf{dyn}, \mathbf{ro}, \mathbf{st} \rangle, \quad (50)$$

where

- ▷ **U**, **X**, **Y** are objects of  $\langle \mathbf{Set} \rangle$ ;
- ▷  $\mathbf{dyn} : \mathbf{U} \circ \mathbf{X} \rightarrow \mathbf{X}$  and  $\mathbf{ro} : \mathbf{X} \rightarrow \mathbf{Y}$  are morphisms in  $\langle \mathbf{Set} \rangle$ ;



▷  $st \in X$ .

3. *Composition*: as defined above in this section.

## 16.4. Standard action of Moore machines

Given a Moore machine  $f = \langle \mathbf{U}_f, \mathbf{X}_f, \mathbf{Y}_f, \text{dyn}_f, \text{ro}_f, \text{st}_f \rangle$  we saw that we can use it to transform an infinite sequence of inputs  $u_0, u_1, u_2, \dots$  into an infinite sequence of outputs  $y_0, y_1, y_2, \dots$  using the following recipe

$$\begin{cases} x_{k+1} = \text{dyn}_f(u_k \circ x_k) \\ y_k = \text{ro}_f(x_k). \end{cases} \quad (51)$$

Rephrased mathematically, this means that (51), together with  $f$ , defines a function

$$\text{act}_f : \text{Stream } \mathbf{U}_f \rightarrow \text{Stream } \mathbf{Y}_f, \quad (52)$$

which takes any sequence of elements of  $\mathbf{U}_f$  and maps it to a corresponding sequence of elements of  $\mathbf{Y}_f$ . We call this the *standard action* of  $f$  on sequences.

**Remark 16.7.** One way to think about the function  $\text{act}_f$  is to imagine it being calculated in two steps.

1. Given a sequence  $u = u_0, u_1, u_2, \dots$ , there is a unique solution  $s = s_0, s_1, s_2, \dots$  of the recursion  $x_{k+1} = \text{dyn}_f(u_k, x_k)$  with  $s_0 = \text{st}_f$ . (Clearly, this solution can be computed iteratively.)
2. Given the solution  $s$ , the sequence  $y = \text{act}_f(u)$  is simply  $y = s \circ \text{ro}_f$ .

**Example 16.8.** Consider a Moore machine  $f$  with  $\mathbf{U} = \mathbf{X} = \mathbf{Y} = \langle \mathbb{N} \rangle$  and let

$$\begin{aligned} \text{dyn}_f : \mathbf{U} \circ \mathbf{X} &\rightarrow \mathbf{X} \\ \langle m \rangle \circ \langle n \rangle &\mapsto \langle m + n \rangle \end{aligned} \quad (53)$$

and

$$\begin{aligned} \text{ro}_f : \mathbf{X} &\rightarrow \mathbf{Y} \\ \langle n \rangle &\mapsto \langle n + 1 \rangle \end{aligned} \quad (54)$$

and  $\text{st} = \langle 0 \rangle$ .

Given a sequence of inputs of the form  $u = \langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 4 \rangle, \langle 5 \rangle, \dots$ , what are the first five entries of the corresponding sequence of outputs?

To compute this, let us first calculate the first five entries of the sequence of states  $s = s_0, s_1, s_2, \dots$  that solves (51). We have

$$s_0 = \text{st} = \langle 0 \rangle, \quad (55)$$

$$s_1 = \langle 1 + 0 \rangle = \langle 1 \rangle, \quad (56)$$

$$s_2 = \langle 2 + 1 \rangle = \langle 3 \rangle, \quad (57)$$

$$s_3 = \langle 3 + 3 \rangle = \langle 6 \rangle, \quad (58)$$

$$s_4 = \langle 4 + 6 \rangle = \langle 10 \rangle. \quad (59)$$

Now, applying  $\text{ro}_f$  to the entries of this sequence of state, we obtain the first five entries of the output sequence:

$$y_0 = \langle 0 + 1 \rangle = \langle 1 \rangle, \quad (60)$$

$$y_1 = \langle 1 + 1 \rangle = \langle 2 \rangle, \quad (61)$$

$$y_2 = \langle 3 + 1 \rangle = \langle 4 \rangle, \quad (62)$$

$$y_3 = \langle 6 + 1 \rangle = \langle 7 \rangle, \quad (63)$$

$$y_4 = \langle 10 + 1 \rangle = \langle 11 \rangle. \quad (64)$$

**Graded exercise E.11** (ComputingMooreActions)

Consider the Moore machine  $f$  with  $\mathbf{U} = \mathbf{X} = \mathbf{Y} = \langle \mathbb{Z} \rangle$ ,  $\mathbf{st} = \langle 0 \rangle$ , and

$$\begin{aligned} \text{dyn}_f : \mathbf{U} \circlearrowleft \mathbf{X} &\rightarrow \mathbf{X} \\ \langle m \rangle \circlearrowleft \langle n \rangle &\mapsto \langle n - m \rangle, \end{aligned} \quad (65)$$

$$\begin{aligned} \text{ro}_f : \mathbf{X} &\rightarrow \mathbf{Y} \\ \langle n \rangle &\mapsto \langle n + 2 \rangle. \end{aligned} \quad (66)$$

Given a stream  $u \in \text{Stream } \mathbf{U}$  of inputs of the form  $u = \langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 4 \rangle, \langle 5 \rangle, \dots$ , compute the first five entries of the corresponding stream of outputs for the following two actions:

1. The standard action defined via the recursion

$$\begin{cases} x_{k+1} = \text{dyn}_f(u_k \circlearrowleft x_k) \\ y_k = \text{ro}_f(x_k) \end{cases} \quad (67)$$

with  $x_0 = \mathbf{st}$ .

2. The action defined via the recursion

$$\begin{cases} x_{k+1} = \text{dyn}_f(u_k \circlearrowleft x_k) \\ \tilde{x}_{k+1} = \text{dyn}_f(u_k \circlearrowleft x_{k+1}) \\ y_k = \text{ro}_f(\tilde{x}_k) \end{cases} \quad (68)$$

with  $x_0 = \tilde{x}_0 = \mathbf{st}$ .

## Compositionality

**Proposition 16.9.** The standard action of Moore machines on signal sequences is compatible with Moore machine composition in the sense that

$$\text{act}_{f \circ g} = \text{act}_f \circ \text{act}_g \quad (69)$$

for any composable Moore machines  $f$  and  $g$ .

*Proof.* Suppose we have Moore machines

$$f = \langle \mathbf{U}_f, \mathbf{X}_f, \mathbf{Y}_f, \text{dyn}_f, \text{ro}_f, \mathbf{st}_f \rangle \quad (70)$$

and

$$g = \langle \mathbf{U}_g, \mathbf{X}_g, \mathbf{Y}_g, \text{dyn}_g, \text{ro}_g, \mathbf{st}_g \rangle, \quad (71)$$

with  $\mathbf{Y}_f = \mathbf{U}_g$ .

We will first calculate the right-hand side of (69), following the procedure of Remark 16.7.

Given a sequence  $u \in \text{Stream } \mathbf{U}_f$ , to calculate  $\text{act}_f(u)$  we first map  $u$  to the unique solution  $r_u \in \text{Stream } \mathbf{X}_f$  of the recursion

$$x_{k+1} = \text{dyn}_f(u_k, x_k) \quad k \in \mathbb{N}, \quad (72)$$

such that  $r_u(0) = \mathbf{st}_f$ . Then  $\text{act}_f(u) = r_u \circ \text{ro}_f$ , an element of  $\text{Stream } \mathbf{Y}_f = \text{Stream } \mathbf{U}_g$ .

Next, to calculate  $(\text{act}_f \circ \text{act}_g)(u)$ , we first map  $\text{act}_f(u)$  to the unique solu-

tion  $s_{\text{act}_f(u)} \in \text{Stream } \mathbf{X}_g$  of the recursion

$$x_{k+1} = \text{dyn}_g(\text{act}_f(u_k, x_k)) \quad k \in \mathbb{N} \quad (73)$$

such that  $s_{\text{act}_f(u)}(0) = \text{st}_g$ , and then we have

$$(\text{act}_f \circ \text{act}_g)(u) = s_{\text{act}_f(u)} \circ \text{ro}_g. \quad (74)$$

We calculate the left-hand side of (69). Given  $u \in \text{Stream } \mathbf{U}_f$ , we map it to the unique solution  $q_u \in \text{Stream } \mathbf{X}_{f \circ g} = \text{Stream } (\mathbf{X}_f \circ \mathbf{X}_g)$  of the recursion

$$x_{k+1} = \text{dyn}_{f \circ g}(u_k, x_k) \quad k \in \mathbb{N} \quad (75)$$

with  $q_u(0) = \text{st}_f \circ \text{st}_g$ . Using the definition of  $\text{dyn}_{f \circ g}$  and the fact that we can write any  $x_k \in \mathbf{X}_f \circ \mathbf{X}_g$  as

$$x_k = x_k^f \circ x_k^g \quad \text{with } x_k^f \in \mathbf{X}_f, x_k^g \in \mathbf{X}_g, \quad (76)$$

we can rewrite (75) as

$$x_{k+1}^f \circ x_{k+1}^g = \text{dyn}_f(u_k, x_k^f) \circ \text{dyn}_g(\text{ro}_f(x_k^f), x_k^g) \quad k \in \mathbb{N}. \quad (77)$$

Now observe that the sequence  $r_u(k) \circ s_{\text{act}_f(u)}(k)$  solves this recursion: by definition  $r_u$  solves the recursion

$$x_{k+1} = \text{dyn}_f(u_k, x_k) \quad k \in \mathbb{N}, \quad (78)$$

and, recalling that  $\text{act}_f(u)_k = \text{ro}_f(r_u(k))$ , we see that  $s_{\text{act}_f(u)}$  solves the recursion

$$x_{k+1} = \text{dyn}_g(\text{ro}_f(r_u(k)), x_k) \quad k \in \mathbb{N}. \quad (79)$$

Since  $r_u(0) \circ s_{\text{act}_f(u)}(0) = \text{st}_f \circ \text{st}_g$ , this implies that the unique solution  $q_u$  above is precisely  $q_u(k) = r_u(k) \circ s_{\text{act}_f(u)}(k)$ .

Finally, we have

$$\text{act}_{f \circ g}(u) = q_u \circ \text{ro}_{f \circ g}. \quad (80)$$

Evaluating at any  $k \in \mathbb{N}$  we find

$$(q_u \circ \text{ro}_{f \circ g})(k) = \text{ro}_{f \circ g}(q_u(k)) \quad (81)$$

$$= \text{ro}_{f \circ g}(r_u(k) \circ s_{\text{act}_f(u)}(k)) \quad (82)$$

$$= \text{ro}_g(s_{\text{act}_f(u)}(k)) \quad (83)$$

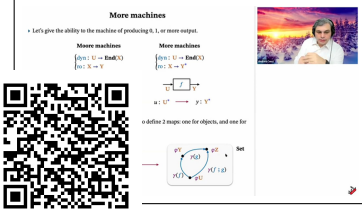
$$= (s_{\text{act}_f(u)} \circ \text{ro}_g)(k). \quad (84)$$

Comparing with (74), we conclude that

$$\text{act}_{f \circ g}(u) = (\text{act}_f \circ \text{act}_g)(u). \quad (85)$$

□

Watch: *Semi-category action* (3 minutes).



## 16.5. Semicategory actions

In this section we generalize from actions of a semigroup to actions of a semicategory, using the example of Moore machines acting on signal sequences.

To motivate our story, let us first consider Moore machines of the form

$$\langle \mathbf{U}, \mathbf{X}, \mathbf{U}, \text{dyn}, \text{ro}, \text{st} \rangle \quad (86)$$

where the input and output sets are equal. In other words, we are considering the set  $\text{Hom}_{\mathbf{Moo}}(\mathbf{U}; \mathbf{U})$ , which is a semigroup under morphism composition.

In Section 16.4 we defined a standard action which associates to every morphism  $f \in \text{Hom}_{\mathbf{Moo}}(\mathbf{U}; \mathbf{U})$  a function

$$\text{act}_f : \text{Stream } \mathbf{U} \rightarrow \text{Stream } \mathbf{U}. \quad (87)$$

This action defines a function

$$\text{act}_\bullet : \text{Hom}_{\mathbf{Moo}}(\mathbf{U}; \mathbf{U}) \rightarrow \text{End}(\text{Stream } \mathbf{U}) \quad (88)$$

where

$$\text{act}_\bullet(f) = \text{act}_f. \quad (89)$$

And this function  $\text{act}_\bullet$  is in fact a semigroup morphism, and so a semigroup action, since we proved in Prop. 16.9 that

$$\text{act}_{f \circ g} = \text{act}_f \circ \text{act}_g. \quad (90)$$

(Compare with Def. 8.2 of semigroup action.)

Now consider the general situation of Moore machines acting on signals. Given any Moore machine of the general form

$$f = \langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle \quad (91)$$

(the input and output spaces are no longer necessarily equal) we again have an associated function on signals

$$\text{act}_f : \text{Stream } \mathbf{U} \rightarrow \text{Stream } \mathbf{Y}. \quad (92)$$

We can assemble this data as a family of functions (all which we call  $\text{act}_\bullet$ )

$$\text{act}_\bullet : \text{Hom}_{\mathbf{Moo}}(\mathbf{U}; \mathbf{Y}) \rightarrow \text{Hom}_{\mathbf{Set}}(\text{Stream } \mathbf{U}; \text{Stream } \mathbf{Y}), \quad (93)$$

where  $\mathbf{U}$  and  $\mathbf{Y}$  range over all objects of  $\mathbf{Moo}$ . Or, if you will,

$$\text{act}_\bullet : \text{Mor}_{\mathbf{Moo}} \rightarrow \text{Mor}_{\mathbf{Set}}. \quad (94)$$

From Prop. 16.9 we have that this function is compatible with the composition operations in  $\mathbf{Moo}$  and  $\mathbf{Set}$ :

$$\text{act}_\bullet(f \circ g) = \text{act}_\bullet(f) \circ \text{act}_\bullet(g). \quad (95)$$

Note that the sets  $\text{Stream } \mathbf{U}$  and  $\text{Stream } \mathbf{Y}$  involved in the right-hand side of (93) depend on the objects  $\mathbf{U}$  and  $\mathbf{Y}$  on the left-hand side. We will encode this also with a function

$$\begin{aligned} \text{act} : \text{Ob}_{\mathbf{Moo}} &\rightarrow \text{Ob}_{\mathbf{Set}}, \\ \mathbf{U} &\mapsto \text{Stream } \mathbf{U}, \end{aligned} \quad (96)$$

in which case (93) becomes

$$\text{act}_\bullet : \text{Hom}_{\text{Moo}}(\mathbf{U}; \mathbf{Y}) \rightarrow \text{Hom}_{\text{Set}}(\text{act}_\bullet \mathbf{U}; \text{act}_\bullet \mathbf{Y}). \quad (97)$$

In summary, we have reformulated Moore machine actions as consisting of a pair of functions

$$\text{act}_\bullet : \text{Mor}_{\text{Moo}} \rightarrow \text{Mor}_{\text{Set}} \quad \text{and} \quad \text{act}_\bullet : \text{Ob}_{\text{Moo}} \rightarrow \text{Ob}_{\text{Set}} \quad (98)$$

which work together as in (97) and such that  $\text{act}_\bullet$  is compatible with composition as in (95).

Now we formalize this situation as a general definition.

**Definition 16.10** (Semicategory action)

A *semicategory action* of a semicategory  $\mathbf{C}$  is

Constituents

1. A map  $\text{act}_\bullet : \text{Ob}_{\mathbf{C}} \rightarrow \text{Ob}_{\text{Set}}$ ;
2. For every two objects  $X, Y \in \text{Ob}_{\mathbf{C}}$ , a map

$$\text{act}_\bullet : \text{Hom}_{\mathbf{C}}(X; Y) \rightarrow \text{Hom}_{\text{Set}}(\text{act}_\bullet(X); \text{act}_\bullet(Y)). \quad (99)$$

Conditions

- ▷ For all composable morphisms  $f$  and  $g$ ,

$$\text{act}_\bullet(f \circ g) = \text{act}_\bullet(f) \circ \text{act}_\bullet(g). \quad (100)$$

The compatibility of the action with composition is illustrated in Fig. 3.

For reference, let us also fix the following definition.

**Definition 16.11** (Standard action of Moore machines)

The *standard action of Moore machines* on sequences is given by

$$\begin{aligned} \text{act}_\bullet : \text{Ob}_{\text{Moo}} &\rightarrow \text{Ob}_{\text{Set}}, \\ \mathbf{U} &\mapsto \text{Stream } \mathbf{U}, \end{aligned} \quad (101)$$

on the level of objects, and on the level of morphisms, the functions

$$\text{act}_\bullet : \text{Hom}_{\text{Moo}}(\mathbf{U}; \mathbf{Y}) \rightarrow \text{Hom}_{\text{Set}}(\text{Stream } \mathbf{U}; \text{Stream } \mathbf{Y}) \quad (102)$$

are defined via the recursion equations

$$\begin{cases} x_{k+1} = \text{dyn}_f(u_k \circ x_k) \\ y_k = \text{ro}_f(x_k) \end{cases} \quad (103)$$

as in Section 16.4.

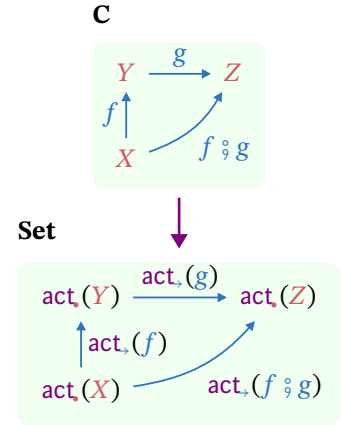


Figure 3.

## 16.6. Different actions of Moore machines

Moore machines need not necessarily act in the standard way given in Def. 16.11. In this section, we illustrate other actions of Moore machines.

### Changing how morphisms act

Recall that, given a Moore machine  $f = \langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle$ , the standard action defined a function

$$\text{Stream } \mathbf{U} \rightarrow \text{Stream } \mathbf{Y} \quad (104)$$

via solving the recursion equations

$$\begin{cases} x_{k+1} = \text{dyn}_f(u_k \circ x_k) \\ y_k = \text{ro}_f(x_k). \end{cases} \quad (105)$$

One way define other actions is by changing (105) to some other recipe. It is necessary then to check that the resulting action really satisfies the condition of Def. 16.10.

For example, consider the recursion equations

$$\begin{cases} x_{k+1} = \text{dyn}_f(u_k \circ x_k) \\ \tilde{x}_{k+1} = \text{dyn}_f(u_k \circ \tilde{x}_{k+1}) \\ y_k = \text{ro}_f(\tilde{x}_k). \end{cases} \quad (106)$$

Given  $f = \langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle$  we can use these to define an action of  $f$  by interpreting the variables  $x_k$  and  $\tilde{x}_k$  (for all  $k \in \mathbb{N}$ ) both as denoting elements of  $\mathbf{X}$ , and setting  $x_0 = \tilde{x}_0 = \text{st}$ . The rest is analogous to how we defined the standard action: given a sequence  $u = u_0, u_1, u_2, \dots$  we can iteratively compute from it sequences  $x = x_0, x_1, x_2, \dots$  and  $\tilde{x} = \tilde{x}_0, \tilde{x}_1, \tilde{x}_2, \dots$ , and then the output sequence  $y = y_0, y_1, y_2, \dots$  is defined by  $y_k = \text{ro}_f(\tilde{x}_k)$ .

**Example 16.12.** Consider the same Moore machine  $f$  as in Example 16.8, but now consider the action that uses (106), instead of the standard action. If we give the same input sequence as we did in Example 16.8, what does the corresponding output sequence look like now?

We calculate the first five entries of the output sequence. First, recall that  $\mathbf{U} = \mathbf{X} = \mathbf{Y} = \langle \mathbb{N} \rangle$  and

$$\begin{aligned} \text{dyn}_f : \mathbf{U} \circ \mathbf{X} &\rightarrow \mathbf{X} \\ \langle m \rangle \circ \langle n \rangle &\mapsto \langle m + n \rangle \end{aligned} \quad (107)$$

and

$$\begin{aligned} \text{ro}_f : \mathbf{X} &\rightarrow \mathbf{Y} \\ \langle n \rangle &\mapsto \langle n + 1 \rangle \end{aligned} \quad (108)$$

and  $\text{st} = \langle 0 \rangle$ .

The input sequence has the form  $u = \langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 4 \rangle, \langle 5 \rangle, \dots$ . We first compute the first five entries of the sequences of states  $s = s_0, s_1, s_2, \dots$  and  $\tilde{s} = \tilde{s}_0, \tilde{s}_1, \tilde{s}_2, \dots$  that solve (106):

$$s_0 = \text{st} = \langle 0 \rangle, \quad (109)$$

$$s_1 = \langle 1 + 0 \rangle = \langle 1 \rangle, \quad (110)$$

$$s_2 = \langle 2 + 1 \rangle = \langle 3 \rangle, \quad (111)$$

$$s_3 = \langle 3 + 3 \rangle = \langle 6 \rangle, \quad (112)$$

$$s_4 = \langle 4 + 6 \rangle = \langle 10 \rangle \quad (113)$$

and

$$\tilde{s}_0 = st = \langle 0 \rangle, \quad (114)$$

$$\tilde{s}_1 = \langle 1 + 1 \rangle = \langle 2 \rangle, \quad (115)$$

$$\tilde{s}_2 = \langle 3 + 2 \rangle = \langle 5 \rangle, \quad (116)$$

$$\tilde{s}_3 = \langle 6 + 3 \rangle = \langle 9 \rangle, \quad (117)$$

$$\tilde{s}_4 = \langle 10 + 4 \rangle = \langle 14 \rangle. \quad (118)$$

Applying  $\text{rof}$  to the entries the sequence  $\tilde{x}$ , the first five entries of the output sequence are:

$$y_0 = \langle 0 + 1 \rangle = \langle 1 \rangle, \quad (119)$$

$$y_1 = \langle 2 + 1 \rangle = \langle 3 \rangle, \quad (120)$$

$$y_2 = \langle 5 + 1 \rangle = \langle 6 \rangle, \quad (121)$$

$$y_3 = \langle 9 + 1 \rangle = \langle 10 \rangle, \quad (122)$$

$$y_4 = \langle 14 + 1 \rangle = \langle 15 \rangle. \quad (123)$$

**Lemma 16.13.** The recursion equations (106) do indeed define a semicategory action of Moore machines in the sense of Def. 16.10.

## Changing the signal spaces

The action defined above using (106) differs from the standard action of Section 16.4 in terms of how morphisms act; it defines a different map

$$\text{Mor}_{\text{Moo}} \rightarrow \text{Mor}_{\text{Set}}. \quad (124)$$

However, on the level of objects, the action stayed the same:

$$\begin{aligned} \text{Ob}_{\text{Moo}} &\rightarrow \text{Ob}_{\text{Set}} \\ \mathbf{U} &\mapsto \text{Stream } \mathbf{U} \end{aligned} \quad (125)$$

Here we define an action which is different at the level of objects. Namely, we now make a different choice for a map

$$\text{act}_\cdot : \text{Ob}_{\text{Moo}} \rightarrow \text{Ob}_{\text{Set}}. \quad (126)$$

Instead of mapping any object  $X$  to the set of signals of the type  $\text{Stream } X$ , which are sequences of elements of  $X$ , our new action maps  $X$  to the set  $\text{List } X$  of lists of elements of  $X$ . In other words, we model Moore machines now as taking finite lists as inputs, rather than infinite sequences.

On the level of morphisms, we will keep the recipe of the original standard action of Moore machines.

### Definition 16.14

We define the standard action of Moore machines on lists as follows. The action on object is

$$\begin{aligned} \text{act}_\cdot : \text{Ob}_{\text{Moo}} &\rightarrow \text{Ob}_{\text{Set}}, \\ \mathbf{U} &\mapsto \text{List } \mathbf{U}. \end{aligned} \quad (127)$$

On the level of morphisms, the functions

$$\text{act}_\cdot : \text{Hom}_{\text{Moo}}(\mathbf{U}; \mathbf{Y}) \rightarrow \text{Hom}_{\text{Set}}(\text{List } \mathbf{U}; \text{List } \mathbf{Y}) \quad (128)$$



are defined via the recursion equations

$$\begin{cases} x_{k+1} = \text{dyn}_f(u_k \circ x_k) \\ y_k = \text{ro}_f(x_k) \end{cases} \quad (129)$$

analogously as for the standard action on sequences. The only special cases are when the empty list  $[]_{\mathbf{U}}$  is the input: then we set

$$\text{act}_f(f)([]_{\mathbf{U}}) = [\text{ro}_f(\text{st}_f)]_{\mathbf{Y}} \quad (130)$$

for any Moore machine  $f = \langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle$ .

**Example 16.15.** Consider the same Moore machine  $f$  as in Example 16.8 and Example 16.12, namely with  $\mathbf{U} = \mathbf{X} = \mathbf{Y} = \langle \mathbb{N} \rangle$  and

$$\begin{aligned} \text{dyn}_f : \mathbf{U} \circ \mathbf{X} &\rightarrow \mathbf{X} \\ \langle m \rangle \circ \langle n \rangle &\mapsto \langle m + n \rangle \end{aligned} \quad (131)$$

and

$$\begin{aligned} \text{ro}_f : \mathbf{X} &\rightarrow \mathbf{Y} \\ \langle n \rangle &\mapsto \langle n + 1 \rangle \end{aligned} \quad (132)$$

and  $\text{st} = \langle 0 \rangle$ .

If we take the list

$$[\langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 4 \rangle, \langle 5 \rangle]_{\mathbf{U}} \quad (133)$$

as input, then the corresponding output is

$$[\langle 0 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 6 \rangle, \langle 10 \rangle]_{\mathbf{Y}}, \quad (134)$$

analogous to Example 16.8.

If we take  $[]_{\mathbf{U}}$  as input, then the output is  $[\langle 0 \rangle]_{\mathbf{Y}}$ .

**Lemma 16.16.** Def. 16.14 defines a semicategory action.

## 16.7. More machines

A Moore machine outputs 1 element at each step; what if a machine was able to output more than one or zero output? We will call these *More machines*.

We define their dynamics and readout to be of the form

$$\begin{cases} \text{dyn} : \mathbf{U} \circledast \mathbf{X} \rightarrow \mathbf{X}, \\ \text{ro} : \mathbf{X} \rightarrow \text{List } \mathbf{Y}, \end{cases} \quad (135)$$

where the output set is not just  $\mathbf{Y}$  but  $\text{List } \mathbf{Y}$ , the set of finite list of elements of  $\mathbf{Y}$ . In other words, for a given state, the machine can produce zero or more outputs in the form of a list.

We'll specify More machines as a tuple  $\langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle$ , like we did for Moore machines.

### Composition

Given More machines  $f = \langle \mathbf{U}_f, \mathbf{X}_f, \mathbf{Y}_f, \text{dyn}_f, \text{ro}_f, \text{st}_f \rangle$  and  $g = \langle \mathbf{U}_g, \mathbf{X}_g, \mathbf{Y}_g, \text{dyn}_g, \text{ro}_g, \text{st}_g \rangle$  with  $\mathbf{Y}_f = \mathbf{U}_g$ , their composition is the More machine with

$$\begin{aligned} \mathbf{U}_{f \circledast g} &= \mathbf{U}_f, \\ \mathbf{X}_{f \circledast g} &= \mathbf{X}_f \circledast \mathbf{X}_g, \\ \text{st}_{f \circledast g} &= \text{st}_f \circledast \text{st}_g, \\ \mathbf{Y}_{f \circledast g} &= \mathbf{Y}_g. \end{aligned} \quad (136)$$

The dynamics of the composite  $f \circledast g$  is

$$\begin{aligned} \text{dyn}_{f \circledast g} : \mathbf{U}_f \circledast \mathbf{X}_f \circledast \mathbf{X}_g &\rightarrow \mathbf{X}_f \circledast \mathbf{X}_g, \\ u \circledast x_f \circledast x_g &\mapsto \text{dyn}_f(u \circledast x_f) \circledast \text{dyn}_g(y_f[n] \circledast \text{dyn}_g(y_f[n-1] \circledast \dots \text{dyn}_g(y_f[1] \circledast x_g) \dots)). \end{aligned} \quad (137)$$

where  $y_f = \text{ro}_f(x_f) \in \text{List } \mathbf{Y}_f$  and  $n$  is its length.

The readout of  $f \circledast g$  is

$$\begin{aligned} \text{ro}_{f \circledast g} : \mathbf{X}_f \circledast \mathbf{X}_g &\rightarrow \text{List } \mathbf{Y}_g, \\ x_f \circledast x_g &\mapsto \text{ro}_g(x_g). \end{aligned} \quad (138)$$

#### Definition 16.17 (Mor)

The *semicategory of More machines* **Mor** is given by:

1. *Objects*: objects of  $\langle \mathbf{Set} \rangle$ .
2. *Morphisms*: A morphism is a tuple

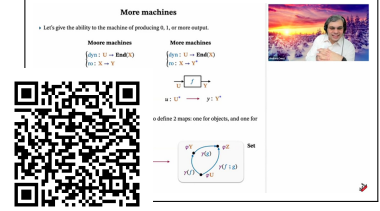
$$f = \langle \mathbf{U}_f, \mathbf{X}_f, \mathbf{Y}_f, \text{dyn}_f, \text{ro}_f, \text{st}_f \rangle, \quad (139)$$

where:

- ▷  $\mathbf{U}_f, \mathbf{X}_f, \mathbf{Y}_f$  are objects of  $\langle \mathbf{Set} \rangle$ ;
- ▷  $\text{st}_f \in \mathbf{X}_f$ ;
- ▷  $\text{dyn}_f : \mathbf{U}_f \circledast \mathbf{X}_f \rightarrow \mathbf{X}_f$ ;
- ▷  $\text{ro}_f : \mathbf{X}_f \rightarrow \text{List } \mathbf{Y}_f$ .

3. *Composition of morphisms*: Composition is given by (137) and (138).

Watch: *More machines* (5 minutes).



## Examples

**Example 16.18** (Duplicator). We consider an example of a More machine  $d$  which we call a *duplicator*. Practically, this machine takes an input, and duplicates it. The machine is written as

$$\langle \mathbf{U}_d, \mathbf{X}_d, \mathbf{Y}_d, \text{dyn}_d, \text{ro}_d, \text{st}_d \rangle, \quad (140)$$

where  $\mathbf{U}_d = \mathbf{X}_d = \mathbf{Y}_d$ , and

$$\begin{aligned} \text{dyn}_d : \mathbf{U}_d \circledast \mathbf{X}_d &\rightarrow \mathbf{X}_d, \\ u \circledast x &\mapsto u, \end{aligned} \quad (141)$$

$$\begin{aligned} \text{ro}_d : \mathbf{X}_d &\rightarrow \text{List } \mathbf{Y}_d, \\ x &\mapsto [x, x]_{\mathbf{Y}_d}, \end{aligned} \quad (142)$$

and  $\text{st} = \langle \rangle \in \mathbf{X}_d$ .

**Example 16.19** (Discarder). Here is an example of a More machine  $e$  which we call a *discarder*. Practically, this machine discards every other input. The machine is written as:

$$\langle \mathbf{U}_e, \mathbf{X}_e, \mathbf{Y}_e, \text{dyn}_e, \text{ro}_e, \text{st}_e \rangle, \quad (143)$$

where  $\mathbf{X}_e = \langle \{\perp, \top\} \rangle \circledast \mathbf{U}_e$ , and

$$\begin{aligned} \text{dyn}_e : \mathbf{U}_e \circledast \langle \{\perp, \top\} \rangle \circledast \mathbf{U}_e &\rightarrow \langle \{\perp, \top\} \rangle \circledast \mathbf{U}_e \\ u \circledast \langle \perp \rangle \circledast x &\mapsto \langle \top \rangle \circledast u \\ u \circledast \langle \top \rangle \circledast x &\mapsto \langle \perp \rangle \circledast u, \end{aligned} \quad (144)$$

$$\begin{aligned} \text{ro}_e : \langle \{\perp, \top\} \rangle \circledast \mathbf{U}_e &\rightarrow \text{List } \mathbf{Y}_e \\ \langle \perp \rangle \circledast x &\mapsto []_{\mathbf{Y}_e} \\ \langle \top \rangle \circledast x &\mapsto [x]_{\mathbf{Y}_e} \end{aligned} \quad (145)$$

and  $\text{st}_e = \langle \perp \rangle \circledast \langle x \rangle \in \mathbf{X}_e$ , for an arbitrary  $x \in \mathbf{U}_e$ .

**Example 16.20** (Terminator). We describe a More machine  $t$  which we call a *terminator*. Practically, this machine terminates any input, outputting an empty list. The machine is written as

$$\langle \mathbf{U}_t, \mathbf{X}_t, \mathbf{Y}_t, \text{dyn}_t, \text{ro}_t, \text{st}_t \rangle, \quad (146)$$

where:

$$\begin{aligned} \text{dyn}_t : \mathbf{U}_t \circledast \mathbf{X}_t &\rightarrow \mathbf{X}_t, \\ u \circledast x &\mapsto x, \end{aligned} \quad (147)$$

$$\begin{aligned} \text{ro}_t : \mathbf{X}_t &\rightarrow \text{List } \mathbf{Y}_t, \\ \mathbf{X}_t &\mapsto []_{\mathbf{Y}_t}, \end{aligned} \quad (148)$$

and  $\text{st}_t \in \mathbf{X}_t$  can be any element.

## 16.8. LTI systems

We now define LTI systems.

### Definition 16.21 (LTI System)

A *linear time-invariant dynamical (LTI) system*, in a so-called state-space representation, is specified by real vector spaces  $\mathbf{U} = \mathbb{R}^l$  (input space),  $\mathbf{Y} = \mathbb{R}^m$  (output space), and  $\mathbf{X} = \mathbb{R}^n$  (state space), along with a system of equations of the form

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \quad (149)$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t), \quad (150)$$

and an *initial state*  $\mathbf{st} \in \mathbf{X}$ , where  $t \in \mathbb{R}_{\geq 0}$ ,  $\mathbf{u}(t) \in \mathbf{U}$ ,  $\mathbf{y}(t) \in \mathbf{Y}$ ,  $\mathbf{x}(t) \in \mathbf{X}$ , and where  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\mathbf{C}$ ,  $\mathbf{D}$  are real matrices of appropriate dimension.

**Remark 16.22.** Def. 16.21 includes the particular case of matrices with 0 dimension. For instance, in a system with  $\mathbf{U} = \mathbb{R}^l$  and  $\mathbf{Y} = \mathbb{R}^m$ , we could have  $\mathbf{A} \in \mathbb{R}^{0 \times 0}$  (in other words, no state). This would imply  $\mathbf{B} \in \mathbb{R}^{0 \times l}$ ,  $\mathbf{C} \in \mathbb{R}^{m \times 0}$ , and  $\mathbf{D} \in \mathbb{R}^{m \times l}$ . Matrices with zero rows and/or zero columns multiply exactly as other matrices. For instance, the multiplication of a  $\mathbb{R}^{0 \times 0}$  matrix with a  $\mathbb{R}^{0 \times l}$  matrix, will return a  $\mathbb{R}^{0 \times l}$  matrix.

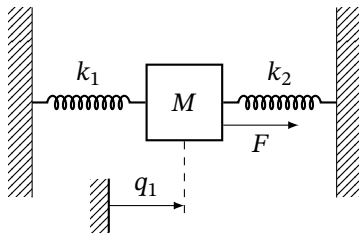
We compactly refer to an LTI system by writing it as a tuple  $\langle \mathbf{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ .

The equations (149) describe the dynamics. The equations (150) describe the output, or, one might say, the variables that are “exposed” or externally visible.

The matrix  $\mathbf{D}$  is called the *feedthrough term*. We will call an LTI system *proper* if  $\mathbf{D} = \mathbf{0}^{l \times m}$ .

**Remark 16.23.** When using state-space LTI systems as models, it is typical that the equations (149) are chosen using physical laws and first principles reasoning, while the matrices  $\mathbf{C}$  and  $\mathbf{D}$  in (150) are rather chosen based on what information from our model is explicitly relevant or accessible.

**Example 16.24.** Consider a mass  $m$  lying on a frictionless surface, and attached to two springs as depicted in Example 16.24.



The coordinate  $q(t)$  describes position of the mass along one horizontal dimension,  $F$  denotes a force that is applied to  $m$  in that horizontal direction, and  $k_1$  and  $k_2$  are the spring constants of the respective springs.

The dynamics of the position coordinate  $q$  as a function of time is described by the differential equation

$$(k_1 + k_2)q + m\ddot{q} = F. \quad (151)$$

We may rewrite this as a proper LTI system in a state-space representation by

choosing the state-variable to be

$$\mathbf{x} = \begin{bmatrix} q(t) \\ \dot{q}(t) \end{bmatrix}. \quad (152)$$

Then

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 \\ \frac{-(k_1+k_2)}{m} & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} F \quad (153)$$

describes the dynamics, and as output we might choose

$$\mathbf{y} = \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \end{bmatrix} F. \quad (154)$$

## Equivalent LTI systems

**Definition 16.25** (Equivalence of LTI systems)

Two systems  $\langle \mathbf{st}_f, \mathbf{A}_f, \mathbf{B}_f, \mathbf{C}_f, \mathbf{D}_f \rangle$  and  $\langle \mathbf{st}_g, \mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g \rangle$  are *equivalent* if and only if there exists an invertible linear transformation  $\mathbf{x}_g(t) = \mathbf{T}\mathbf{x}_f(t)$  such that

$$\mathbf{A}_g = \mathbf{T}\mathbf{A}_f\mathbf{T}^{-1}, \mathbf{B}_g = \mathbf{T}\mathbf{B}_f, \mathbf{C}_g = \mathbf{C}_f\mathbf{T}^{-1}, \mathbf{D}_g = \mathbf{D}_f, \mathbf{st}_g = \mathbf{T}\mathbf{st}_f. \quad (155)$$

$\mathbf{T}$  is called an *equivalence transformation*.

We think of equivalent LTI-systems as different ways of specifying what is essentially “the same system”. What is different in each specification is only different by a change of coordinates.

## Category of LTI systems

We define a category of LTI systems **LTI**.

**Definition 16.26** (Category LTI)

The category **LTI** of LTI systems is defined by:

1. *Objects*: natural numbers.
2. *Morphisms*: A morphism in **LTI** from  $l \in \mathbb{N}$  to  $m \in \mathbb{N}$  is a continuous time LTI system  $\langle \mathbf{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ . Note that the dimensions of the input/output are implicit.
3. *Composition*: Given morphisms  $f : a \rightarrow b$  and  $g : b \rightarrow c$ , described by the LTI systems

$$\begin{aligned} &\langle \mathbf{st}_f, \mathbf{A}_f, \mathbf{B}_f, \mathbf{C}_f, \mathbf{D}_f \rangle \\ &\langle \mathbf{st}_g, \mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g \rangle, \end{aligned} \quad (156)$$

their composition  $(f \circ g) : a \rightarrow c$  is the LTI system  $\langle \mathbf{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ , where

$$\begin{aligned} \mathbf{st} &= \begin{bmatrix} \mathbf{st}_f \\ \mathbf{st}_g \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0} \\ \mathbf{B}_g\mathbf{C}_f & \mathbf{A}_g \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_f \\ \mathbf{B}_g\mathbf{D}_f \end{bmatrix}, \\ \mathbf{C} &= [\mathbf{D}_g\mathbf{C}_f \quad \mathbf{C}_g], \quad \mathbf{D} = \mathbf{D}_g\mathbf{D}_f. \end{aligned} \quad (157)$$

4. *Identities*: the identity for  $l \in \mathbb{N}$  is the system  $\langle \mathbf{0}^{0 \times 1}, \mathbf{0}^{0 \times 0}, \mathbf{0}^{0 \times l}, \mathbf{0}^{m \times 0}, \mathbf{0}^{m \times l} \rangle$ .

**Remark 16.27.** Again, all of this works with matrices with zero rows and/or zero columns. Let’s see practically how via a simple instance. Consider the system  $f : a \rightarrow b$  given by  $\langle \mathbf{st}_f, \mathbf{A}_f, \mathbf{B}_f, \mathbf{C}_f, \mathbf{D}_f \rangle$  and the system  $g : b \rightarrow c$  given by

$\langle \mathbf{0}^{0 \times 1}, \mathbf{0}^{0 \times 0}, \mathbf{0}^{0 \times b}, \mathbf{0}^{c \times 0}, \mathbf{0}^{c \times b} \rangle$ . Their composition  $f \circ g : a \rightarrow c$  is a system  $\langle \mathbf{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$  with:

$$\mathbf{st} = \begin{bmatrix} \mathbf{st}_f \\ \mathbf{0}^{0 \times 1} \end{bmatrix} = \mathbf{st}_f.$$

How could we write the above equation?  $\mathbf{st}_f$  has  $s$  rows and 1 column, and  $\mathbf{0}^{0 \times 1}$  has 0 rows and 1 column (it has to have 1 column for us to be able to write the above block matrix), allowing us to write the expression as  $\mathbf{st}_f$ . We can now write:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0}^{s \times 0} \\ \mathbf{0}^{0 \times b} \mathbf{C}_f & \mathbf{0}^{0 \times 0} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0}^{s \times 0} \\ \mathbf{0}^{0 \times s} & \mathbf{0}^{0 \times 0} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_f \\ \mathbf{0}^{0 \times s} \end{bmatrix} = \mathbf{A}_f,$$

The above equality teaches us how to manipulate matrices with zero rows and/or zero columns. First, since  $\mathbf{C}_f$  has  $b$  rows and  $s$  columns, the multiplication  $\mathbf{0}^{0 \times b} \mathbf{C}_f$  is well defined and gives the zero matrix with 0 rows and  $s$  columns. Once we realize this, we see that we end up with a matrix made of four block matrices. Given their zero rows/columns, one can then simplify as shown. Similar arguments can be made for the other matrices resulting from the composition.

**Exercise 32.** Prove that **LTI** is indeed a category.

*See solution on page 300.*

**Example 16.28.** Consider the LTI for the spring-mass system from Example 16.24, and define the LTI

$$\dot{\mathbf{z}}(t) = \mathbf{p}(t) + \mathbf{C}q(t)$$

$$\mathbf{w}(t) = \mathbf{p}(t),$$

taking as input the output produced by the spring-mass system (the position of the mass along the horizontal dimension) and transforms it by a factor  $\mathbf{C}$ . We can compose the two systems, obtaining the system  $\langle \mathbf{st}, \mathbf{A}, \mathbf{B}, \mathbf{C} \rangle$  with

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{k_1+k_2}{m} & 0 & 0 \\ C & 0 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}.$$

This can be visualized more intuitively with the explicit composed dynamics:

$$\begin{bmatrix} \dot{q}(t) \\ \ddot{q}(t) \\ \dot{p}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{k_1+k_2}{m} & 0 & 0 \\ C & 0 & 1 \end{bmatrix} \begin{bmatrix} q(t) \\ \dot{q}(t) \\ p(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} F$$

$$\mathbf{w}(t) = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q(t) \\ \dot{q}(t) \\ p(t) \end{bmatrix}$$

Given an input force, we will get a transformed horizontal position as an output.

## Standard action of LTI systems

**Definition 16.29** (LTI standard action)

We define a standard action of **LTI** via:

▷ A map

$$\begin{aligned} \text{act} : \text{Ob}_{\text{LTI}} &\rightarrow \text{Ob}_{\text{Set}}, \\ n &\mapsto C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^n). \end{aligned} \quad (158)$$

▷ A map

$$\text{act}_\downarrow : \text{Hom}_{\text{LTI}}(m; n) \rightarrow \text{Hom}_{\text{Set}}(C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^m); C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^n)) \quad (159)$$

where  $\text{act}_\downarrow$  takes an LTI system  $f : m \rightarrow n$  given by

$$\langle \text{st}_f, \mathbf{A}_f, \mathbf{B}_f, \mathbf{C}_f, \mathbf{D}_f \rangle, \quad (160)$$

and returns the function

$$\begin{aligned} \text{act}_\downarrow(f) : C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^m) &\rightarrow C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^n), \\ \mathbf{u}(t) &\mapsto \mathbf{C}_f \mathbf{s}_f(t) + \mathbf{D}_f \mathbf{u}(t), \end{aligned} \quad (161)$$

where  $\mathbf{s}_f$  is the unique solution of the initial value problem

$$\begin{cases} \dot{\mathbf{x}}(t) &= \mathbf{A}_f \mathbf{x}(t) + \mathbf{B}_f \mathbf{u}(t) \\ \mathbf{x}(0) &= \text{st}_f. \end{cases} \quad (162)$$

**Remark 16.30.** The initial value problem

$$\begin{cases} \dot{\mathbf{x}}(t) &= \mathbf{A} \mathbf{x}(t) + \mathbf{B} \mathbf{u}(t) \\ \mathbf{x}(0) &= \text{st} \end{cases} \quad (163)$$

is a system of linear first-order non-homogenous differential equations. It does *not* have constant coefficients, because the inhomogenous term  $\mathbf{B}_f \mathbf{u}(t)$  is not constant (it depends on the independent variable  $t$ ).

The theorem of Picard-Lindelöf guarantees that initial value problems of the form (163) always have a unique solution. There is general formula for the solution  $\mathbf{s}(t)$  of (163), namely

$$\mathbf{s}(t) = e^{\mathbf{A}t} \text{st} + \int_0^t e^{\mathbf{A}(t-\tau)} \mathbf{B} \mathbf{u}(s) d\tau. \quad (164)$$

**Lemma 16.31.** Def. 16.29 indeed defines a category action.

*Proof.* We need to prove that

$$\text{act}_\downarrow(f \circ_{\text{LTI}} g) = \text{act}_\downarrow(f) \circ_{\text{Set}} \text{act}_\downarrow(g). \quad (165)$$

Consider  $f : m \rightarrow n$  and  $g : n \rightarrow o$ . We first look at the map

$$\begin{aligned} \text{act}_\downarrow(f \circ_{\text{LTI}} g) : C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^m) &\rightarrow C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^o), \\ \mathbf{u}(t) &\mapsto \mathbf{C}_{f \circ g} \mathbf{s}_{f \circ g}(t) + \mathbf{D}_{f \circ g} \mathbf{u}(t), \end{aligned} \quad (166)$$

where  $\mathbf{s}_{f \circ g}$  is the unique solution of the initial value problem

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}_{f \circ g} \mathbf{x}(t) + \mathbf{B}_{f \circ g} \mathbf{u}(t), \\ \mathbf{x}(0) &= \text{st}_{f \circ g}, \end{aligned} \quad (167)$$

and

$$\mathbf{C}_{f \circ g} = [\mathbf{D}_g \mathbf{C}_f \quad \mathbf{C}_g], \quad \mathbf{D} = \mathbf{D}_g \mathbf{D}_f. \quad (168)$$

From the definition of composition of LTI systems (Def. 16.26), we know

that we can expand (167) into

$$\begin{aligned} \dot{\mathbf{x}}(t) = \begin{bmatrix} \dot{\mathbf{x}}_f(t) \\ \dot{\mathbf{x}}_g(t) \end{bmatrix} &= \begin{bmatrix} \mathbf{A}_f & \mathbf{0} \\ \mathbf{B}_g \mathbf{C}_f & \mathbf{A}_g \end{bmatrix} \begin{bmatrix} \mathbf{x}_f(t) \\ \mathbf{x}_g(t) \end{bmatrix} + \begin{bmatrix} \mathbf{B}_f \\ \mathbf{B}_g \mathbf{D}_f \end{bmatrix} \mathbf{u}(t) \\ \mathbf{st} &= \begin{bmatrix} \mathbf{st}_f \\ \mathbf{st}_g \end{bmatrix}. \end{aligned} \quad (169)$$

By instead looking at  $\text{act}_\rightarrow(f)$  we have

$$\begin{aligned} \text{act}_\rightarrow(f) : C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^m) &\rightarrow C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^n), \\ \mathbf{u} &\mapsto \mathbf{C}_f \mathbf{s}_f + \mathbf{D}_f \mathbf{u}, \end{aligned} \quad (170)$$

where  $\mathbf{s}_f$  is the unique solution of the initial value problem

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}_f \mathbf{x}(t) + \mathbf{B}_f \mathbf{u}(t), \\ \mathbf{x}(0) &= \mathbf{st}_f, \end{aligned} \quad (171)$$

and by looking at  $\text{act}_\rightarrow(g)$  we have

$$\begin{aligned} \text{act}_\rightarrow(g) : C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^n) &\rightarrow C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^o), \\ \mathbf{u} &\mapsto \mathbf{C}_g \mathbf{s}_g + \mathbf{D}_g \mathbf{u}, \end{aligned} \quad (172)$$

where  $\mathbf{s}_g$  is the unique solution of the initial value problem

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}_g \mathbf{x}(t) + \mathbf{B}_g \mathbf{u}(t), \\ \mathbf{x}(0) &= \mathbf{st}_g. \end{aligned} \quad (173)$$

Clearly, considering  $\text{act}_\rightarrow(f) \circ \text{act}_\rightarrow(g)$  is equivalent to considering  $\mathbf{u}_g = \text{act}_\rightarrow(f)(\mathbf{u}_f)$ . By substitution into (172) we obtain

$$\begin{aligned} \text{act}_\rightarrow(f) \circ \text{act}_\rightarrow(g) : C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^m) &\rightarrow C^1(\mathbb{R}_{\geq 0}, \mathbb{R}^o), \\ \mathbf{u} &\mapsto [\mathbf{D}_g \mathbf{C}_f \quad \mathbf{C}_g] \begin{bmatrix} \mathbf{s}_f \\ \mathbf{s}_g \end{bmatrix} + \mathbf{D}_g \mathbf{D}_f \mathbf{u}, \end{aligned} \quad (174)$$

proving the statement.  $\square$

**Lemma 16.32.** Two equivalent systems have the same LTI category action.

*Proof.* Consider two equivalent LTI systems

$$\langle \mathbf{st}_f, \mathbf{A}_f, \mathbf{B}_f, \mathbf{C}_f, \mathbf{D}_f \rangle, \quad \langle \mathbf{st}_g, \mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g \rangle. \quad (175)$$

The initial value problem  $\langle \mathbf{st}_f, \mathbf{A}_f, \mathbf{B}_f, \mathbf{C}_f, \mathbf{D}_f \rangle$  poses reads:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}_f \mathbf{x}(t) + \mathbf{B}_f \mathbf{u}(t), \\ \mathbf{x}(0) &= \mathbf{st}_f. \end{aligned} \quad (176)$$

Let  $\mathbf{s}(t)$  be the solution of (176). Consider the equivalence transformation  $\mathbf{r}(t)$



$= \mathbf{T}\mathbf{s}(t)$ . Then, we have

$$\begin{aligned}
 \dot{\mathbf{r}}(t) &= \mathbf{T}\dot{\mathbf{s}}(t) \\
 &= \mathbf{T}\mathbf{A}_f\mathbf{s}(t) + \mathbf{T}\mathbf{B}_f\mathbf{u}(t) \\
 &= \mathbf{T}\mathbf{A}_f\mathbf{T}^{-1}\mathbf{r}(t) + \mathbf{T}\mathbf{B}_f\mathbf{u}(t) \\
 &= \mathbf{A}_g\mathbf{r}(t) + \mathbf{B}_g\mathbf{u}(t),
 \end{aligned} \tag{177}$$

and  $\mathbf{st}_g = \mathbf{Tst}_f$ . Therefore, the action of the system  $\langle \mathbf{st}_g, \mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g \rangle$  is:

$$\begin{aligned}
 \text{act}_g(\mathbf{g})(\mathbf{u}(t)) &= \mathbf{C}_g\mathbf{r}(t) + \mathbf{D}_g\mathbf{u}(t) \\
 &= \mathbf{C}_f\mathbf{T}^{-1}\mathbf{T}\mathbf{s} + \mathbf{D}_f\mathbf{u}(t) \\
 &= \mathbf{C}_f\mathbf{s}(t) + \mathbf{D}_f\mathbf{u}(t) \\
 &= \text{act}_f(\mathbf{f})(\mathbf{u}(t)).
 \end{aligned} \tag{178}$$

□

**Remark 16.33.** Two LTI systems with the same LTI category action are not necessarily equivalent.

*Proof.* For a simple counterexample, consider the LTI system

$$\begin{aligned}
 \dot{\mathbf{x}}(t) &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{u}(t) \\
 \mathbf{y}(t) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x}(t) + \mathbf{u}(t).
 \end{aligned} \tag{179}$$

The LTI category action of this system will be the same as the one of any system

$$\begin{aligned}
 \dot{\mathbf{x}}(t) &= \begin{bmatrix} 1 & 0 \\ 0 & \alpha \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{u}(t) \\
 \mathbf{y}(t) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x}(t) + \mathbf{u}(t),
 \end{aligned} \tag{180}$$

where  $\alpha \in \mathbb{R}$ , because the output  $\mathbf{y}$  generation does not care about the second component of  $\mathbf{x}$ . However, there is no linear, invertible transformation which relates these systems. Note that control theory has developed tools to deal with hidden parts of the dynamics. Examples are the notion of controllability (ability of an actuator/input to control all states of a system) and observability (ability to estimate all states of the system). □



# 17. Constructing categories

In this chapter we discuss various ways of building new categories from old ones. Given categories  $\mathbf{C}$  and  $\mathbf{D}$ , we will see how to build their cartesian product  $\mathbf{C} \times \mathbf{D}$  and their direct sum  $\mathbf{C} + \mathbf{D}$ , respectively. In addition to these constructions, we will see how to start with a category  $\mathbf{C}$  and construct its *opposite category*  $\mathbf{C}^{\text{op}}$ , as well as various other categories derived from  $\mathbf{C}$  by thinking of kinds of diagrams in  $\mathbf{C}$  as objects themselves.

17.1 Product of Categories . . . . .	286
17.2 Disjoint Union of Categories . .	287
17.3 Opposite Category . . . . .	288
17.4 Arrow construction . . . . .	289
17.5 Twisted arrow construction . . .	290
17.6 (Co)slice construction . . . . .	292

Swiss International Air Lines AG (also known as Swiss), is the flag carrier company of Switzerland. The airline was formed following the bankruptcy of Swissair in 2002.

## 17.1. Product of Categories

### Definition 17.1 (Cartesian product of categories)

Given two categories  $\mathbf{C}$  and  $\mathbf{D}$ , their *cartesian product*  $\mathbf{C} \times \mathbf{D}$  is the category specified as follows:

1. *Objects*: Objects are pairs  $\langle X, Y \rangle$ , with  $X \in \text{Ob}_{\mathbf{C}}$  and  $Y \in \text{Ob}_{\mathbf{D}}$ .
2. *Morphisms*: A morphism from  $\langle X, Y \rangle$  to  $\langle Z, U \rangle$  is a pair of morphisms

$$\langle f, g \rangle : \langle X, Y \rangle \rightarrow \langle Z, U \rangle, \quad (1)$$

with  $f : X \rightarrow_{\mathbf{C}} Z, g : Y \rightarrow_{\mathbf{D}} U$ .

3. *Composition*: The composition of morphisms is given by composing each component of the pair separately:

$$\langle f, g \rangle \circ_{\mathbf{C} \times \mathbf{D}} \langle h, i \rangle = \langle f \circ_{\mathbf{C}} h, g \circ_{\mathbf{D}} i \rangle. \quad (2)$$

4. *Identity morphisms*: Given objects  $X \in \text{Ob}_{\mathbf{C}}$  and  $Y \in \text{Ob}_{\mathbf{D}}$ , the identity morphism on  $\langle X, Y \rangle$  is the pair  $\langle \text{id}_X, \text{id}_Y \rangle$ .

**Remark 17.2.** In a manner analogous to Def. 17.1 we can also define the cartesian product of any finite number of categories. For example, if  $\mathbf{A}, \mathbf{B}, \mathbf{C}$  are three categories, their triple cartesian product  $\mathbf{A} \times \mathbf{B} \times \mathbf{C}$  is a category whose objects are triples  $\langle X, Y, Z \rangle$ , with  $X \in \text{Ob}_{\mathbf{A}}, Y \in \text{Ob}_{\mathbf{B}}, Z \in \text{Ob}_{\mathbf{C}}$ . For any  $n \in \mathbb{N}$ , the  $n$ -fold cartesian product of a category  $\mathbf{C}$  with itself is denoted  $\mathbf{C}^n$ .

## 17.2. Disjoint Union of Categories

### Definition 17.3 (Disjoint union category)

Given two categories  $\mathbf{C}$  and  $\mathbf{D}$ , their *disjoint union*  $\mathbf{C} + \mathbf{D}$  is the category specified as follows:

1. *Objects*:  $\text{Ob}_{\mathbf{C}+\mathbf{D}} := \text{Ob}_{\mathbf{C}} + \text{Ob}_{\mathbf{D}}$ . That is, objects of  $\mathbf{C} + \mathbf{D}$  are tuples of the form  $\langle i, X \rangle$ , with  $i = 1$  and  $X \in \text{Ob}_{\mathbf{C}}$  or  $i = 2$  and  $X \in \text{Ob}_{\mathbf{D}}$ .
2. *Morphisms*: Given objects  $\langle i, X \rangle, \langle j, Y \rangle \in \text{Ob}_{\mathbf{C}+\mathbf{D}}$ ,

$$\text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle i, X \rangle; \langle j, Y \rangle) := \begin{cases} \text{Hom}_{\mathbf{C}}(X; Y) & \text{if } i = j = 1, \\ \text{Hom}_{\mathbf{D}}(X; Y) & \text{if } i = j = 2, \\ \emptyset & \text{else.} \end{cases} \quad (3)$$

3. *Composition*: The composition operations  $\circ_{\mathbf{C}+\mathbf{D}}$ , which are functions from

$$\text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle i, X \rangle; \langle j, Y \rangle) \times \text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle j, Y \rangle; \langle k, Z \rangle)$$

to

$$\text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle i, X \rangle; \langle k, Z \rangle),$$

are equal to

$$\begin{cases} \circ_{\mathbf{C}} & \text{if } i = j = k = 1, \\ \circ_{\mathbf{D}} & \text{if } i = j = k = 2, \end{cases}$$

and equal to the unique function

$$\emptyset \rightarrow \text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle i, X \rangle; \langle k, Z \rangle)$$

in all other cases.

4. *Identity morphisms*: The identities are copied from either category:

$$\text{id}_{\langle 1, X \rangle}^{\mathbf{C}+\mathbf{D}} := \text{id}_X^{\mathbf{C}}, \quad (4)$$

$$\text{id}_{\langle 2, X \rangle}^{\mathbf{C}+\mathbf{D}} := \text{id}_X^{\mathbf{D}}. \quad (5)$$

**Remark 17.4.** If you think about categories in diagrammatic form, this operation corresponds to placing two categories side-by-side, without connecting them.

**Remark 17.5.** A remark similar to Remark 17.2 holds here, too: we can form the disjoint union of any finite number of categories. For example,  $\mathbf{A} + \mathbf{B} + \mathbf{C}$  for three categories  $\mathbf{A}, \mathbf{B}, \mathbf{C}$ .

**Exercise 33.** Show that the disjoint union of two categories is indeed again a category.

*See solution on page 301.*

### 17.3. Opposite Category

**Definition 17.6** (Opposite category)

Given a category  $\mathbf{C}$ , its *opposite category*  $\mathbf{C}^{\text{op}}$  is specified by:

1. **Objects:**  $\text{Ob}_{\mathbf{C}^{\text{op}}} = \text{Ob}_{\mathbf{C}}$ .
2. **Morphisms:** Given objects  $X, Y \in \text{Ob}_{\mathbf{C}^{\text{op}}} = \text{Ob}_{\mathbf{C}}$ ,

$$\text{Hom}_{\mathbf{C}^{\text{op}}}(X; Y) := \text{Hom}_{\mathbf{C}}(Y; X). \quad (6)$$

For each morphism  $f : X \rightarrow_{\mathbf{C}} Y$ , there is a morphism  $f^{\text{op}} : Y \rightarrow_{\mathbf{C}^{\text{op}}} X$ . Graphically, given

$$X \xrightarrow{f} Y \quad (7)$$

we have

$$Y \xrightarrow{f^{\text{op}}} X \quad (8)$$

3. **Composition:** Given morphisms

$$f^{\text{op}} : Z \rightarrow_{\mathbf{C}^{\text{op}}} Y \quad \text{and} \quad g^{\text{op}} : Y \rightarrow_{\mathbf{C}^{\text{op}}} X \quad (9)$$

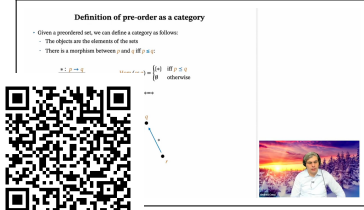
their composition is defined as

$$f^{\text{op}} \circ_{\mathbf{C}^{\text{op}}} g^{\text{op}} := (g \circ_{\mathbf{C}} f)^{\text{op}}. \quad (10)$$

4. **Identity morphisms:** given by the identities of the original category  $\mathbf{C}$ .

Given  $X \in \text{Ob}_{\mathbf{C}}$ , we will sometimes (though not always) write  $X^{\text{op}}$  to signify when we are thinking of  $X$  as an object of  $\text{Ob}_{\mathbf{C}^{\text{op}}}$ .

Watch: *Pre-order as a category* (2 minutes).


**Graded exercise E.12** (OppositeCat)

Verify that Def. 17.6 defines a category. In other words, check that its constituents satisfy the conditions of associativity and unitality.

**Example 17.7** (Opposite of a poset). A single poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  can be described as a category, in which each point is an object, and there is a morphism between two objects  $x$  and  $y$  if and only if  $x \leq_{\mathbf{P}} y$ . We have defined the opposite of a poset in Section 10.5. The opposite category of a category for a poset, is the category for the opposite poset.

$$(\mathbf{C}(\mathbf{P}))^{\text{op}} \simeq \mathbf{C}(\mathbf{P}^{\text{op}}). \quad (11)$$

## 17.4. Arrow construction

### Definition 17.8 (Arrow category)

Given any category  $\mathbf{C}$ , its *arrow category*  $\mathbf{Arr} \mathbf{C}$  is:

1. *Objects*: morphisms of  $\mathbf{C}$ .
2. *Morphisms*: Given objects  $f : X \rightarrow_{\mathbf{C}} Y$  and  $g : Z \rightarrow_{\mathbf{C}} U$ , a morphism  $\varphi : f \rightarrow_{\mathbf{Arr} \mathbf{C}} g$  in  $\mathbf{Arr} \mathbf{C}$  is a pair of morphisms  $\langle \varphi_s, \varphi_t \rangle$  in  $\mathbf{C}$  that make the following diagram

$$\begin{array}{ccc} X & \xrightarrow{\varphi_s} & Z \\ f \downarrow & & \downarrow g \\ Y & \xrightarrow{\varphi_t} & U \end{array} \quad (12)$$

a commutative square in  $\mathbf{C}$ .

3. *Composition*: Composition in  $\mathbf{Arr} \mathbf{C}$  is given by placing commutative squares side by side. Consider  $\langle \varphi_s, \varphi_t \rangle : f \rightarrow g$  and  $\langle \psi_s, \psi_t \rangle : g \rightarrow h$  in  $\mathbf{Arr} \mathbf{C}$  giving rise to the following composite commutative diagram

$$\begin{array}{ccccc} X & \xrightarrow{\varphi_s} & Z & \xrightarrow{\psi_s} & V \\ f \downarrow & & \downarrow g & & \downarrow h \\ Y & \xrightarrow{\varphi_t} & U & \xrightarrow{\psi_t} & W \end{array} \quad (13)$$

Since this diagram is again commutative, we define

$$\langle \varphi_s, \varphi_t \rangle \circ_{\mathbf{Arr} \mathbf{C}} \langle \psi_s, \psi_t \rangle := \langle \varphi_s \circ_{\mathbf{C}} \psi_s, \varphi_t \circ_{\mathbf{C}} \psi_t \rangle. \quad (14)$$

4. *Identities*: given an object  $f : X \rightarrow Y$  of  $\mathbf{Arr} \mathbf{C}$ , its identity morphism is  $\langle \text{id}_X, \text{id}_Y \rangle$ .

**Example 17.9** (Intervals). Consider a poset  $\mathbf{P}$ . The arrow category  $\mathbf{Arr}(\mathbf{C}(\mathbf{P}))$  is isomorphic to the poset (viewed as a category)  $\mathbf{C}((\mathbf{Arr} \mathbf{P}))$  of nonempty *intervals* in  $\mathbf{P}$ :

$$\mathbf{Arr}(\mathbf{C}(\mathbf{P})) \simeq \mathbf{C}((\mathbf{Arr} \mathbf{P})). \quad (15)$$

## 17.5. Twisted arrow construction

### Definition 17.10 (Twisted arrow category)

Given a category  $\mathbf{C}$ , its *twisted arrow category*  $\mathbf{Tw} \mathbf{C}$  is:

1. *Objects*: morphisms in  $\mathbf{C}$ .
2. *Morphisms*: A morphism in  $\mathbf{Tw} \mathbf{C}$  from  $f : X \rightarrow_{\mathbf{C}} Y$  to  $g : Z \rightarrow_{\mathbf{C}} U$  is given by a pair of morphisms  $\langle \varphi_s, \varphi_t \rangle$  in  $\mathbf{C}$  such that the following diagram commutes:

$$\begin{array}{ccc} X & \xleftarrow{\varphi_s} & Z \\ f \downarrow & & \downarrow g \\ Y & \xrightarrow{\varphi_t} & U \end{array} \quad (16)$$

3. *Composition*: Composition in  $\mathbf{Tw} \mathbf{C}$  is given by placing commutative squares side by side. Consider  $\langle \varphi_s, \varphi_t \rangle : f \rightarrow g$  and  $\langle \psi_s, \psi_t \rangle : g \rightarrow h$  in  $\mathbf{Arr} \mathbf{C}$ , giving rise to the following composite diagram

$$\begin{array}{ccccc} X & \xleftarrow{\varphi_s} & Z & \xleftarrow{\psi_s} & V \\ f \downarrow & & \downarrow g & & \downarrow h \\ Y & \xrightarrow{\varphi_t} & U & \xrightarrow{\psi_t} & W \end{array} \quad (17)$$

It is commutative because its component squares are, and hence it defines a morphism  $f \rightarrow h$  in  $\mathbf{Tw} \mathbf{C}$ , namely

$$\langle \varphi_s, \varphi_t \rangle \circ_{\mathbf{Arr} \mathbf{C}} \langle \psi_s, \psi_t \rangle := \langle \psi_s \circ_{\mathbf{C}} \varphi_s, \varphi_t \circ_{\mathbf{C}} \psi_t \rangle. \quad (18)$$

4. *Identities*: given an object  $f : X \rightarrow Y$  of  $\mathbf{Tw} \mathbf{C}$ , its identity morphism is  $\langle \text{id}_X, \text{id}_Y \rangle$ .

**Remark 17.11.** The above construction might be more precisely called the “source twisted arrow category” of  $\mathbf{C}$ , because it is a modification of the arrow construction where we are “twisting” the arrow between source objects of morphisms of  $\mathbf{C}$  by reversing its direction. An analogous construction exists where instead we twist the arrow construction by reversing the arrow between targets of morphisms of  $\mathbf{C}$ . This latter construction might be called the “target twisted arrow category”. For brevity, we have only spelled out the source twisted variant here.

### Graded exercise E.13 (TwistedCat)

Let  $\mathbf{C}$  be a category, and let  $\mathbf{Tw} \mathbf{C}$  be the associated twisted arrow category. Check that the definition of  $\mathbf{Tw} \mathbf{C}$  does indeed define a category. Specifically, check that for  $\mathbf{Tw} \mathbf{C}$

1. composition of composable morphism does again define a morphism of  $\mathbf{Tw} \mathbf{C}$ ;
2. composition is associative;
3. identity morphisms satisfy the identity laws (that they behave neutrally for composition).

**Example 17.12** (Twisted construction in posets and categories). Consider a poset  $\mathbf{P}$ . The twisted arrow category  $\mathbf{Tw} \mathbf{C}(\mathbf{P})$  is isomorphic to the poset (viewed as a category)  $\mathbf{C}((\mathbf{Tw} \mathbf{P}))$  of nonempty *intervals* in  $\mathbf{P}$ :

$$\mathbf{Tw}(\mathbf{C}(\mathbf{P})) \simeq \mathbf{C}((\mathbf{Tw} \mathbf{P})). \quad (19)$$

**Exercise 34.** Prove the statement in Example 17.12. Recall that, given elements  $x, y \in \mathbf{P}$ , the interval  $[x, y]$  is

$$[x, y] := \{z \in \mathbf{P} \mid x \leq_{\mathbf{P}} z \leq_{\mathbf{P}} y\}. \quad (20)$$

Start to show that the partial order is equivalent to a twisted morphism.

*See solution on page 301.*



## 17.6. (Co)slice construction

### Definition 17.13 (Slice categories)

Let  $\mathbf{C}$  be a category and fix an object  $T \in \text{Ob}_{\mathbf{C}}$ . The *slice category*  $\mathbf{C}/T$  of  $\mathbf{C}$  over  $T$  is:

1. *Objects*: morphisms in  $\mathbf{C}$  of the type  $X \rightarrow T$ , where  $X$  ranges over all objects of  $\mathbf{C}$ .
2. *Morphisms*: given objects  $X \xrightarrow{f} T$  and  $Y \xrightarrow{g} T$ , a morphism

$$\varphi_{\mathbf{C}/T} : (X \xrightarrow{f} T) \rightarrow_{\mathbf{C}/T} (Y \xrightarrow{g} T) \quad (21)$$

is specified by a morphism  $\varphi : X \rightarrow_{\mathbf{C}} Y$  such that the diagram

$$\begin{array}{ccc} X & \xrightarrow{\varphi} & Y \\ & \searrow f & \swarrow g \\ & T & \end{array} \quad (22)$$

commutes.

3. *Composition*: defined via the composition in  $\mathbf{C}$ . Concretely, given composable morphisms  $\varphi_{\mathbf{C}/T}$  and  $\psi_{\mathbf{C}/T}$  of  $\mathbf{C}/T$ , we define

$$\varphi_{\mathbf{C}/T} \circ_{\mathbf{C}/T} \psi_{\mathbf{C}/T} := (\varphi \circ_{\mathbf{C}} \psi)_{\mathbf{C}/T}. \quad (23)$$

4. *Identities*: defined by the identities in  $\mathbf{C}$ .

### Graded exercise E.14 (SliceCat)

Let  $\mathbf{C}$  be a category, fix  $T \in \text{Ob}_{\mathbf{C}}$ , and consider the slice category  $\mathbf{C}/T$ . Your task is to check that the composition of two composable morphisms in  $\mathbf{C}/T$  is again in fact a morphism in  $\mathbf{C}/T$ .

### Definition 17.14 (Coslice categories)

Let  $\mathbf{C}$  be a category and fix an object  $S \in \text{Ob}_{\mathbf{C}}$ . The *coslice category*  $S/\mathbf{C}$  of  $\mathbf{C}$  under  $S$  is:

1. *Objects*: morphisms in  $\mathbf{C}$  of the type  $S \rightarrow X$ , where  $X$  ranges over all objects of  $\mathbf{C}$ .
2. *Morphisms*: given objects  $S \xrightarrow{f} X$  and  $S \xrightarrow{g} Y$ , a morphism

$$\varphi_{S/\mathbf{C}} : (S \xrightarrow{f} X) \rightarrow_{S/\mathbf{C}} (S \xrightarrow{g} Y) \quad (24)$$

is specified by a morphism  $\varphi : X \rightarrow_{\mathbf{C}} Y$  such that the diagram

$$\begin{array}{ccc} X & \xrightarrow{\varphi} & Y \\ & \swarrow f & \searrow g \\ & S & \end{array} \quad (25)$$

commutes.

3. *Composition*: defined by the composition in  $\mathbf{C}$ .
4. *Identities*: defined by the identities in  $\mathbf{C}$ .



## 18. Culture

Culture is what you don't think about. Mathematics has a quite different culture than engineering. Engineering is about getting things to work *in practice*. Conventions are important. Specifics of protocols are important. In mathematics, certain things just not matter: one studies what is true regardless of conventions.

<b>18.1 Definition vs computation . . . .</b>	<b>294</b>
<b>18.2 Things that don't matter . . . . .</b>	<b>295</b>
<b>18.3 Choice of symbols . . . . .</b>	<b>296</b>
<b>18.4 Typographical conventions . . . .</b>	<b>297</b>

*Schwingen* is the traditional Swiss wrestling native to the pre-alpine parts of German-speaking Switzerland. Wrestlers wear *Schwingerhosen* that can be used to hold and grapple the adversary.

## 18.1. Definitional impetus vs computational goals

The category **Curr** represents the set of all possible currency exchangers that could ever exist. However, in this set there would be very irrational agents. For example, there is a currency exchanger that, given 1 USD, will give you back 2 USD; there is one currency exchanger that corresponds to converting USD to CHF back and forth 21 times before getting you the money. There is even one that will not give you back any money.

Moreover, using the composition operations we could produce many more morphisms. In fact, if there are loops, we could traverse the loops multiple times, and, depending on the numbers, finding new morphisms, possibly infinitely many more.

This highlights a recurring topic: often mathematicians will be happy to define a broader category of objects, while, in practice, the engineer will find herself thinking about a more constrained set of objects. In particular, while the mathematician is more concerned with defining categories as hypothetical universes of things, the engineer is typically interested in representing concrete things, and solve some computational problem on the represented structure.

For example, in the case of the currency exchangers, the problem might be that of finding the sequence of the best conversions between a source and a target currency.

First, the engineer would add more constraints to the definition to work with more well-behaved objects. For example, it is reasonable to limit the universe of morphisms in such a way that the action of converting back and forth the same currency to have a cost (through the commission) higher than 0.

In that case, we will find that the optimal paths of currencies never pass through a currency more than once. To see this, consider three currencies **A**, **B**, **C**, a currency exchanger  $\langle a, b \rangle$  from **A** to **B**, a currency exchanger  $\langle c, d \rangle$  from **B** to **C**, and a currency exchanger  $\langle e, f \rangle$  from **C** to **A**. The composition of the currency exchangers reads:

$$\langle eca, ecb + ed + f \rangle = \langle g, h \rangle. \quad (1)$$

Assuming  $e = a^{-1}$  (in words, an exchange rate direction is not more profitable than the other), and  $h \neq 0$ , because of the commissions we can show that there are multiple morphisms from **A** to **A**, and that the identity morphism is the most “convenient” one. If we only pass through each currency at most once, there are only a finite amount of paths to check, and this might simplify the computational problem.

Second, the engineer might be interested in keeping track only of the “dominant” currency exchangers. For example, if we have two exchangers with the same rate but different commission, we might want to keep track only of the one with the lowest commission.

In the next chapters we will see that there are concepts that will be useful to model these situations:

- ▷ There is a concept of *subcategory* that allows to define more specific categories of a parent one, in a way that still satisfies the axioms.
- ▷ There is a concept called *locally posetal* categories, in which the set of morphisms between two objects is assumed to be a *poset* rather than a *set*, that is, we assume that there is an order, and that this order will be compatible with the operation of composition.

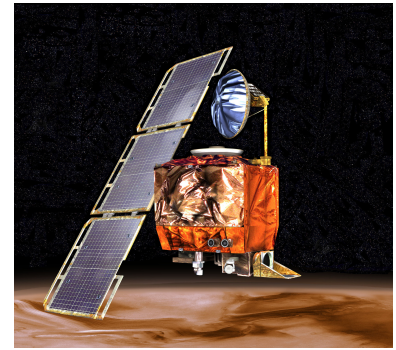
## 18.2. Things that don't matter

In engineering we know that **using the right conventions is essential**.

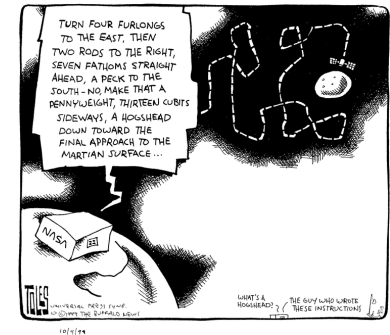
There are many famous examples of unit mismatches causing disasters or near-disasters:

- ▷ The loss of the Mars Climate Orbiter in 1999 was due to the fact that NASA used the metric system, while contractor Lockheed Martin used (by mistake) imperial units (Fig. 1).
- ▷ In 1983, an Air Canada's Boeing 767 jet ran out of fuel in mid-flight because there was a miscalculation of the fuel needed for the trip. In the end, the pilot managed to successfully land the “Gimli Glider”.
- ▷ Going back in history, Columbus wound up in the Bahamas because he miscalculated the Earth's circumference, due to several mistakes, and one of them was assuming that his sources were using the *Roman mile* rather than the *Arabic mile* [22]. Columbus' mathematical mistakes led to a happy incident for him, but not so great outcomes for many others.

However, in category theory, we look at the “essence” of things, and we consider **what is true regardless of conventions**.



(a) Mars climate orbiter



(b)

Figure 1.





**Figure 2.:** A page from Muḥammad ibn Mūsā al-Khwārizmī's *Algebra*. The word *algorithm* comes from the name *al-Khwārizmī*.

### 18.3. The choice of symbols does not matter

For example, it doesn't matter what symbols we use to represent numerals. It also

٠	١	٢	٣	٤	٥	٦	٧	٨	٩
0	1	2	3	4	5	6	7	8	9
sifr	waahid	eeth-nayn	thalaatha	arba'a	khamsa	sitta	sab'a	thamaaneeya	tis'a

should not matter that we use base-10 numerals—certainly mathematical truths do not depend on how many fingers humans have.

Just like this book is written in rather plain English, and could be translated to another language while preserving the meaning, in category theory we look at what is not changed by a 1:1 translation that can be reversed.

This will be covered later in a section on “isomorphisms”; but for now we can look at this intuitively.

## 18.4. Typographical conventions don't matter

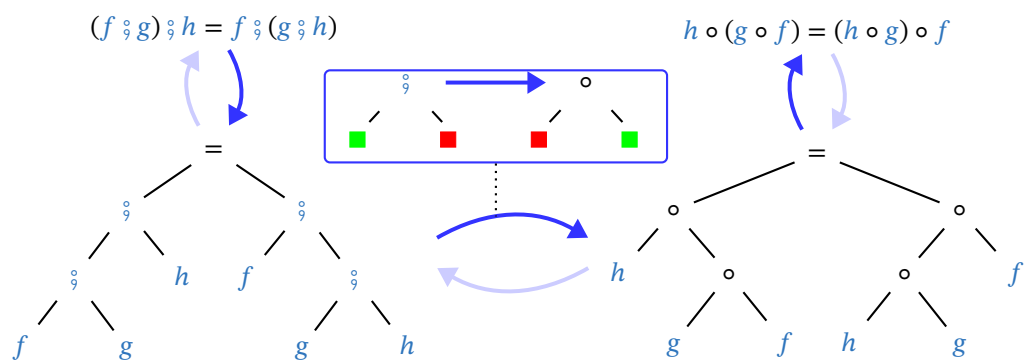
Some of you might have objected to the conventions that we used in this chapter for the notation for composition of morphisms. We have used the notation  $f \circ g$  (“ $f$  then  $g$ ”) while usually in the rest of mathematics we would have used  $g \circ f$  (“ $g$  after  $f$ ”). However, any concept we will use is “invariant” to the choice of notation. We can decide to rewrite the book using the other convention and still all the theorems would remain true, and all the falsities will remain false. More technically, we can take any formula written in one convention and rewrite it with the other convention, and vice versa using a specific mechanical rule. For example, the formula

$$(f \circ g) \circ h = f \circ (g \circ h) \quad (2)$$

would be transformed in

$$h \circ (g \circ f) = (h \circ g) \circ f. \quad (3)$$

(A bit more advanced category theory can describe this transformation more precisely.)



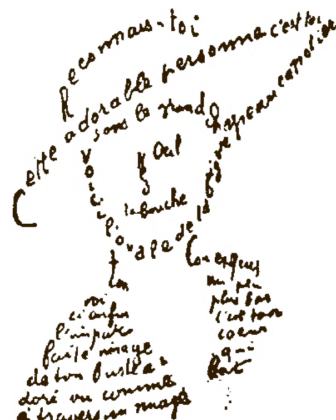
**Figure 3.:** Mechanical rule to transform one convention to another.


This invariance to mechanical invertible transformations only holds for mathematical and technical writing. In other contexts it might fail.  
For example, “Veni Vidi Vici” has a sound to it that the English translation does not have (Fig. 4).

VENI VIDI VICI            I came; I saw; I conquered.

Figure 4.

Sometimes, the meaning is in the typography, as in Apollinaire’s poem “Poème à Lou” (Fig. 5).





Reconnais-toi  
Cette adorable personne c’est toi  
Sous le grand chapeau canotier  
Oeil  
Nez  
La bouche  
Voici l’ovale de ta figure  
Ton cou exquis  
Voici enfin l’imparfaite image de ton buste adoré  
vu comme à travers un nuage  
Un peu plus bas c’est ton coeur qui bat

Apollinaire - 1915 - Poème à Lou

Figure 5.

## Solutions to selected exercises

**Solution of Exercise 25.** We know:

- ▷  $\text{src}(1) = a$  and  $\text{tgt}(1) = b$ : we have  $F_*(a) = \alpha$  and  $F_*(b) = \beta$ , meaning that  $F_*(1) = \text{I}$ .
- ▷  $\text{src}(2) = b$  and  $\text{tgt}(2) = c$ : we have  $F_*(b) = \beta$  and  $F_*(c) = \gamma$ , meaning that  $F_*(2) = \text{II}$ .
- ▷  $\text{src}(3) = c$  and  $\text{tgt}(3) = d$ : we have  $F_*(c) = \gamma$  and  $F_*(d) = \alpha$ , meaning that  $F_*(3) = \text{III}$ .
- ▷  $\text{src}(4) = d$  and  $\text{tgt}(4) = e$ : we have  $F_*(d) = \alpha$  and  $F_*(e) = \gamma$ , meaning that  $F_*(4) = \text{IV}$ .
- ▷  $\text{src}(5) = e$  and  $\text{tgt}(5) = a$ : we have  $F_*(e) = \gamma$  and  $F_*(a) = \alpha$ , meaning that  $F_*(5) = \text{III}$ .

Therefore, the map  $F_*$  is as reported in Fig. 6.

**Solution of Exercise 26.** We define the category **InjSet** to be such that its objects are all sets, its morphisms are injective functions, composition is the usual composition of functions, and identity morphisms are the usual identity functions.

1. We show that the composition of two injective functions is injective. Given functions  $f : X \rightarrow Y$  and  $g : Y \rightarrow Z$  injective, suppose that  $g(f(x_1)) = g(f(x_2))$  for some  $x_1, x_2 \in X$ . By the injectivity of  $g$  it follows that  $f(x_1) = f(x_2)$ , and then using the injectivity of  $f$ , we can conclude that  $x_1 = x_2$ .
2. Identity functions are clearly injective.
3. Associativity of composition holds because it holds for all functions, so in particular also for injective ones.

**Solution of Exercise 27.**

**Solution of Exercise 28.**

**Solution of Exercise 29.**

**Solution of Exercise 30.** The concept of this exercise is very similar to the one of **Curr**. In general, we can write a temperature converter (morphism) from **e** to **f** as a pair of numbers  $\langle a, d \rangle$ ,  $a, d \in \mathbb{R}$ . For each morphism we have a map which actually transform an amount of the first temperature into an amount of the second temperature:

$$\begin{aligned} f_{\langle a, d \rangle} : \mathbb{R} &\rightarrow \mathbb{R} \\ x &\mapsto ax + d. \end{aligned} \quad (4)$$

Now, all the possible conversions between the three temperature conventions feature specific values for  $a, d$ , listed in Table 18.1 (rows are to be intended as the source, and columns as the target convention).

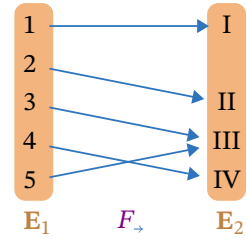
**Table 18.1.:** Temperature conversion factors.

	Celsius	Kelvin	Fahrenheit
Celsius	$a = 1, d = 0$	$a = 1, d = 273$	$a = 9/5, d = 32$
Kelvin	$a = 1, d = -273$	$a = 1, d = 0$	$a = 9/5, d = -459.4$
Fahrenheit	$a = 5/9, d = -17.7$	$a = 5/9, d = 255.2$	$a = 1, d = 0$

We now define the category **Temp** as being constituted of:

- ▷ **Objects:**  $\text{Ob}_{\text{Temp}} = \{\text{Celsius}, \text{Kelvin}, \text{Fahrenheit}\}$ ;
- ▷ **Morphisms:** There is a single morphism from **e** to **f** for any  $e, f \in \text{Ob}_{\text{Temp}}$ , given by  $\langle a, d \rangle$ , with  $a, d$  as in Table 18.1;

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.



**Figure 6.**



▷ *Composition of morphisms*: Analogous to the one in **Curr**;

▷ *Identity morphisms*: Analogous to the ones in **Curr**.

This indeed forms a category, as we have shown for **Curr**.

**Solution of Exercise 31.** The only nontrivial step is checking that associativity holds for composition of  $\text{dur}$ . Consider 3 compatible morphisms  $f_1, f_2, f_3$ . We know from (34) that

$$\text{dur}_{1;2}(n) = \text{dur}_1(n) + \text{dur}_2(\sigma_1(n)). \quad (5)$$

Now we compose  $f_{1;2}$  with  $f_3$ . For  $\text{dur}$  we obtain

$$\text{dur}_{(1;2);3}(n) = \text{dur}_{1;2}(n) + \text{dur}_3(\sigma_{1;2}(n)) \quad (6)$$

$$= \text{dur}_1(n) + \text{dur}_2(\sigma_1(n)) + \text{dur}_3(\sigma_2(\sigma_1(n))) \quad (7)$$

Instead, if we first compute  $f_2 \circ f_3$ , we have

$$\text{dur}_{2;3}(n) = \text{dur}_2(n) + \text{dur}_3(\sigma_2(n)). \quad (8)$$

If we now compose  $f_1$  with  $f_2 \circ f_3$  we get:

$$\text{dur}_{1;(2;3)}(n) = \text{dur}_1(n) + \text{dur}_{2;3}(\sigma_1(n)) \quad (9)$$

$$= \text{dur}_1(n) + \text{dur}_2(n) + \text{dur}_3(\sigma_2(\sigma_1(n))), \quad (10)$$

which is the same as (7).

**Solution of Exercise 32.** We start with associativity. Consider morphism  $f : l \rightarrow m$  given by  $\langle \text{st}_f, \mathbf{A}_f, \mathbf{B}_f, \mathbf{C}_f, \mathbf{D}_f \rangle$ , and  $g : m \rightarrow n$  given by  $\langle \text{st}_g, \mathbf{A}_g, \mathbf{B}_g, \mathbf{C}_g, \mathbf{D}_g \rangle$ , and  $h : n \rightarrow o$  given by  $\langle \text{st}_h, \mathbf{A}_h, \mathbf{B}_h, \mathbf{C}_h, \mathbf{D}_h \rangle$ . The morphism  $f \circ g$  is described by the LTI

$$\langle \text{st}_{f,g}, \mathbf{A}_{f,g}, \mathbf{B}_{f,g}, \mathbf{C}_{f,g}, \mathbf{D}_{f,g} \rangle, \quad (11)$$

where

$$\text{st}_{f,g} = \begin{bmatrix} \text{st}_f \\ \text{st}_g \end{bmatrix}, \quad \mathbf{A}_{f,g} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0} \\ \mathbf{B}_g \mathbf{C}_f & \mathbf{A}_g \end{bmatrix}, \quad \mathbf{B}_{1,2} = \begin{bmatrix} \mathbf{B}_f \\ \mathbf{B}_g \mathbf{D}_f \end{bmatrix}, \quad \mathbf{C}_{f,g} = [\mathbf{D}_g \mathbf{C}_f \quad \mathbf{C}_g], \quad \mathbf{D}_{f,g} = \mathbf{D}_g \mathbf{D}_f. \quad (12)$$

The morphism  $(f \circ g) \circ h$  is described by

$$\langle \text{st}_{(f,g),h}, \mathbf{A}_{(f,g),h}, \mathbf{B}_{(f,g),h}, \mathbf{C}_{(f,g),h}, \mathbf{D}_{(f,g),h} \rangle, \quad (13)$$

where:

$$\text{st}_{(f,g),h} = \begin{bmatrix} \text{st}_f \\ \text{st}_g \\ \text{st}_h \end{bmatrix}, \quad \mathbf{A}_{(f,g),h} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0} & \mathbf{0} \\ \mathbf{B}_g \mathbf{C}_f & \mathbf{A}_g & \mathbf{0} \\ \mathbf{B}_h \mathbf{D}_g \mathbf{C}_f & \mathbf{B}_h \mathbf{C}_g & \mathbf{A}_h \end{bmatrix}, \quad \mathbf{B}_{(f,g),h} = \begin{bmatrix} \mathbf{B}_f \\ \mathbf{B}_g \mathbf{D}_f \\ \mathbf{B}_h \mathbf{D}_g \mathbf{D}_f \end{bmatrix}, \quad (14)$$

$$\mathbf{C}_{(f,g),h} = [\mathbf{D}_h \mathbf{D}_g \mathbf{C}_f \quad \mathbf{D}_h \mathbf{C}_g \quad \mathbf{C}_h], \quad \mathbf{D}_{(f,g),h} = \mathbf{D}_h \mathbf{D}_g \mathbf{D}_f.$$

On the other hand, the morphism  $g \circ h$  is described by

$$\langle \text{st}_{g,h}, \mathbf{A}_{g,h}, \mathbf{B}_{g,h}, \mathbf{C}_{g,h}, \mathbf{D}_{g,h} \rangle, \quad (15)$$

where:

$$\text{st}_{g,h} = \begin{bmatrix} \text{st}_g \\ \text{st}_h \end{bmatrix}, \quad \mathbf{A}_{g,h} = \begin{bmatrix} \mathbf{A}_g & \mathbf{0} \\ \mathbf{B}_h \mathbf{C}_g & \mathbf{A}_h \end{bmatrix}, \quad \mathbf{B}_{g,h} = \begin{bmatrix} \mathbf{B}_g \\ \mathbf{B}_h \mathbf{D}_g \end{bmatrix}, \quad \mathbf{C}_{g,h} = [\mathbf{D}_h \mathbf{C}_g \quad \mathbf{C}_h], \quad \mathbf{D}_{g,h} = \mathbf{D}_h \mathbf{D}_g. \quad (16)$$

Furthermore, the morphism  $f \circ (g \circ h)$  is described by

$$\langle \text{st}_{f,(g,h)}, \mathbf{A}_{f,(g,h)}, \mathbf{B}_{f,(g,h)}, \mathbf{C}_{f,(g,h)}, \mathbf{D}_{f,(g,h)} \rangle, \quad (17)$$

where:

$$\begin{aligned} \text{st}_{f,(g,h)} &= \begin{bmatrix} \text{st}_f \\ \text{st}_g \\ \text{st}_h \end{bmatrix}, \quad \mathbf{A}_{f,(g,h)} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0} & \mathbf{0} \\ \mathbf{B}_g \mathbf{C}_f & \mathbf{A}_g & \mathbf{0} \\ \mathbf{B}_h \mathbf{D}_g \mathbf{C}_f & \mathbf{B}_h \mathbf{C}_g & \mathbf{A}_h \end{bmatrix}, \quad \mathbf{B}_{f,(g,h)} = \begin{bmatrix} \mathbf{B}_f \\ \mathbf{B}_g \mathbf{D}_f \\ \mathbf{B}_h \mathbf{D}_g \mathbf{D}_f \end{bmatrix}, \\ \mathbf{C}_{f,(g,h)} &= [\mathbf{D}_h \mathbf{D}_g \mathbf{C}_f \quad \mathbf{D}_h \mathbf{C}_g \quad \mathbf{C}_h], \quad \mathbf{D}_{f,(g,h)} = \mathbf{D}_h \mathbf{D}_g \mathbf{D}_f. \end{aligned} \quad (18)$$

Clearly, the matrices in (14) and (18) coincide, showing associativity.

We now show unitality. Consider a morphism  $f : l \rightarrow m$ , described by  $\langle \text{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ . The morphism  $\text{id}_l \circ f$  is a morphism  $l \rightarrow m$  still given by  $\langle \text{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ . Similarly, the morphism  $f \circ \text{id}_m$  is a morphism  $l \rightarrow m$ , given by  $\langle \text{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle$ . Therefore, **LTI** is a category.

**Solution of Exercise 33.** We check the two conditions. First, consider a morphism  $f : X \rightarrow Y \in \text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle X, i \rangle; \langle Y, i \rangle)$  (the index  $i$  is repeated, because following the definition of morphisms, no morphism connects objects of one category to objects of the other one). We have

$$\text{id}_{\mathbf{C}+\mathbf{D}} \circ f = \begin{cases} \text{id}_{\mathbf{C}} \circ f = f & \text{if } i = 1, \\ \text{id}_{\mathbf{D}} \circ f = f & \text{if } i = 2, \end{cases} \quad (19)$$

and

$$f \circ \text{id}_{\mathbf{C}+\mathbf{D}} = \begin{cases} f \circ \text{id}_{\mathbf{C}} = f & \text{if } i = 1, \\ f \circ \text{id}_{\mathbf{D}} = f & \text{if } i = 2. \end{cases} \quad (20)$$

Second, consider the morphisms  $f : X \rightarrow Y \in \text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle X, i \rangle; \langle Y, i \rangle)$ ,  $g : Y \rightarrow Z \in \text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle Y, j \rangle; \langle Z, j \rangle)$ , and  $h : Z \rightarrow U \in \text{Hom}_{\mathbf{C}+\mathbf{D}}(\langle Z, k \rangle; \langle U, k \rangle)$ . We have

$$(f \circ_{\mathbf{C}+\mathbf{D}} g) \circ_{\mathbf{C}+\mathbf{D}} h := \begin{cases} (f \circ_{\mathbf{C}} g) \circ_{\mathbf{C}} h = f \circ_{\mathbf{C}} g \circ_{\mathbf{C}} h & \text{if } i = j = k = 1, \\ (f \circ_{\mathbf{D}} g) \circ_{\mathbf{D}} h = f \circ_{\mathbf{D}} g \circ_{\mathbf{D}} h & \text{if } i = j = k = 2, \\ \text{does not exist} & \text{else.} \end{cases} \quad (21)$$

and

$$f \circ_{\mathbf{C}+\mathbf{D}} (g \circ_{\mathbf{C}+\mathbf{D}} h) := \begin{cases} f \circ_{\mathbf{C}} (g \circ_{\mathbf{C}} h) = f \circ_{\mathbf{C}} g \circ_{\mathbf{C}} h & \text{if } i = j = k = 1, \\ f \circ_{\mathbf{D}} (g \circ_{\mathbf{D}} h) = f \circ_{\mathbf{D}} g \circ_{\mathbf{D}} h & \text{if } i = j = k = 2, \\ \text{does not exist} & \text{else.} \end{cases} \quad (22)$$

**Solution of Exercise 34.** Consider  $\mathbf{C}((\text{Tw } \mathbf{P}))$ . Take two morphisms in  $\mathbf{P}$ :  $f : X \rightarrow Y$  (from  $X \leq Y$ ) and  $g : Z \rightarrow U$  (from  $Z \leq U$ ). These are two objects in  $\mathbf{C}((\text{Tw } \mathbf{P}))$ . Now, a morphism in  $\mathbf{C}((\text{Tw } \mathbf{P}))$  is a pair  $\langle h, i \rangle$  where  $h : Z \rightarrow X$  (from  $Z \leq X$ ) and  $i : Y \rightarrow U$  (from  $Y \leq U$ ). Therefore, we have  $Z \leq X \leq Y \leq U$ , which corresponds to  $[X, Y] \leq_{\text{Tw } \mathbf{P}} [Z, U]$ . Therefore, morphisms in  $\mathbf{C}((\text{Tw } \mathbf{P}))$  between arrows (intervals) correspond to order relations between intervals.



# PART F.FUNCTORS



---

<b>19. Translation</b>	<b>305</b>
<b>20. Specialization</b>	<b>317</b>
<b>21. Syntax and semantics</b>	<b>323</b>
<b>22. Up the ladder of abstraction</b>	<b>327</b>

---

The “Stoosbahn”, also known as “Schwyz-Stoos” funicular, is a Swiss funicular railway. On a length of 1.7 kilometers, it climbs a height difference of 744 metres: it is the steepest funicular railway in Europe, second in the world (surpassed by the “Katoomba Scenic Railway” in Australia).





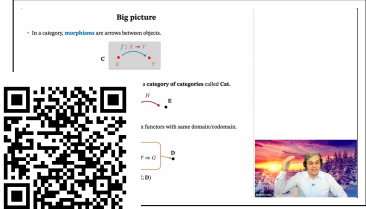


## 19. Translation

In this chapter we discuss *functors*, which are generalizations of *morphisms* and *actions*. Functors are maps from one category to another that “preserve the structure”, in the sense that the application of a functor commutes with category composition.

<b>19.1 Layers of abstraction . . . . .</b>	<b>306</b>
<b>19.2 Semifunctors . . . . .</b>	<b>307</b>
<b>19.3 Functors . . . . .</b>	<b>308</b>
<b>19.4 More examples of functors . . . .</b>	<b>311</b>
<b>19.5 Categorical Databases . . . . .</b>	<b>315</b>

Watch: *Big picture, zipping* (3 minutes).



## 19.1. Layers of abstraction

We can think of a given category **C** as a “compositional world”: inside **C** we have objects, morphisms between them, and a way to talk about composing morphisms. Now we will zoom out a level, and consider different categories – different worlds – simultaneously, and how to relate them to each other.

The most basic notion of how to “map” one category to another is given by the concept of a *functor*.

Just like a morphism

$$f : X \rightarrow Y \tag{1}$$

is an arrow between objects in a category, a functor

$$F : \mathbf{C} \rightarrow \mathbf{D} \tag{2}$$

is an arrow between two categories. In fact, we will see in the next chapters that *functors are morphisms in a category of categories*.

Later on, we will see that there is yet another level of generalization: we can define *arrows between functors*, called natural transformations.

## 19.2. Semifunctors

### Definition 19.1 (Semifunctor)

Given semicategories  $\mathbf{C}$  and  $\mathbf{D}$ , a *semifunctor*  $F : \mathbf{C} \rightarrow \mathbf{D}$  from  $\mathbf{C}$  to  $\mathbf{D}$  is defined by:

#### Constituents

1. A map

$$F_{\bullet} : \text{Ob}_{\mathbf{C}} \rightarrow \text{Ob}_{\mathbf{D}}. \quad (3)$$

2. For every pair of objects  $X, Y \in \text{Ob}_{\mathbf{C}}$  a map

$$F_{\rightarrow} : \text{Hom}_{\mathbf{C}}(X; Y) \rightarrow \text{Hom}_{\mathbf{D}}(F_{\bullet}(X); F_{\bullet}(Y)). \quad (4)$$

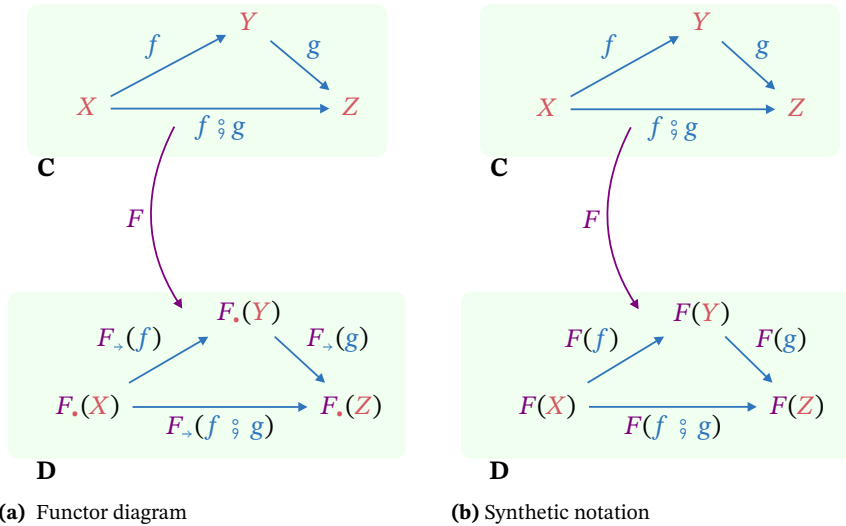
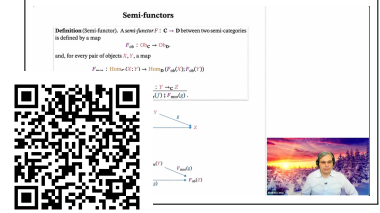
#### Conditions

1. Semifunctor application to morphisms is compatible with the respective category composition operations:

$$\frac{f : X \rightarrow_{\mathbf{C}} Y \quad g : Y \rightarrow_{\mathbf{C}} Z}{F_{\rightarrow}(f \circ_{\mathbf{C}} g) = F_{\rightarrow}(f) \circ_{\mathbf{D}} F_{\rightarrow}(g)}. \quad (5)$$

This situation is depicted graphically in Fig. 1a. It is common to overload the notation and use  $F$  to mean both  $F_{\bullet}$  and  $F_{\rightarrow}$ . The diagram with this overloaded “synthetic notation” is in Fig. 1b.

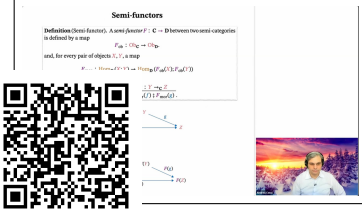
Watch: *Definition of semi-functor* (4 minutes).



**Figure 1.:** Commuting diagrams for semifunctors, with verbose notation (left) and synthetic notation (right).



Watch: *Definition of functor* (1 minutes).



## 19.3. Functors

For categories, we have the stronger concept of functor. Categories have identities, and functors are required to preserve the identities.

### Definition 19.2 (Functor)

A *functor* from category  $\mathbf{C}$  to a category  $\mathbf{D}$  is a semifunctor  $F : \mathbf{C} \rightarrow \mathbf{D}$  that satisfies the condition

$$F_{\rightarrow}(\text{id}_X) = \text{id}_{F_{\rightarrow}(X)} \quad (6)$$

for all objects  $X$  in  $\mathbf{C}$ .

**Example 19.3** (Powerset functor). We define a functor  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  which maps each set to its power set. In other words,  $F_{\rightarrow}(\mathbf{A}) = \text{Pow } \mathbf{A}$  for any set  $\mathbf{A}$ . On the level of morphisms, given a function  $f : \mathbf{A} \rightarrow \mathbf{B}$ , we define

$$\begin{aligned} F_{\rightarrow}(f) : \text{Pow } \mathbf{A} &\rightarrow \text{Pow } \mathbf{B} \\ \mathbf{C} &\mapsto \{f(c) \mid c \in \mathbf{C}\}. \end{aligned} \quad (7)$$

Here is a concrete illustration. Consider the two sets  $\mathbf{A} = \{\text{🍷}, \text{🍷}, \text{🍷}\}$  and  $\mathbf{B} = \{\text{🍌}, \text{🍇}, \text{🍏}\}$ . Applying the functor to  $\mathbf{A}$  gives

$$F_{\rightarrow}(\mathbf{A}) = \{\emptyset, \{\text{🍷}\}, \{\text{🍷}\}, \{\text{🍷}\}, \{\text{🍷}, \text{🍷}\}, \{\text{🍷}, \text{🍷}\}, \{\text{🍷}, \text{🍷}\}, \{\text{🍷}, \text{🍷}, \text{🍷}\}\} \quad (8)$$

and applying it to  $\mathbf{B}$  gives

$$F_{\rightarrow}(\mathbf{B}) = \{\emptyset, \{\text{🍌}\}, \{\text{🍇}\}, \{\text{🍏}\}, \{\text{🍌}, \text{🍇}\}, \{\text{🍌}, \text{🍏}\}, \{\text{🍇}, \text{🍏}\}, \{\text{🍌}, \text{🍇}, \text{🍏}\}\}. \quad (9)$$

Furthermore, consider the map

$$\begin{aligned} f : \mathbf{A} &\rightarrow \mathbf{B}, \\ \text{🍷} &\mapsto \text{🍌}, \\ \text{🍷} &\mapsto \text{🍇}, \\ \text{🍷} &\mapsto \text{🍏}. \end{aligned} \quad (10)$$

This would for instance give  $F_{\rightarrow}(f)(\{\text{🍷}, \text{🍷}\}) = \{f(\text{🍷}), f(\text{🍷})\} = \{\text{🍌}, \text{🍇}\}$ .

Now let us check that  $F$  so-defined really is a functor.

First let us check that it is compatible with composition. Consider functions  $f : \mathbf{A} \rightarrow \mathbf{B}$ ,  $g : \mathbf{B} \rightarrow \mathbf{C}$ . On the one hand we have

$$F_{\rightarrow}(f \circ g)(\mathbf{C}) = \{g(f(c)) \mid c \in \mathbf{C}\}, \quad (11)$$

and on the other hand

$$\begin{aligned} (F_{\rightarrow}(f) \circ F_{\rightarrow}(g))(\mathbf{C}) &= F_{\rightarrow}(g)(\{f(c) \mid c \in \mathbf{C}\}) \\ &= \{g(d) \mid d \in \{f(c) \mid c \in \mathbf{C}\}\} \\ &= \{g(f(c)) \mid c \in \mathbf{C}\}. \end{aligned} \quad (12)$$

Second, let us check that  $F$  is compatible with identity morphisms. We have

$$\begin{aligned} F_{\rightarrow}(\text{id}_{\mathbf{A}})(\mathbf{C}) &= \{\text{id}_{\mathbf{A}}(c) \mid c \in \mathbf{C}\} \\ &= \text{id}_{F_{\rightarrow}(\mathbf{C})}. \end{aligned} \quad (13)$$

**Example 19.4.** There is a functor

$$\text{List} : \mathbf{Set} \rightarrow \mathbf{Mon} \quad (14)$$

from the category of sets to the category of monoids, defined as follows.

Given a set  $\mathbf{A}$ , the functor returns a specific monoid

$$\mathbf{List}(\mathbf{A}) := \langle \mathbf{List} \mathbf{A}, [\ ]_{\mathbf{A}}, \circ \rangle. \quad (15)$$

Given a map  $f : \mathbf{A} \rightarrow \mathbf{B}$ , we have

$$\begin{aligned} \mathbf{List}(f) : \mathbf{List} \mathbf{A} &\rightarrow \mathbf{List} \mathbf{B}, \\ [\dots, a_i, \dots] &\mapsto [\dots, f(a_i), \dots], \end{aligned} \quad (16)$$

which applies  $f$  entry-wise in the list. The empty list in  $\mathbf{List} \mathbf{A}$  is mapped to the empty list in  $\mathbf{List} \mathbf{B}$ .  $\mathbf{List}$  is a functor, because identity functions in  $\mathbf{Set}$  are mapped to the corresponding identity morphisms on lists, and

$$\begin{aligned} \mathbf{List}(f \circ g)([\dots, a_i, \dots]) &= [\dots, (f \circ g)(a_i), \dots] \\ &= [\dots, g(f(a_i)), \dots] \\ &= \mathbf{List}(g)([\dots, f(a_i), \dots]) \\ &= (\mathbf{List}(f) \circ \mathbf{List}(g))([\dots, a_i, \dots]). \end{aligned} \quad (17)$$

#### Graded exercise F.1 (DifferentiationFunctor)

Consider the category  $\mathbf{Euc}_*$  from Graded Exercise E.3, as well as the category  $\mathbf{Vect}_{\mathbb{R}}$  of real vector spaces studied in Graded Exercise E.2. In this exercise we will define a functor  $F : \mathbf{Euc}_* \rightarrow \mathbf{Vect}_{\mathbb{R}}$  corresponding to differentiation and it is your task to check that it is in fact a functor.

$F$  on objects:  $F(\langle \mathbb{R}^n, x \rangle) = \mathbb{R}^n$ .

$F$  on morphisms: given a morphism  $f : \langle \mathbb{R}^n, x \rangle \rightarrow \langle \mathbb{R}^m, y \rangle$  in  $\mathbf{Euc}_*$ , the linear map  $F_{\rightarrow}(f)$  is the derivative

$$Df|_x : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad (18)$$

which is typically represented by the Jacobian matrix.

The intuition for the on-objects part of this functor is that  $\langle \mathbb{R}^n, x \rangle$  is mapped to the tangent space of “vectors starting at  $x$ ”, which is isomorphic to  $\mathbb{R}^n$ . On the level of morphisms, the differential of a function  $f$  at  $x$  maps vectors starting at  $x$  to vectors starting at  $f(x)$ .

#### Graded exercise F.2 (FixedPointFunctor)

We will propose a functor  $F : \mathbf{EndSet} \rightarrow \mathbf{Set}$  from the category  $\mathbf{EndSet}$  defined in Graded Exercise E.4 to the category of sets and functions. Your task is to check if this is a functor.

On objects: given an object  $\langle \mathbf{A}, \varphi \rangle$  of  $\mathbf{EndSet}$ , we define  $F(\langle \mathbf{A}, \varphi \rangle) = \text{Fix}(\varphi)$ , where

$$\text{Fix}(\varphi) = \{x \in \mathbf{A} \mid \varphi(x) = x\} \quad (19)$$

is the set of fixed points of  $\varphi$ .

On morphisms: given a morphism  $f : \langle \mathbf{A}, \varphi \rangle \rightarrow \mathbf{EndSet} \langle \mathbf{B}, \psi \rangle$  of  $\mathbf{EndSet}$ , we define

$$F_{\rightarrow}(f) = f|_{\text{Fix}(\varphi)}. \quad (20)$$

In other words, we restrict  $f$  to the subset  $\text{Fix}(\varphi) \subseteq \mathbf{A}$ .

A functor from a category to itself is called an *endofunctor*. The simplest example of an endofunctor is the identity functor.

## Identity functor

### Definition 19.5 (Identity (semi)functor)

For any (semi)category  $\mathbf{C}$ , we can define the *identity (semi)functor*

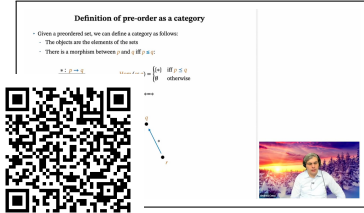
$$\text{id}_{\mathbf{C}} : \mathbf{C} \rightarrow \mathbf{C}, \quad (21)$$

which maps each object to itself and each morphism to itself.

**Exercise 35.** Check that the identity functor is a functor.

*See solution on page 331.*

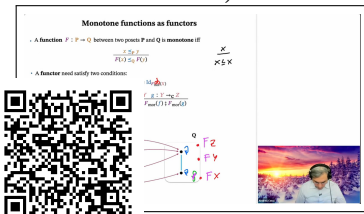
Watch: *Pre-order as a category* (2 minutes).



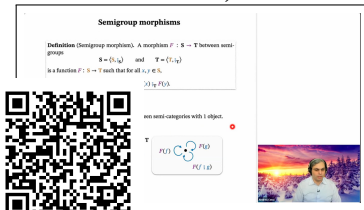
Watch: *The skeleton of a pre-order is a poset* (2 minutes).



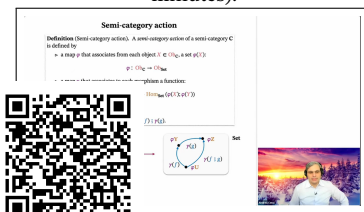
Watch: *Monotone functions as functors* (6 minutes).



Watch: *Semigroup morphisms as semi-functors* (2 minutes).



Watch: *Semi-category actions as functors* (2 minutes).



## Functors generalize monoid morphisms.

We have seen that monoids can be viewed as categories, where the elements of a monoid play the role of morphisms. From this point of view, a functor corresponds to a morphism of monoids.

## Functors generalize monotone maps.

Recall that a single poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  can be viewed as a category  $\mathbf{C}(\mathbf{P})$ , in which each element of the poset is an object, and there is a morphism between two objects  $x$  and  $y$  if and only if  $x \leq_{\mathbf{P}} y$  (Def. 14.16).

**Lemma 19.6.** A monotone map between posets  $\mathbf{P}, \mathbf{Q}$  is the same thing as a functor between the “posetal categories”  $\mathbf{C}(\mathbf{P})$  and  $\mathbf{C}(\mathbf{Q})$ .

*Proof.* We start by specifying the functor  $F$  and two posetal categories  $\mathbf{C}(\mathbf{P})$  and  $\mathbf{C}(\mathbf{Q})$ . We first specify the action of  $F$  on objects (elements of a poset considered as objects of the posetal category) and on morphisms (order relations considered as morphisms of the posetal category). A monotone function maps each element of a poset  $x \in \mathbf{P}$  to  $F(x) \in \mathbf{Q}$ , and it guarantees that for every  $x, y \in \mathbf{P}$ , if  $x \leq_{\mathbf{P}} y$  then  $F(x) \leq_{\mathbf{Q}} F(y)$ . We now need to check the two conditions that a functor must satisfy. First, consider the identity morphism for  $x \in \mathbf{P}$ , namely  $x \leq_{\mathbf{P}} x$ . The application of the map  $F$  results in the condition  $F(x) \leq_{\mathbf{Q}} F(x)$ , which is the identity morphism on  $\mathbf{Q}$ . Second, morphisms  $x \leq_{\mathbf{P}} y$  and  $y \leq_{\mathbf{P}} z$  in  $\mathbf{P}$ , by applying the map  $F$  to the morphism composition  $x \leq_{\mathbf{P}} z$  we obtain  $F(x) \leq_{\mathbf{Q}} F(z)$ , which is the same as the composition of  $F(x) \leq_{\mathbf{Q}} F(y)$  and  $F(y) \leq_{\mathbf{Q}} F(z)$ .  $\square$

## Functors generalize semicategory actions

Semi-functors are a generalization of the various semigroup morphisms that we saw in the previous chapter.

In particular, they are a generalization of semicategory actions (Def. 16.10), which we can re-define as follows.

### Definition 19.7 (Semicategory actions, redefined)

A *semicategory action* of  $\mathbf{C}$  is a semifunctor  $F : \mathbf{C} \rightarrow \mathbf{Set}$ .

## 19.4. More examples of functors

**Example 19.8** (Equivalence Classes). Consider the category **EquivRel** from Def. 14.11. We define a functor

$$F : \mathbf{EquivRel} \rightarrow \mathbf{Set} \quad (22)$$

as follows.

Given an object  $\langle \mathbf{A}, \sim_{\mathbf{A}} \rangle$  of **EquivRel** (that is, a set equipped with an equivalence relation), we define  $F(\langle \mathbf{A}, \sim_{\mathbf{A}} \rangle) = \mathbf{A} / \sim_{\mathbf{A}}$  to be the quotient of  $\mathbf{A}$  by  $\sim_{\mathbf{A}}$ .

Recall that  $\mathbf{A} / \sim_{\mathbf{A}}$  is the set of equivalence classes of  $\sim_{\mathbf{A}}$ . Its elements are the subsets of  $\mathbf{A}$  which form the partition of  $\mathbf{A}$  induced by  $\sim_{\mathbf{A}}$ . Each such subset is the set of all elements of  $\mathbf{A}$  which are mutually equivalent to each other according to  $\sim_{\mathbf{A}}$ . For any element  $x \in \mathbf{A}$ , the equivalence class it belongs to is denoted  $[x]$  and in this case  $x$  is called a *representative* of the equivalence class  $[x]$ .

To define  $F_{\rightarrow}$ , let  $f : \langle \mathbf{A}, \sim_{\mathbf{A}} \rangle \rightarrow \langle \mathbf{B}, \sim_{\mathbf{B}} \rangle$  be a morphism in **EquivRel**. We let

$$F_{\rightarrow}(f) : \mathbf{A} / \sim_{\mathbf{A}} \rightarrow \mathbf{B} / \sim_{\mathbf{B}}, [x] \mapsto [f(x)]. \quad (23)$$

It may be readily checked that this function is well-defined, irrespective of the (arbitrary) choice of element  $x$  used to represent a given equivalence class  $[x]$ .

**Exercise 36.** Prove that the functor defined in Example 19.8 is in fact a functor.

*See solution on page 331.*

**Example 19.9** (Double dual). Let  $\mathbf{Vect}_{\mathbb{R}}$  be the category whose objects are all real vector spaces and whose morphisms are  $\mathbb{R}$ -linear maps. Composition is the usual composition of linear maps.

There is an endofunctor  $F : \mathbf{Vect}_{\mathbb{R}} \rightarrow \mathbf{Vect}_{\mathbb{R}}$  whose action on objects is

$$F(V) = V^{**}. \quad (24)$$

(Recall that  $V^{**} = \{\text{linear maps } V^* \rightarrow \mathbb{R}\} = \mathbf{Hom}_{\mathbf{Vect}_{\mathbb{R}}}(V^*; \mathbb{R})$ ). The action of  $F$  on morphisms is as follows. Given a linear map  $f : V \rightarrow W$ , we can write

$$\begin{aligned} F(f) : V^{**} &\rightarrow W^{**}, \\ \xi &\mapsto [l \mapsto \xi(f \circ l)]. \end{aligned} \quad (25)$$

### Graded exercise F.3 (DoubleDualFunctor)

Prove that  $F$  as defined in Example 19.9 is in fact a functor.

**Example 19.10.** Let  $\mathbf{G}$  be a group. We've seen that we can view  $\mathbf{G}$  as a category with just a single object (denote that object by  $\star$ ) and where the morphisms are the elements of  $\mathbf{G}$  (with composition given by the group's composition operation). We claim that specifying a functor  $\mathbf{G} \rightarrow \mathbf{Set}$  is “the same thing” as specifying a set  $\mathbf{A}$  together with a (covariant) action of  $\mathbf{G}$  on  $\mathbf{A}$ .

**Exercise 37.** Prove the claim made in Example 19.10.

*See solution on page 331.*

### Graded exercise F.4 (MultiplicationWithASet)

Let  $\mathbf{S}$  be a fixed set. We define a functor  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  which acts on objects by

$$F(\mathbf{A}) = \mathbf{A} \times \mathbf{S} \quad (26)$$

and on morphisms by

$$F_{\rightarrow}(f) = f \times \text{id}_{\mathbf{S}}. \quad (27)$$

Prove that this is indeed a functor.

**Graded exercise F.5** (ExponentiationWithASet)

Let  $\mathbf{S}$  be a fixed set. We can define a functor  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  which acts on objects by

$$F_{\bullet}(\mathbf{A}) = \mathbf{A}^{\mathbf{S}}. \quad (28)$$

(Recall that  $\mathbf{A}^{\mathbf{S}}$  denotes the set of functions from  $\mathbf{S}$  to  $\mathbf{A}$ .) For the action on morphisms, suppose we have a function  $F : \mathbf{A} \rightarrow \mathbf{B}$ . Then

$$\begin{aligned} F_{\rightarrow}(f) : \mathbf{A}^{\mathbf{S}} &\rightarrow \mathbf{B}^{\mathbf{S}}, \\ \varphi &\mapsto \varphi \circ f. \end{aligned} \quad (29)$$

Prove that this does really define a functor.

**Graded exercise F.6** (GraphsViaFunctors)

Consider the following category, which has two objects,  $\mathbf{V}$  and  $\mathbf{A}$ , and four morphisms: besides the identity morphisms, there are two morphisms,  $s$  and  $t$ , from  $\mathbf{A}$  to  $\mathbf{V}$ . See Fig. 2. Call this category  $\mathbf{G}$ .

Can you explain the following statement? “Specifying a functor  $\mathbf{G} \rightarrow \mathbf{Set}$  is the “same thing” as specifying a directed graph”.

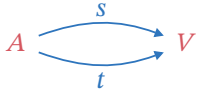


Figure 2.

**Graded exercise F.7** (UpperSetsViaFunctors)

Recall that  $\mathbf{Bool}$  denotes the category with two objects,  $\top$  and  $\perp$ , and with precisely one non-identity morphism which goes from  $\perp$  to  $\top$ . Let  $\mathbf{P}$  be a poset. View it as a category  $\mathbf{P}$ , and let  $F : \mathbf{P} \rightarrow \mathbf{Bool}$  be a functor. In other words,  $F = F_{\bullet}$  is a monotone function. Prove that the set

$$\mathbf{S} := \{p \in \mathbf{P} \mid F(p) = \top\} \subseteq \mathbf{P} \quad (30)$$

is an upper set.

**Graded exercise F.8** (CartProdAsFunctor)

Recall that, given categories  $\mathbf{C}$  and  $\mathbf{D}$ , we can form the product category  $\mathbf{C} \times \mathbf{D}$ . In this exercise we will use this construction in a situation where we consider the product of the category  $\mathbf{Set}$  with itself.

Your task in this exercise is to show that there is a functor  $F : \mathbf{Set} \times \mathbf{Set} \rightarrow \mathbf{Set}$  which is defined by the operation of “taking the cartesian product of sets and functions”.

Concretely, given any two sets  $\mathbf{A}, \mathbf{B}$ , let

$$F_{\bullet}(\langle \mathbf{A}, \mathbf{B} \rangle) := \mathbf{A} \times \mathbf{B}$$

(this defines  $F$  on the level of objects) and given any two functions  $f : \mathbf{A} \rightarrow \mathbf{C}, g : \mathbf{B} \rightarrow \mathbf{D}$ , let

$$F_{\rightarrow}(\langle f, g \rangle) := f \times g$$

(this defines  $F$  on the level of morphisms).

To show that  $F$  is in fact a functor, check the two conditions in the definition of

a functor, namely that  $F$  is compatible with composition and with identities.

### Graded exercise F.9 (ProbabilityFunctor)

In this exercise we will define a functor  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  and your task is to check that it is in fact a functor.

$F$  on objects: for any set  $\mathbf{A}$  we define  $F_*(\mathbf{A})$  to be the set of *finitely supported probability measures* on  $\mathbf{A}$ . These are functions  $p : \mathbf{A} \rightarrow [0, 1]$  with only finitely-many non-zero values and such that these sum to 1:

$$\sum_{x \in \mathbf{A}} p(x) = 1. \quad (31)$$

$F$  on morphisms: for any function  $f : \mathbf{A} \rightarrow \mathbf{B}$ , the function  $F_*(f) : F_*(\mathbf{A}) \rightarrow F_*(\mathbf{B})$  is defined by

$$\begin{aligned} F_*(f)(p) : \mathbf{B} &\rightarrow [0, 1], \\ y &\mapsto \sum_{x \in f^{-1}(y)} p(x). \end{aligned} \quad (32)$$

The following is a helpful visualization. We may think of finitely-supported probability measures on  $\mathbf{A}$  as “finite normalized histograms over the elements of  $\mathbf{A}$ ”, and the functor  $F$  moves the columns of a histogram on  $\mathbf{A}$  “along”  $f$  to make a histogram on  $\mathbf{B}$ , stacking columns on top of each other whenever they end up over the same element. (Credit: this nice description is taken from Paolo Perrone’s *Notes on category theory*.)

**Example 19.11.** Let  $\mathbf{C}$  be any category. There is a functor

$$F : \mathbf{C}^{\text{op}} \times \mathbf{C} \rightarrow \mathbf{Set} \quad (33)$$

defined on objects by

$$F_*(\langle X, Y \rangle) = \text{Hom}_{\mathbf{C}}(X, Y). \quad (34)$$

To see how  $F$  is defined on morphisms, let  $f^{\text{op}} : U \rightarrow_{\mathbf{C}^{\text{op}}} X$  and  $g : Y \rightarrow_{\mathbf{C}} Z$  be morphisms in  $\mathbf{C}^{\text{op}}$  and  $\mathbf{C}$ , respectively, and recall that  $f^{\text{op}} : U \rightarrow_{\mathbf{C}^{\text{op}}} X$  is, by definition, a morphism  $f : X \rightarrow_{\mathbf{C}} U$ . We define  $F_*(\langle f^{\text{op}}, g \rangle)$  to be the function

$$\begin{aligned} F_*(\langle f^{\text{op}}, g \rangle) : \text{Hom}_{\mathbf{C}}(X, Y) &\rightarrow \text{Hom}_{\mathbf{C}}(U, Z), \\ \varphi &\mapsto f \circ \varphi \circ g. \end{aligned} \quad (35)$$

### Graded exercise F.10 (HomFunctor)

Prove that the hom-functor defined in Example 19.11 is in fact a functor.

## Planning as the search for a functor

**Example 19.12.** Recall the category **Berg** introduced in Section 15.2 and define a category **Plans** where objects are activities and morphisms describe activities order constraints, illustrated in Fig. 3 (left).

For instance, there is a morphism from “mountain lodge” to “panoramic lake”, which describes the plan of going from the lodge area to the lake area. We call such morphisms *plans*. Plans can be composed via concatenation. For instance, given a plan to go from “mountain lodge” to “panoramic lake”, and a plan to go

from “panoramic lake” to “peak”, their composition is the plan of going from “mountain lodge” to the “peak”, passing through the “panoramic lake”.

When we talk about “planning” in this context, we refer to the action of finding a functor  $F$  from **Plans** to **Berg**. The objects of **Berg** are tuples  $\langle p, v \rangle$ , where  $p$  represent coordinates of a specific location and  $v \in \mathbb{R}^3$  represents velocities. Morphisms in **Berg** are paths that connect locations. For the sake of our planning, we can identify areas of the mountain as sets of locations. Such areas are, for instance, the “mountain lodge”, “panoramic lake”, and the “peak” (note that the “peak” represents an area corresponding to a single location). Given some plans as in Fig. 3 (left), we want to find a map  $P$  which maps each object in **Plans** (activity) to an object of **Berg** (specific location and velocity). Similarly, it must map each morphism in **Plans** (activity order constraints) to a morphism in **Berg** (specific paths). This is illustrated in Fig. 3.

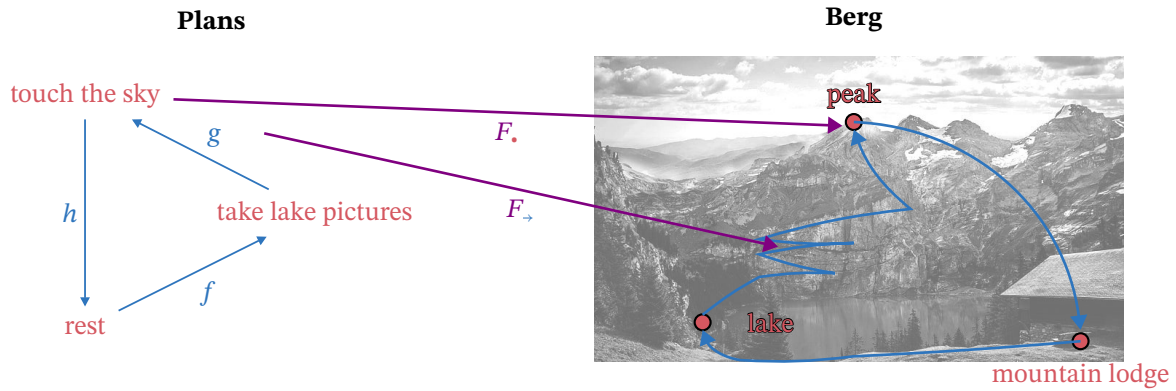


Figure 3.: Planning functor.

## 19.5. Categorical Databases

In this section we look at how a relational database can be modeled using the notion of functors. This view of databases is due to Spivak [30, 31].

To model a database, we need to model two things:

1. The *schema* of the database is its structure. This includes which tables are present, what fields are included in each table, and what constraints there are among tables.
2. The *data* that resides in the database.

Spivak's idea is that categories can be used for modeling the schema, not the data. The data can be modeled as a *functor*.







# 20. Specialization

Oftentimes, two categories are in a special relation: one is a specialization of the other. There are two interesting situations.

First, a category can be a *subcategory* of another if it contains a subset of the objects and morphisms of the other, in such a way that it is closed to composition.

Second, there is a generalization called *embedding*: objects and morphisms are different, but we can find a functor that “draws” the first category in the second.

20.1 Subcategories . . . . . 318

20.2 Subcategories of endomorphisms 319

20.3 Other examples . . . . . 320

20.4 Subcategories of Berg . . . . . 321

The *Pontifical Swiss Guard* is an armed forces that protects the Pope and the Apostolic Palace, serving as the military of Vatican City. Rectruits to the guards must be Catholic, single males with Swiss citizenship, who have completed basic training with the Swiss Armed Forces.

## 20.1. Subcategories

### Definition 20.1 (Subcategory)

A *sub(semi)category*  $\mathbf{D}$  of a (semi)category  $\mathbf{C}$  is a category for which:

1. All the objects in  $\mathbf{Ob}_{\mathbf{D}}$  are in  $\mathbf{Ob}_{\mathbf{C}}$ ;
2. For any two objects  $X, Y \in \mathbf{Ob}_{\mathbf{D}}$ , the morphisms of  $\mathbf{D}$  between them are a subset of the morphisms of  $\mathbf{C}$ :

$$\mathbf{Hom}_{\mathbf{D}}(X; Y) \subseteq \mathbf{Hom}_{\mathbf{C}}(X; Y); \quad (1)$$

3. If  $f : X \rightarrow Y$  and  $g : Y \rightarrow Z$  in  $\mathbf{D}$ , then the composite  $f \circ g$  in  $\mathbf{C}$  is in  $\mathbf{D}$  and represents the composite in  $\mathbf{D}$ .
4. (Categories) If  $X \in \mathbf{Ob}_{\mathbf{D}}$ , then the identity  $\text{id}_X$  in  $\mathbf{Hom}_{\mathbf{C}}(X; X)$  is also in  $\mathbf{Hom}_{\mathbf{D}}(X; X)$  and acts as its identity morphism.

### Subcategories of Rel and Set

Two important examples of subcategory are the following.

**Example 20.2 (FinSet).** **FinSet** is the category of finite sets and all functions between them. It is a subcategory of the category **Set** of sets and functions. While an object  $X \in \mathbf{Ob}_{\mathbf{Set}}$  is a set with arbitrary cardinality,  $\mathbf{Ob}_{\mathbf{FinSet}}$  only includes sets which have finitely many elements. Objects of **FinSet** are in **Set**, but the converse is not true. Furthermore, given  $X, Y \in \mathbf{Ob}_{\mathbf{FinSet}}$ , we take  $\mathbf{Hom}_{\mathbf{FinSet}}(X; Y) = \mathbf{Hom}_{\mathbf{Set}}(X; Y)$ .

**Example 20.3 (Set and Rel).** The category **Set** is a subcategory of **Rel**. To show this, we need to prove the conditions presented in Def. 20.1.

1. In both **Rel** and **Set**, the collection of objects is all sets.
2. Given  $X, Y \in \mathbf{Ob}_{\mathbf{Set}}$ , we know that  $\mathbf{Hom}_{\mathbf{Set}}(X; Y) \subseteq \mathbf{Hom}_{\mathbf{Rel}}(X; Y)$ , since all functions between sets  $X, Y$  are a particular subset of all relations between  $X, Y$ .
3. Let  $R \subseteq X \times Y$  and  $S \subseteq Y \times Z$  be relations which are functions. We need to show that their composition in **Rel**, expressed as  $R \circ S \subseteq X \times Z$ , is again a function. This was proven in Lemma 5.8.
4. For each  $X \in \mathbf{Ob}_{\mathbf{Set}}$ , the identity relation  $\text{id}_X = \{\langle x, x' \rangle \in X \times X \mid x = x'\}$  corresponds to the identity function  $\text{id}_X : X \rightarrow X$  in **Set**.

## 20.2. Subcategories of endomorphisms

### Definition 20.4 (Drawings)

There exists a category **Draw** in which:

1. An object in  $\alpha \in \text{Ob}_{\text{Draw}}$  is a black-and-white drawing, that is a function  $\alpha : \mathbb{R}^2 \rightarrow \text{Bool}$ .
2. A morphism in  $\text{Hom}_{\text{Draw}}(\alpha; \beta)$  between two drawings  $\alpha$  and  $\beta$  is an invertible map  $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  such that  $\alpha(x) = \beta(f(x))$ .
3. The identity function at any object  $\alpha$  is the identity map on  $\mathbb{R}^2$ .
4. Composition is given by function composition.

**Exercise 38.** Check whether just considering

- ▷ affine invertible transformations, or
- ▷ rototranslations, or
- ▷ scalings, or
- ▷ translations, or
- ▷ rotations,

as morphisms forms a subcategory of **Draw**.

*See solution on page 331.*

## 20.3. Other examples of subcategories in engineering

In engineering, it is very common to look at specific types of functions; in many cases, the properties of a certain type of function are preserved by function composition, and so they form a category.

### InjSet is a subcategory of Set

**Example 20.5.** We can define a category **InjSet** that has the same objects as **Set** but restricts the morphisms to be *injective functions* (Def. 4.14). We want to show that **InjSet** is a subcategory of **Set**. Composition and identity morphisms are defined as in **Set**.

Since  $\text{Ob}_{\text{InjSet}} = \text{Ob}_{\text{Set}}$ , the first condition of Def. 20.1 is satisfied. Injective functions are a particular type of functions: this satisfies the second condition. Given  $X \in \text{Ob}_{\text{InjSet}}$ , the identity morphism  $\text{id}_X \in \text{Hom}_{\text{Set}}(X; X)$  corresponds to the identity morphism in  $\text{Hom}_{\text{InjSet}}(X; X)$ : the identity function is injective. This proves the fourth condition. To check the third condition, consider two morphisms  $f \in \text{Hom}_{\text{Set}}(X; Y)$ ,  $g \in \text{Hom}_{\text{Set}}(Y; Z)$  such that  $f \in \text{Hom}_{\text{InjSet}}(X; Y)$  and  $g \in \text{Hom}_{\text{InjSet}}(Y; Z)$ . From the injectivity of  $f, g$ , we know that given  $x, x' \in X$ ,

$$\frac{f(x) = f(x')}{x = x'}, \quad (2)$$

and  $y, y' \in Y$ ,

$$\frac{g(y) = g(y')}{y = y'}. \quad (3)$$

Furthermore, we have:

$$\frac{(f \circ g)(x) = (f \circ g)(x')}{\frac{f(x) = f(x')}{x = x'}}, \quad (4)$$

which proves the third condition of Def. 20.1: the composition of injective functions is injective.

## 20.4. Subcategories of Berg

Recall the category **Berg** presented in Section 15.2. In the following, we want to give both a positive and a negative example of subcategories related to **Berg**.

We start our discussion by introducing a *limited* version of **Berg**, called **Berg<sub>α</sub>**, which only considers paths (morphisms) in **Berg**, whose steepness does not exceed the critical value  $\alpha \in [0, 1]$ . Is **Berg<sub>α</sub>** a subcategory of **Berg**? We check the different conditions:

1. The constraint on the maximum steepness restricts the objects which are acceptable in **Berg<sub>α</sub>** via the identity morphisms of **Berg**. Indeed, recall that given an object  $\langle p, v \rangle \in \text{Ob}_{\text{Berg}}$ , the identity morphism is defined as  $1_{\langle p, v \rangle} = \langle \gamma, 0 \rangle$ , with  $\gamma(0) = p$  and  $\dot{\gamma}(0) = v$ . The steepness is computed via  $v$ . In particular, **Berg<sub>α</sub>** will only contain objects whose identity morphisms do not exceed the steepness constraint, In other words  $\text{Ob}_{\text{Berg}_\alpha} \subseteq \text{Ob}_{\text{Berg}}$ .
2. For  $X, Y \in \text{Ob}_{\text{BergAma}}$ , we know that paths satisfying the steepness constraint are specific paths in **Berg**:  $\text{Hom}_{\text{Berg}_\alpha} \subseteq \text{Hom}_{\text{Berg}}$ .
3. Given two morphisms  $f, g$  which can be composed in **Berg<sub>α</sub>**, the maximum steepness of their composition  $f \circ g$  is given by:

$$\text{MaxSteep}(f \circ g) = \max\{\text{MaxSteep}(f), \text{MaxSteep}(g)\} < \alpha. \quad (5)$$

4. The identity morphisms in **Berg** which satisfy the steepness constraint are, by definition, in **Berg<sub>α</sub>** and they act as identities there.

This shows that **Berg<sub>α</sub>** is a subcategory of **Berg**.

What would an example of non-subcategory of **Berg** be? We could try defining a new category **BergLazy**, which now discriminates morphisms based on the lengths of the paths they represent. For instance, assume that as amateur hikers, we don't want to consider morphisms which are more than 1 km long. By concatenating two paths (morphisms) of length 0.6 km in **BergLazy**, the resulting composition will be 1.2 km, violating the posed constraint and hence not being in **BergLazy**. This violates the third property of Def. 20.1.







## 21. Syntax and semantics

To write...

21.1 Specification verses behavior . . . 324

*Raclette* is a Swiss dish, based on heating cheese and scraping off the melted part. In Switzerland, raclette is typically served with potatoes, cornichons, pickled onions, and Fendant wine.



## 21.1. Specification verses behavior

### Systems

In the following discussion we will often use the word “system”. This term here is not a precise one; however, it is often a way of saying something like “engineering-flavored morphism” or “system component” or “dynamical system”, but without necessarily getting precise about which kind of things we are exactly referring to or which (semi)categories might be involved.

We will think of a system as something that has input and output ports with which it can interact with other systems (or its broader environment), and that a system in some way establishes a relationship between input and output signals. This relationship might be a deterministic, causal relationship – inputs determining outputs – or it might be another form of lawfulness.

A typical notation to depict a system diagrammatically is to draw it as a box, with externally extended wires indicating the input and output ports. We will usually orient such diagrams horizontally, and assume that the left-hand wires indicate input ports, and right-hand wires indicate output ports.

### Composing systems

We assume that our concept of system is compositional: systems can be connected together to build larger, composite systems.

An illustrative example of systems are Moore machines, which we discussed in some length in Chapter 16. Given a Moore machine

$$\langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle \quad (1)$$

with an input set  $\mathbf{U}$  and an output  $\mathbf{Y}$ . When the output set of one machine matches the input set of another, we have already seen how to compose Moore machines in series such that the result is again a Moore machine.

### System specification vs. system behavior

We will make a distinction between ways of specifying a system, and ways of describing how a system might behave.

To see what we mean, consider the example of Moore machines. A way to specify a Moore machine, according to our Def. 16.5, is to specify a tuple of the form (1).

On the other hand, we saw that Moore machines can act on sequences or lists of signals in various ways. We think of these actions as encoding “behaviors” that a Moore machine can exhibit. The idea is that an action encodes a way that a Moore machine “does something” – how it relates input signals to output signals.

Strictly speaking, we will think of just the specific relation between inputs and outputs as a behavior, and an action is a way of associating specified machines to specified behaviors.

For example, given a machine

$$f : \mathbf{U} \rightarrow \mathbf{Y} \quad (2)$$

specified by (1), the standard action Def. 16.11 associates to it a behavior

$$\text{act}(f) : \text{Stream } \mathbf{U} \rightarrow \text{Stream } \mathbf{Y}. \quad (3)$$

We can study specifications of machines and possible behaviors of machines each in their own right, and we can study ways that specifications and behaviors can be connected.

In Section 16.6 we saw that different kinds of behaviors are possible, via different actions. For example, not only behaviors of the type

$$\text{Stream } \mathbf{U} \rightarrow \text{Stream } \mathbf{Y} \quad (4)$$

but also of the type

$$\text{List } \mathbf{U} \rightarrow \text{List } \mathbf{Y} \quad (5)$$

are possible.

In Section 16.8 below on LTI systems, we will see an example of how a single system might be specified in different ways.





## 22. Up the ladder of abstraction

In this section we “zoom out” one level and describe how functors are morphisms in a category of categories.

<b>22.1 Functor composition</b>	<b>328</b>
<b>22.2 A category of categories</b>	<b>329</b>
<b>22.3 Products and sums of functors</b>	<b>330</b>

## 22.1. Functor composition

### Definition 22.1 (Functor composition)

Consider categories  $\mathbf{A}, \mathbf{B}, \mathbf{C}$  and functors  $F : \mathbf{A} \rightarrow \mathbf{B}, G : \mathbf{B} \rightarrow \mathbf{C}$ . Functor composition is given by  $F \circ G : \mathbf{A} \rightarrow \mathbf{C}$ , where:

▷ Given  $X \in \text{Ob}_{\mathbf{A}}$ , we have

$$(F \circ G)(X) := G(F(X)). \quad (1)$$

▷ Given  $f \in \text{Hom}_{\mathbf{A}}(X; Y)$ , we have

$$(F \circ G)(f) := G(F(f)). \quad (2)$$

**Lemma 22.2.** The composition of functors is a functor.

**Exercise 39.** Prove Lemma 22.2.

*See solution on page 332.*

## 22.2. A category of categories

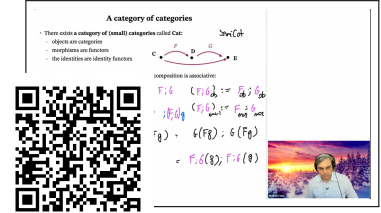
Given the existence of an identity functor and the ability of functors to compose, we can define a category of categories **Cat**. In order to avoid set-theoretic technicalities, we restrict our attention to so-called “small” categories: these are categories whose collection of objects form a set (and not a proper class).

### Definition 22.3 (Category of small categories)

There is a category, called **Cat**, which is constituted of

- ▷ Objects: small categories;
- ▷ Morphisms: functors;
- ▷ Identity morphisms: identity functors;
- ▷ Composition: composition of functors.

Watch: *A category of categories* (6 minutes).



### Graded exercise F.11 (CatProductCategorical)

Prove that the product category  $\mathbf{C} \times \mathbf{D}$  of two small categories “is” the categorical product of  $\mathbf{C}$  and  $\mathbf{D}$  within the category of small categories.

### 22.3. Products and sums of functors

**Definition 22.4** (Product of two functors)

Given functors  $F : \mathbf{A} \rightarrow \mathbf{B}$  and  $G : \mathbf{C} \rightarrow \mathbf{D}$ , their product is the functor

$$F \times G : \mathbf{A} \times \mathbf{C} \rightarrow \mathbf{B} \times \mathbf{D}$$

defined on objects by

$$(F \times G)(\langle X, Y \rangle) = \langle F_*(X), G_*(Y) \rangle \quad (3)$$

and on morphisms by

$$(F \times G)(\langle f, g \rangle) = \langle F_*(f), G_*(g) \rangle. \quad (4)$$

**Definition 22.5** (Sum of two functors)

Given functors  $F : \mathbf{A} \rightarrow \mathbf{B}$  and  $G : \mathbf{C} \rightarrow \mathbf{D}$ , their sum is the functor

$$F + G : \mathbf{A} + \mathbf{C} \rightarrow \mathbf{B} + \mathbf{D}$$

defined on objects by

$$\begin{aligned} (F + G)(\langle 1, X \rangle) &= \langle 1, F_*(X) \rangle, \\ (F + G)(\langle 2, Y \rangle) &= \langle 2, G_*(Y) \rangle, \end{aligned} \quad (5)$$

and on morphisms by

$$\begin{aligned} (F + G)(\langle 1, f \rangle) &= \langle 1, F_*(f) \rangle, \\ (F + G)(\langle 2, g \rangle) &= \langle 2, G_*(g) \rangle. \end{aligned} \quad (6)$$

## Solutions to selected exercises

**Solution of Exercise 35.** To show that this is a valid functor, we need to show that it preserves identities and composition:

▷ Given any  $X \in \text{Ob}_{\mathbf{C}}$ , we have:

$$\begin{aligned} \text{id}_{\mathbf{C}}(\text{id}_X) &= \text{id}_X \\ &= \text{id}_{\text{id}_{\mathbf{C}}(X)} \end{aligned} \quad (7)$$

Furthermore, given composable morphisms  $f, g$  in  $\mathbf{C}$ , we have:

$$\begin{aligned} \text{id}_{\mathbf{C}}(f \circ g) &= f \circ g \\ &= \text{id}_{\mathbf{C}}(f) \circ \text{id}_{\mathbf{C}}(g). \end{aligned} \quad (8)$$

**Solution of Exercise 36.**

**Solution of Exercise 37.**

We check the specializations one by one. In all specializations, we consider the same objects as in **Draw**.

▷ **Scalings.** Let  $s, t \in \mathbb{R}$ . Scalings can be represented as functions of the form

$$\begin{aligned} \text{sc}_{s,t} : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \\ \langle x, y \rangle &\mapsto \langle sx, ty \rangle. \end{aligned} \quad (9)$$

By just considering morphisms which are scalings, we are considering a subset of all morphisms. Furthermore, the composition of two scalings is again a scaling. Indeed, consider scalings  $\text{sc}_{s,t}, \text{sc}_{u,v}$ . We have

$$\begin{aligned} (\text{sc}_{s,t} \circ \text{sc}_{u,v})(x, y) &= \text{sc}_{u,v}(sx, ty) \\ &= \langle usx, vty \rangle \\ &= \text{sc}_{us,vt}. \end{aligned} \quad (10)$$

Finally, the identity morphism in **Draw** corresponds to a scaling of the form  $\text{sc}_{1,1}$ .

▷ **Translations.** Let  $s, t \in \mathbb{R}$ . Translations are functions of the form

$$\begin{aligned} \text{tra}_{s,t} : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \\ \langle x, y \rangle &\mapsto \langle x + s, y + t \rangle. \end{aligned} \quad (11)$$

By just considering morphisms which are translations, we are considering a subset of all morphisms. Furthermore, the composition of two translations is again a translation. Indeed, consider scalings  $\text{tra}_{s,t}, \text{tra}_{u,v}$ . We have

$$\begin{aligned} (\text{tra}_{s,t} \circ \text{tra}_{u,v})(x, y) &= \text{tra}_{u,v}(x + s, y + t) \\ &= \langle x + s + u, y + t + v \rangle \\ &= \text{tra}_{s+u, t+v}. \end{aligned} \quad (12)$$

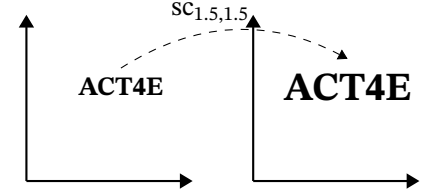
Finally, the identity morphism in **Draw** corresponds to a translation of the form  $\text{tra}_{0,0}$ .

▷ **Rotations.** Let  $\theta \in [0, 2\pi)$ . Rotations are functions of the form

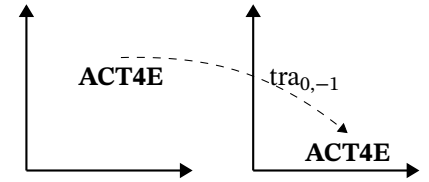
$$\begin{aligned} \text{rot}_{\theta} : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \\ \langle x, y \rangle &\mapsto \langle x \cos(\theta) + y \sin(\theta), y \cos(\theta) - x \sin(\theta) \rangle. \end{aligned} \quad (13)$$

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

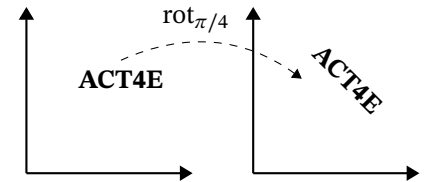
**Solution of Exercise 38.**



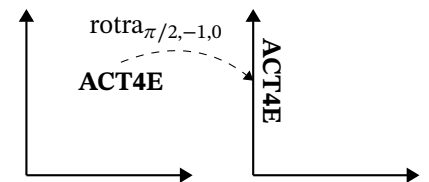
**Figure 1.:** Example of scaling.



**Figure 2.:** Example of translation.

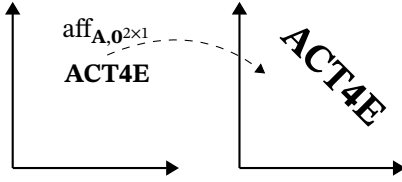


**Figure 3.:** Example of rotation.



**Figure 4.:** Example of rototranslation.





**Figure 5.:** Example of affine transformation with  $A = 1.5 \begin{bmatrix} \cos(\pi/4) & \sin(\pi/4) \\ -\sin(\pi/4) & \cos(\pi/4) \end{bmatrix}$ .

By just considering morphisms which are rotations, we are considering a subset of all morphisms. Furthermore, the composition of two rotations is again a rotation. Indeed, consider rotations  $\text{rot}_\theta, \text{rot}_\phi$ . We have

$$\begin{aligned} (\text{rot}_\theta \circ \text{rot}_\phi)(x, y) &= \text{rot}(\langle x \cos(\theta) + y \sin(\theta), y \cos(\theta) - x \sin(\theta) \rangle) \\ &= \langle x \cos(\theta + \phi) + y \sin(\theta + \phi), y \cos(\theta + \phi) - x \sin(\theta + \phi) \rangle \\ &= \text{rot}_{\theta+\phi}. \end{aligned} \quad (14)$$

Finally, the identity morphism in **Draw** corresponds to a rotation of the form  $\text{rot}_0$ .

- **Rototranslations.** Let  $s, t \in \mathbb{R}$  and  $\theta \in [0, 2\pi)$ . Rototranslations are functions arising from the combination of rotations and translations, and are therefore of the form:

$$\begin{aligned} \text{rotra}_{\theta,s,t} : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \\ \langle x, y \rangle &\mapsto \langle x \cos(\theta) + y \sin(\theta) + s, y \cos(\theta) - x \sin(\theta) + t \rangle. \end{aligned} \quad (15)$$

By just considering morphisms which are rotations, we are considering a subset of all morphisms. Furthermore, the composition of two rototranslations is again a rototranslation. Consider rototranslations  $\text{rotra}_{\theta,s,t}, \text{rotra}_{\phi,u,v}$ . We have:

$$\begin{aligned} (\text{rotra}_{\theta,s,t} \circ \text{rotra}_{\phi,u,v})(x, y) &= \text{rotra}_{\phi,u,v}(x \cos(\theta) + y \sin(\theta) + s, y \cos(\theta) - x \sin(\theta) + t) \\ &= \langle (x \cos(\theta) + y \sin(\theta) + s) \cos(\phi) + (y \cos(\theta) - x \sin(\theta) + t) \sin(\phi) + u, \\ &\quad (y \cos(\theta) - x \sin(\theta) + t) \cos(\phi) - (x \cos(\theta) + y \sin(\theta) + s) \sin(\phi) + v \rangle \\ &= \langle x \cos(\theta + \phi) + y \sin(\theta + \phi) + s \cos(\phi) + t \sin(\phi) + u, \\ &\quad y \cos(\theta + \phi) - x \sin(\theta + \phi) + t \cos(\phi) - s \sin(\phi) + v \rangle \\ &= \text{rotra}_{\theta+\phi, s \cos(\phi) + t \sin(\phi) + u, t \cos(\phi) - s \sin(\phi) + v}(x, y). \end{aligned} \quad (16)$$

Finally, the identity morphism in **Draw** corresponds to a rotation of the form  $\text{rotra}_{0,0,0}$ .

- **Affine transformations.** Let  $\mathbf{A} \in \mathbb{R}^{2 \times 2}$  and  $\mathbf{b} \in \mathbb{R}^{2 \times 1}$ . Affine transformations are functions that could arise from the combination of rotations and translations, and scalings, and are therefore of the form:

$$\begin{aligned} \text{aff}_{\mathbf{A},\mathbf{b}} : \mathbb{R}^2 &\rightarrow \mathbb{R}^2, \\ \langle x, y \rangle &\mapsto \langle a_{11}x + a_{12}y + b_{11}, a_{21}x + a_{22}y + b_{21} \rangle, \end{aligned} \quad (17)$$

where  $*_{ij}$  represents the element at the  $i$ -th row and  $j$ -th column of  $*$ .

Some special cases are:

- With  $\mathbf{A} = \mathbb{1}$  we obtain translations  $\text{tra}_{b_{11}, b_{21}}$ ;
- With  $\mathbf{b} = [0 \ 0]^T$  and  $A = \begin{bmatrix} s & 0 \\ 0 & t \end{bmatrix}$  we obtain scalings  $\text{sc}_{s,t}$ ;
- With  $\mathbf{b} = [0 \ 0]^T$  and  $A = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$  we obtain rotations  $\text{rot}_\theta$ .

By just considering morphisms which are affine transformations, we are considering a subset of all morphisms. Furthermore, the composition of two affine transformations is again an affine transformation. Clearly, the composition of affine transformations  $\text{aff}_{\mathbf{A},\mathbf{b}}, \text{aff}_{\mathbf{C},\mathbf{d}}$  is  $\text{aff}_{\mathbf{CA},\mathbf{Cb}+\mathbf{d}}$ . Finally, the identity morphism in **Draw** corresponds to an affine transformation of the form  $\text{aff}_{\mathbb{1},0^{2 \times 1}}$ .

**Solution of Exercise 39.** In the following, we want to show that functors com-

pose. Given categories  $\mathbf{A}, \mathbf{B}, \mathbf{C}$  and functors  $F : \mathbf{A} \rightarrow \mathbf{B}, G : \mathbf{B} \rightarrow \mathbf{C}$ , we want to show that  $F \circ G$  is a functor. To do this, we show that  $F \circ G$  preserves identities and compositions.

▷ Given an object  $X$  in  $\mathbf{A}$ , we have:

$$\begin{aligned} (F \circ G)(\text{id}_X) &= G(F(\text{id}_X)) \\ &= G(\text{id}_{F(X)}) \\ &= \text{id}_{G(F(X))}, \end{aligned} \tag{18}$$

where we used that  $F$  and  $G$  are functors (they preserve identities).

▷ Furthermore, given composable morphisms  $f, g$  in  $\mathbf{A}$ , we have:

$$\begin{aligned} (F \circ G)(f \circ g) &= G(F(f) \circ F(g)) \\ &= G(F(f)) \circ G(F(g)) \\ &= (F \circ G)(f) \circ (F \circ G)(g), \end{aligned} \tag{19}$$

where again we used that  $F, G$  are functors (they preserve composition).



# PART G.CO-DESIGN



---

<b>23. Design</b>	<b>337</b>
<b>24. Poset bounds</b>	<b>345</b>
<b>25. Monotone Co-Design Theory</b>	<b>357</b>
<b>26. Feasibility</b>	<b>377</b>
<b>27. Lattices</b>	<b>395</b>
<b>28. Lattice structure of DPs</b>	<b>405</b>
<b>29. Constructing design problems</b>	<b>415</b>

---

The *Sechseläuten* is a traditional spring holiday in the Zurich, Switzerland, usually happening on the 3rd monday of April. The old city guilds meet in the city center for a parade, climax of which is the burning of the “Böögg”, a snowman prepared with explosives, considered a weather oracle for the summer.





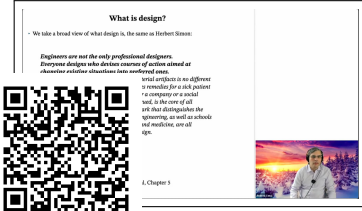
## 23. Design

This chapter introduces basic concepts of engineering design, and describes what are the design queries we want to answer.

23.1 What is “design”? . . . . .	338
23.2 What is “co-design”? . . . . .	339
23.3 Formal engineering design . . .	341
23.4 Queries in design . . . . .	343

Switzerland produces timepieces since the 14th century. Indeed, a “Swiss made” watch is very unique: according to official rules, the mechanics, casing, and inspection of a watch must be carried out in Switzerland to earn the hallmark. Today, Switzerland is the world’s largest watch exporter, counting over 60,000 employees and exporting over \$ 13.7 billion worth products.

Watch: *What is design?* (2 minutes).



## 23.1. What is “design”?

We take a broad view of what it means to “design”, that is not limited to engineering. Citing at length Hebert Simon’s\* *The sciences of the artificial* ([28], Chapter 5):

Engineers are not the only professional designers. Everyone designs who devises courses of action aimed at changing existing situations into preferred ones. The intellectual activity that produces material artifacts is no different fundamentally from the one that prescribes remedies for a sick patient or the one that devises a new sales plan for a company or a social welfare policy for a state. Design, so construed, is the core of all professional training; it is the principal mark that distinguishes the professions from the sciences. Schools of engineering, as well as schools of architecture, business, education, law, and medicine, are all centrally concerned with the process of design.

The metaphors used in the book are biased towards engineering. It is easy for everybody to imagine creating a physical machine out of simple components, and what choices and trade-offs we must deal with. Furthermore, it is easy to imagine what is the boundary between the machine and the world, that is, to delimit the design space.

Yet the theory to be discussed is applicable to other disciplines, if one takes a more abstract view of what is a system and a component. For example, in urban planning, the components of a city are roads, sewers, residential areas, *etc.* In other disciplines, “components” can be logical instead of physical. For example, an economist might ask how to design an incentive scheme such that such scheme (a “component”) will move the system to a more desirable set of states.

\* Hebert A. Simon (1916-2001). Winner of the 1978 Nobel Prize in Economics.

## 23.2. What is “co-design”?

The word “co-design” is not a new one. In this book, we will use a meaning that incorporates and extends the existing meaning.

We take the “Co” in “co-design” to have four meanings:

1. “co” for “compositional”;
2. “co” for “collaborative”;
3. “co” for “computational”;
4. “co” for “continuous”.

These meanings together describe the aspects of modern engineering design.

### “Co” for “compositional”

The first meaning has to do with composition:

co-design = design everything together

We use the word “co-design” to refer to any decision procedure that has to do with making simultaneous choices about the components of a system to achieve system-level goals. This includes the choice of components, the interconnection of components, and the configuration of components. We will see that in most cases, choices that are made at the level of components without looking at the entire system are doomed to be suboptimal. Slightly modifying Haiken’s quote in Section 1.2, we choose this as our slogan:

A system is composed of components;  
a component is something you understand **how to design**.

### “Co” for “collaborative”

In a second broad meaning, “co-” stands for “collaborative”:

co-design = design everything, together

There are two types of collaborations. First, there is the collaboration between human and machine, in the definition and solution of design problems. Second, and most importantly, is the collaboration among different experts or teams of experts in the design process.

The typical situation is that the system design is suboptimal because every expert only knows one component and there are rigid interfaces/contracts designed early on. The problem here is sharing of knowledge across teams, specifically, knowledge about the design of systems.

In this case, this is the slogan:

«A system is composed of components;  
a component is something that **somebody** understands **how to design**.»

There is a tight link between the “composition” and “collaboration” aspects.

As Conway<sup>†</sup> first observed for software systems:

«Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations.»

<sup>†</sup> John Horton Conway (1937–2020) was a mathematician. Probably the most popular idea of his was the invention of the Game of Life, which inspired countless works on cellular automata. We remember him for the discovery of the *surreal numbers*, which should be just called *numbers*, as they contain all other ordered fields.



This “mirroring” hypothesis between system and organization was explored formally and found to hold [19]. The ultimate reason is that “the organization’s governance structures, problem-solving routines and communication patterns constrain the space in which it searches for new solutions”. This appears to be true for generic systems in addition to software.

In the end, civilization is about dividing up the work, and so we must choose where one’s work ends and the other’s work begins. But we need to keep talking if we want that everything works together.

### “Co” for “computational”

The third meaning of “co-” in “co-design” will be **computational**. It is the age of machines, and we need machines to understand what we are doing.

Therefore, we strive to create not only a qualitative modeling for co-design, but also a formal and quantitative description that will be suitable for setting up an optimization problem that can be solved to obtain an optimal design.

Our slogan becomes:

«A system is composed of components;  
a component is something that **somebody** understands **how to design well enough to teach a computer**. »

### “Co” for “continuous”

The fourth meaning of “co-” is **continuous**. We look at designs not as something that exists as a single decision in time, but rather as something that continuous to exist and evolve, independently on the designer.

## 23.3. Basic concepts of formal engineering design

Later, all these concepts will find a formal definition in the language of category theory.

**Functionality and functional requirements** You are an engineer in front of an empty whiteboard, ready to start designing the next product. The first question to ask is: What is the *purpose* of the product to be designed? The purpose of the product is expressed by the *functional requirements*, sometimes called *functional specifications*, *desired behavior*, *objectives*, or simply *function*.

Unfortunately, the word “function” conflicts with the mathematical concept. Therefore, we will talk about *functionality*. Moreover, we will never use the word “function”, and instead use *map* to denote the mathematical concept.

**Example 23.1.** These are a few examples of functional requirements:

- ▷ A car must be able to transport at least  $n \geq 4$  passengers.
- ▷ A battery must store at least 100 kJ of energy.
- ▷ An autonomous vehicle should reach at least 20 mph while guaranteeing safety.

**Resources and resource constraints** We call *resources* what we need to pay to realize the given functionality. In some contexts, these are better called *costs*, or *dependencies*.

**Example 23.2.** These are a few examples of resource constraints:

- ▷ A car should not cost more than 15,000 USD.
- ▷ A battery should not weigh more than 1 kg.
- ▷ A process should not take more than 10 s.

**Duality of functionality and resources** There is an interesting duality between functionality and resources. When designing systems, one is given functional requirements, as a *lower bound* on the functionality to provide, and one is given resource constraints, which are an *upper bound* on the resources to use.

As far as design objectives go, most can be understood as either *minimize resource usage* or *maximize functionality provided*.

This duality between functionality and resources will be at the center of our formalization.

**Non-functional requirements** Functionality and resources do not cover all the requirements— there is, for example, a large class of *non-functional requirements* [35] such as the extensibility and the maintainability of the system. Nevertheless, functionality and resources can express most of the requirements which can be quantitatively evaluated, at least prior to designing, assembling, and testing the entire system.

**Implementation space** The *implementation space* or *design space* is the set of all possible design choices that could be chosen; by *implementation*, or the word “design”, used as a noun, we mean one particular set of choices. The implementation space  $\mathbf{I}$  is the set over which we are optimizing; an implementation  $i \in \mathbf{I}$  is a particular point in that set (Fig. 1).

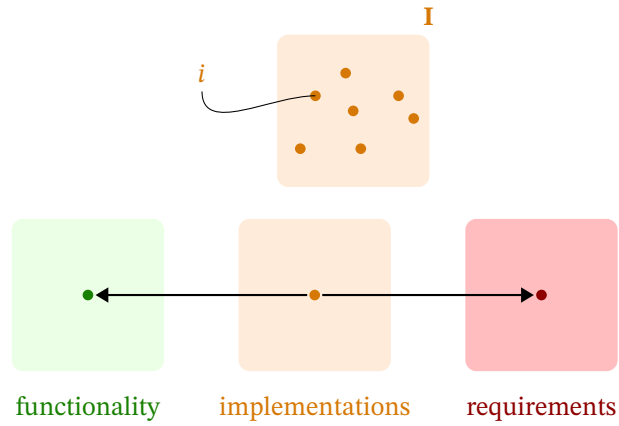
The interconnection between functionality, resources, and implementation spaces is as follows. We will assume that, given one implementation, we can evaluate it to know the functionality and the resources spaces (Fig. 2).

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

Watch: An abstract view of design problems (13 minutes).



**Figure 1.:** An *implementation*  $i$  is a particular point in the implementation space  $I$ .



**Figure 2.:** Evaluation of specific implementations to get functionality and resources spaces.

**Functional Interfaces and interconnection** Components are *interconnected* to create a system. This implies that we have defined the *interfaces* of components, which have the dual function of delimiting when one component ends and another begins, and also to describe exactly what is the nature of their interaction. We will develop a formalism in which the functionality and resources are the interfaces used for interconnection: two components are connected if the resources required by the first correspond to the functionality provided by the second.

**Abstraction** By *abstraction*, we mean that it is possible to “zoom out”, in the sense that a system of components can be seen as a component itself, which can be part of larger systems.

**Compositionality** A *compositional* property is a property that is preserved by interconnection and abstraction; assuming each component in a system satisfies that property, also the system as a whole satisfies the property.

**Example 23.3.** One can compose two electronic circuits by joining their terminals to obtain another electronic circuit. We would say that the property of being an electronic circuit is compositional.

## 23.4. Queries in design

Suppose that we have a model with a functionality space  $\mathbf{F}$ , a requirements space  $\mathbf{R}$ , and an implementation space  $\mathbf{I}$ .

There are several queries we can ask of a model. They all look at the same phenomenon from different angles, so they look similar; however the computational cost of answering each one might be very different.

The first kind of query is one that asks if the design is feasible when fixed all variables.

**Problem** (Feasibility problem). Given a triplet of implementation  $i \in \mathbf{I}$ , functionality  $f \in \mathbf{F}$ , requirements  $r \in \mathbf{R}$ , determine if the design is feasible.

The second type of query is that which fixes the boundary conditions of functionality and requirements, and asks to find a solution.

**Problem** (Find implementation). Given a pair of minimal requested functionality  $f \in \mathbf{F}$  and maximum allowed requirements  $r \in \mathbf{R}$ , determine if there is an implementation  $i \in \mathbf{I}$  that is feasible.

A different type of query is the one in which the design objective (the functionality) is fixed, and we ask what are the least resources necessary.

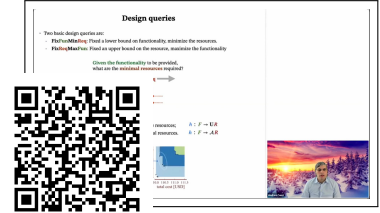
**Problem** (FixFunMinRes). Given a certain functionality  $f \in \mathbf{F}$ , find the set of “minimal” resources in  $\mathbf{R}$  that are needed to realize it (along with the implementations), or provide a proof that there are none (a certificate of infeasibility).

Dually, we can ask, fixed the resources available, what are the functionalities that can be provided.

**Problem** (FixResMaxFun). Given a certain requirement  $r \in \mathbf{R}$ , find the set of “maximal” functionalities in that can realize it (along with the implementations), or provide a proof that there are none (a certificate of infeasibility).

It is very natural to talk about the “minimal” requirements and “maximal” functionalities; after all, we always want to minimize costs and maximize performance. In the next chapter we start to put more mathematical scaffolding in place, starting from defining functionality and requirements as posets.

Watch: *Design queries* (5 minutes).







## 24. Poset bounds

This chapter describes some concepts related to posets: upper and lower sets, minimal/maximal elements, *etc.*

<b>24.1 Minimal and maximal elements</b>	<b>346</b>
<b>24.2 Code exercises - Minimal and maximal elements</b>	<b>347</b>
<b>24.3 Upper/lower bounds</b>	<b>348</b>
<b>24.4 Code exercises - Upper/lower bounds</b>	<b>350</b>
<b>24.5 Upper and lower sets</b>	<b>351</b>
<b>24.6 Code exercises - Upper and lower sets</b>	<b>352</b>
<b>24.7 Antichains</b>	<b>353</b>
<b>24.8 Code exercises - Upper and lower closure</b>	<b>356</b>

## 24.1. Minimal and maximal elements

You know already the operators min/max that give the minimum/maximum values of a set of real numbers. If the set is finite, the minimum and maximum always exist. But for infinite sets, the minimum and maximum might not exist. For example, consider the set of real numbers contained between 0 and 1, excluding the boundaries:

$$\mathbf{A} = \{x \in \mathbb{R} : 0 < x < 1\}. \quad (1)$$

This set does not have a minimum or a maximum.

For a total order, if the minimum and maximum exist, then they are unique. In a partial order, this is not the case. We introduce the operators Min and Max that are the generalization to partial orders of min / max.

### Definition 24.1 (Minimal elements)

$\text{Min} : \text{Pow } \mathbf{P} \rightarrow \text{Anti } \mathbf{P}$  is the map that sends a subset  $\mathbf{S}$  of a poset to the minimal elements of that subset (those elements  $a \in \mathbf{S}$  such that  $a \leq_{\mathbf{P}} b$  for all  $b \in \mathbf{S}$ ). In formulas:

$$\begin{aligned} \text{Min} : \text{Pow } \mathbf{P} &\rightarrow \text{Anti } \mathbf{P}, \\ \mathbf{S} &\mapsto \left\{ c \in \mathbf{S} : \frac{d \in \mathbf{S} \quad d \leq_{\mathbf{P}} c}{c = d} \right\}. \end{aligned} \quad (2)$$

Note that  $\text{Min}(\mathbf{S})$  could be empty.

### Definition 24.2 (Maximal elements)

$\text{Max} : \text{Pow } \mathbf{P} \rightarrow \text{Anti } \mathbf{P}$  is the map that sends a subset  $\mathbf{S}$  of a poset to the maximal elements of that subset (those elements  $a \in \mathbf{S}$  such that  $a \geq_{\mathbf{P}} b$  for all  $b \in \mathbf{S}$ ). In formulas:

$$\begin{aligned} \text{Max} : \text{Pow } \mathbf{P} &\rightarrow \text{Anti } \mathbf{P}, \\ \mathbf{S} &\mapsto \left\{ c \in \mathbf{S} : \frac{d \in \mathbf{S} \quad d \geq_{\mathbf{P}} c}{c = d} \right\}. \end{aligned} \quad (3)$$

Note that  $\text{Max}(\mathbf{S})$  could be empty.

## 24.2. Minimal and maximal elements

### Code exercise G.1 (TestFinitePosetMinMax)

Implement code to compute minimal and maximal elements.

Implement the interface in Listing 86.

Test using:

```
act4e-check TestFinitePosetMinMax
```

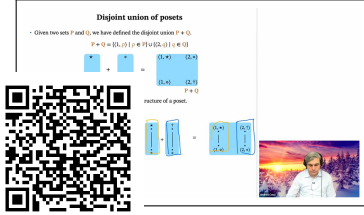
**Listing 86:** The `FinitePosetMinMax` interface.

```
class FinitePosetMinMax(ABC):
    @abstractmethod
    def minimal(self, fp: FinitePoset[E], S: List[E]) -> List[E]:
        """Return the minimal elements of S"""

    @abstractmethod
    def maximal(self, fp: FinitePoset[E], S: List[E]) -> List[E]:
        """Return the maximal elements of S"""
```



Watch: *Upper and lower bounds* (5 minutes).



## 24.3. Upper/lower bounds

### Definition 24.3 (Upper bounds in a poset)

The *upper bounds* of a subset  $S$  of a poset  $P$  are, if they exist, the elements of  $P$  which dominate all elements in  $S$ . In other words, the upper bounds of  $S$  are the elements of the set

$$\text{UppBS} := \{y \in P \mid \forall x \in S : x \leq_P y\}. \quad (4)$$

### Definition 24.4 (Least upper bound / join / supremum)

A *least upper bound* of  $S \subseteq P$ , if it exists, is the least element among the upper bounds of  $S$ . It is denoted  $\vee S$  or  $\text{Sup } S$ , and also called the *join* or *supremum* of  $S$ .

So, given  $S \subseteq P$  and  $y \in P$ ,  $y = \vee S$  if and only if

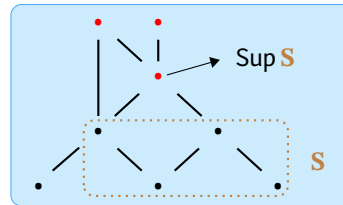
1.  $x \leq_P y$ ,  $\forall x \in S$ , and
2.  $x \leq_P z$ ,  $\forall x \in S \Rightarrow y \leq_P z$ .

If a least upper bound of a subset  $S \subseteq P$  exists, it is unique (can you prove this?), so we speak of “the” least upper bound.

**Exercise 40.** Let  $P$  be a poset and  $S \subseteq P$  a subset of the underlying set of  $P$ . Show that if  $\vee S$  exists, then it is unique. For this, assume that  $y$  and  $z$  are both least upper bounds of  $S$ , and then show that this assumption implies that in fact  $y = z$ .

*See solution on page 421.*

**Example 24.5.** Consider the poset  $P$  and its subset  $S$  depicted in Fig. 1. The red markers  $\bullet$  represent the upper bound of  $S$ . For this specific case, there is a *single* least upper bound.



**Figure 1.:** Example of upper bounds and least upper bound for  $S$ .

**Example 24.6.** Least upper bounds need not necessarily exist even in total orders. For instance, the subset

$$\mathbb{R}_{>0} = \{x \in \mathbb{R} : x > 0\} \quad (5)$$

of the poset  $\mathbb{R}$  (with the usual ordering) does not have a least upper bound.

Analogously to the case of (least) upper bounds, we can define lower bounds and greatest lower bounds.

### Definition 24.7 (Lower bounds in a poset)

The *lower bounds* of a subset  $S$  of a poset  $P$  are, if they exist, the elements which are dominated by all elements in  $S$ . In other words, the lower bounds of  $S$  are the elements of the set

$$\text{LowBS} := \{y \in P \mid \forall x \in S : y \leq_P x\}. \quad (6)$$

**Definition 24.8** (Greatest lower bound / meet / infimum)

The *greatest lower bound*, if it exists, is the greatest among the lower bounds of  $S$ . This is denoted  $\wedge S$  or  $\text{Inf } S$  and also called the *meet* or *infimum* of  $S$ .

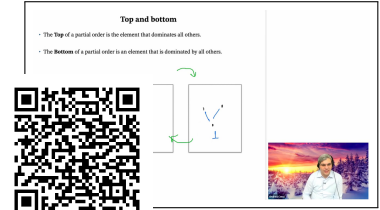
**Exercise 41.** Come up with an example of a subset  $S$  of a poset  $P$  which has lower bounds but no greatest lower bound. Then, modify it to have a greatest lower bound.

*See solution on page 421.*

**Definition 24.9** (Top and bottom)

If there is a least upper bound for the entire lattice  $P$ , it is called the *top* ( $\top$ ).  
If a greatest lower bound exists, it is called the *bottom* ( $\perp$ ).

Watch: *Top and bottom* (1 minutes).



## 24.4. ○ Upper/lower bounds

### Code exercise G.2 (TestFinitePosetInfSup)

Implement code to compute upper/lower bounds and infimum/supremum for a subset of a poset.

Implement the interface in Listing 87.

Test using:

```
act4e-check TestFinitePosetInfSup
```

Listing 87: The `FinitePosetInfSup` interface.

```
class FinitePosetInfSup(ABC):
    @abstractmethod
    def lower_bounds(self, fp: FinitePoset[E], s: List[E]) -> List[E]:
        """Computes the lower bounds for the subset (possibly empty)."""

    @abstractmethod
    def upper_bounds(self, fp: FinitePoset[E], s: List[E]) ->
        Optional[List[E]]:
        """Computes the upper bounds for the subset (possibly empty)."""

    @abstractmethod
    def infimum(self, fp: FinitePoset[E], s: List[E]) -> Optional[E]:
        """Computes the infimum / meet / greatest lower bound
        for the subset, or returns None if one does not exist."""

    @abstractmethod
    def supremum(self, fp: FinitePoset[E], s: List[E]) -> Optional[E]:
        """Computes the supremum for the subset if it exists,
        or returns None if one does not exist."""
```

## 24.5. Upper and lower sets

### Definition 24.10 (Upper set)

An *upper set*  $\mathbf{U}$  is a subset of a poset  $\mathbf{P}$  such that, if  $x \in \mathbf{U}$ , then all elements of  $\mathbf{P}$  that are above  $x$  are also in  $\mathbf{U}$ . In other words:

$$\frac{x \in \mathbf{U} \quad x \leq_{\mathbf{P}} y}{y \in \mathbf{U}}. \quad (7)$$

We call  $\mathbf{USets} \mathbf{P}$  the set of upper sets of  $\mathbf{P}$ .

### Definition 24.11 (Lower set)

A *lower set*  $\mathbf{L}$  is a subset of a poset  $\mathbf{P}$  such that, if  $x \in \mathbf{L}$ , then all elements of  $\mathbf{P}$  that are below  $x$  are also in  $\mathbf{L}$ . In other words:

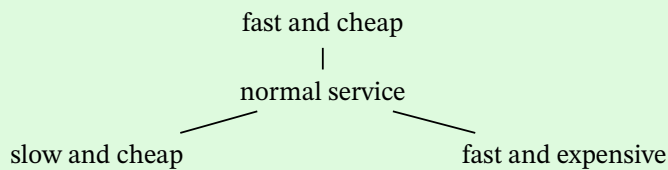
$$\frac{x \in \mathbf{L} \quad y \leq_{\mathbf{P}} x}{y \in \mathbf{L}}. \quad (8)$$

We call  $\mathbf{LSets} \mathbf{P}$  the set of lower sets of  $\mathbf{P}$ .

Given the battery choices  $\{\langle 10 \text{ CHF}, 500 \text{ g} \rangle, \langle 20 \text{ CHF}, 250 \text{ g} \rangle\}$ , we can represent an upper set as in Fig. 2a. The upper set can be interpreted as all the potential battery choices which are dominated by at least one of the two choices we have (in case we want to minimize mass and cost). Similarly, the lower set in Fig. 2b can be interpreted as all the potential battery choices which dominate at least one of the choices we have. Here when considering “the choices we have” in Fig. 2b, we not only consider the two choices directly presented to us, but also any convex combination of them.

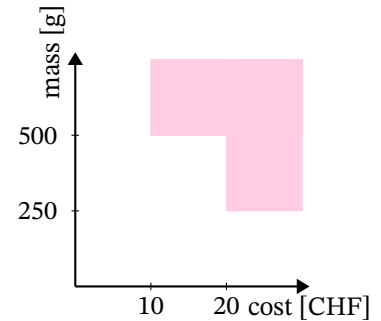
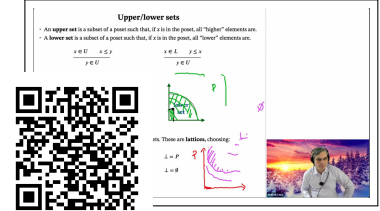
### Graded exercise G.1 (UpperSetsOfPreferences)

Consider the poset  $\mathbf{P}$  described by the following Hasse diagram:

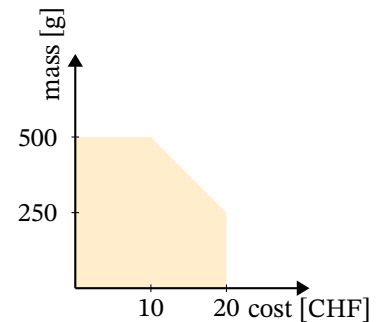


Your task in this exercise is to compute all upper sets of  $\mathbf{P}$ .

Watch: *Upper and lower sets* (4 minutes).



(a)



(b)

Figure 2.

## 24.6. Upper and lower sets

### Code exercise G.3 (TestFinitePosetSubsetProperties2)

Compute checks to ascertain that a subset is a lower or an upper set. Implement the interface in Listing 88.

Test using:

```
act4e-check TestFinitePosetSubsetProperties2
```

Listing 88: The `FinitePosetSubsetProperties2` interface.

```
class FinitePosetSubsetProperties2(ABC):
    @abstractmethod
    def is_lower_set(self, fp: FinitePoset[E], s: List[E]) -> bool: ...

    @abstractmethod
    def is_upper_set(self, fp: FinitePoset[E], s: List[E]) -> bool: ...
```

## 24.7. Upper and lower closure

### Definition 24.12 (Upper closure operator)

The *upper closure operator*  $\uparrow$  maps a subset to the smallest upper set that includes it:

$$\begin{aligned} \uparrow : \text{Pow } P &\rightarrow \text{USets } P, \\ P &\mapsto \{y \in P \mid \exists x \in P : x \leq_P y\}. \end{aligned} \quad (9)$$

**Remark 24.13.** Note that, by definition, an upper set is closed to upper closure.

**Lemma 24.14.** For any  $S \in \text{Pow } P$ ,  $\uparrow S$  is in fact an upper set.

*Proof.* Suppose  $y \in \uparrow S$  and  $z \in P$ , and suppose  $y \leq_P z$ . By definition there exists a  $x$  such that  $x \leq_P y$ , meaning that  $x \leq_P z$ . Thus,  $z \in \uparrow S$ , as was to be shown.  $\square$

**Lemma 24.15.** The upper closure operator  $\uparrow$  is an antitone map.

*Proof.* Consider the posets  $(\text{Pow } P, \subseteq)$  and  $(\text{USets } P, \supseteq)$ , and two sets of sets  $S_1, S_2 \in \text{Pow } P$ . It is clear that given  $S_1 \subseteq S_2$ , we have

$$\{y \in P \mid \exists x \in S_1 : x \leq_P y\} \subseteq \{y \in P \mid \exists x \in S_2 : x \leq_P y\}. \quad (10)$$

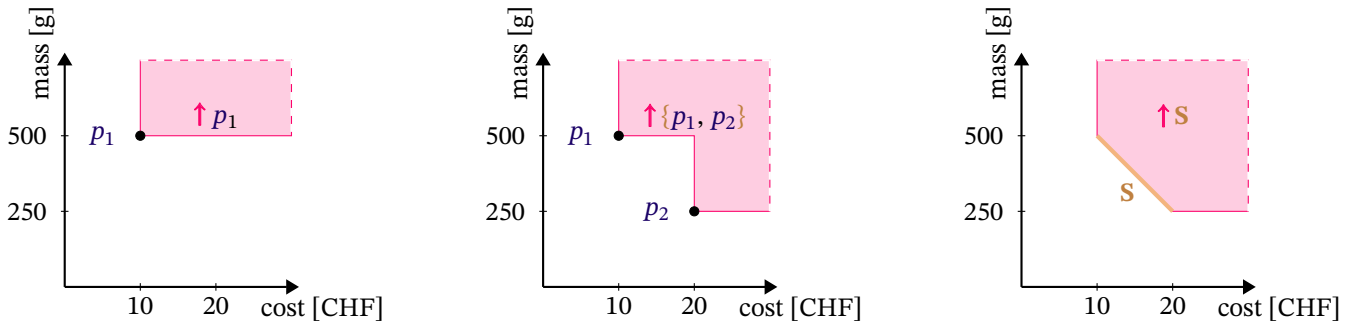
Therefore,  $\uparrow S_1 \subseteq \uparrow S_2$ , satisfying the antitone map property for  $\uparrow$ .  $\square$

In the example of battery choices (in the numerical case), first, consider the upper closure of a single element of the poset, for instance  $p_1 = \langle 10 \text{ CHF}, 500 \text{ g} \rangle$  (Fig. 3, left). Second, we can look at the upper closure when we add the choice  $p_2 = \langle 20 \text{ CHF}, 250 \text{ g} \rangle$  (Fig. 3, center).

Note that the upper set of the subset formed by the two elements is the union of the upper sets of the single elements. Finally, we can also define the set

$$S = \{\langle \text{cost}, \text{mass} \rangle \mid \text{mass} = 750 - 25 \cdot \text{cost}, \forall \text{cost} \in [0, 20]\}, \quad (11)$$

and find its upper closure (Fig. 3, right).

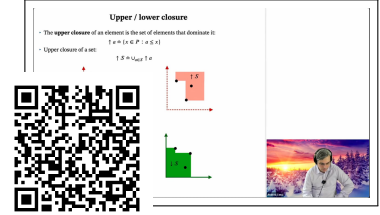


**Figure 3.:** Example of upper closure for different sets of battery choices.

### Definition 24.16 (Lower closure operator)

The *lower closure operator*  $\downarrow$  maps a subset to the smallest lower set that

Watch: *Upper and lower closure* (2 minutes).



includes it:

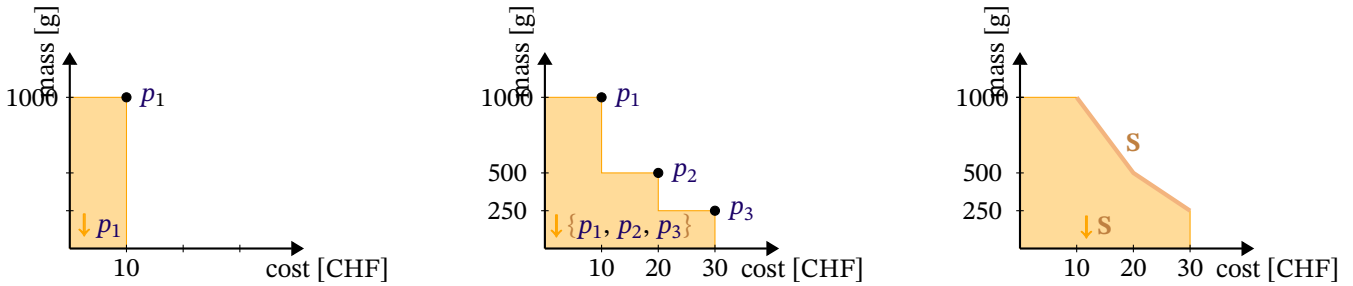
$$\begin{aligned} \downarrow : \text{Pow } \mathbf{P} &\rightarrow \text{LSets } \mathbf{P}, \\ \mathbf{S} &\mapsto \{y \in \mathbf{P} \mid \exists x \in \mathbf{S} : y \leq_{\mathbf{P}} x\}. \end{aligned} \quad (12)$$

**Lemma 24.17.** The lower closure operator  $\downarrow$  is a monotone map.

**Exercise 42.** Prove Lemma 24.17.

*See solution on page 421.*

Consider the battery example, and the antichain given by the battery models  $p_1 = \langle 10 \text{ CHF}, 1000 \text{ g} \rangle$ ,  $p_2 = \langle 20 \text{ CHF}, 500 \text{ g} \rangle$ , and  $p_3 = \langle 30 \text{ CHF}, 250 \text{ g} \rangle$  (Fig. 4, left). The lower closure operator  $\downarrow \{p_1, p_2, p_3\}$  represents all the battery models which, if existing, would dominate  $\{p_1, p_2, p_3\}$ . We could instead consider linear maps between the points getting a poset  $\mathbf{P}$ , and obtain the lower closure depicted in Fig. 4 on the right.



**Figure 4.:** Example of lower closures.

## Antichains and upper sets

**Lemma 24.18.** Let  $\mathbf{A}$  and  $\mathbf{B}$  be subsets of  $\mathbf{P}$  that are antichains. Then

$$\frac{\uparrow \mathbf{A} = \uparrow \mathbf{B}}{\mathbf{A} = \mathbf{B}}. \quad (13)$$

*Proof.* Fix an element  $a \in \mathbf{A}$ . From  $\uparrow \mathbf{A} = \uparrow \mathbf{B}$  we know that in particular  $\mathbf{A} \subseteq \uparrow \mathbf{B}$ . This means that for our fixed  $a \in \mathbf{A}$  there exists  $b \in \mathbf{B}$  such that  $b \leq a$ . From  $\uparrow \mathbf{A} = \uparrow \mathbf{B}$  it also follows that  $\mathbf{B} \subseteq \uparrow \mathbf{A}$ , so to the  $b \in \mathbf{B}$  given above, there exists  $a' \in \mathbf{A}$  such that  $a' \leq b$ . In total, we have  $a' \leq b \leq a$ , and since  $\mathbf{A}$  is an antichain, we must have  $a' = a$ . This implies that  $a' = b = a$ . In particular, we have  $a \in \mathbf{B}$ .

The above shows that  $\mathbf{A} \subseteq \mathbf{B}$ . To show  $\mathbf{B} \subseteq \mathbf{A}$ , we can fix any  $b \in \mathbf{B}$  and repeat the above argumentation, now with the roles of  $\mathbf{A}$  and  $\mathbf{B}$  exchanged.  $\square$

### Definition 24.19 (Downward closed set)

An upper set  $\mathbf{S}$  is *downward-closed* in a poset  $\mathbf{P}$  if

$$\mathbf{S} = \uparrow \text{Min } \mathbf{S}. \quad (14)$$

The set of downward-closed upper sets of  $\mathbf{P}$  is denoted  $\overline{\text{UpSets } \mathbf{P}}$ .

### Definition 24.20 (Upward closed set)

A lower set  $S$  is *upward-closed* in a poset  $P$  if

$$S = \downarrow \text{Max } S. \quad (15)$$

The set of upward-closed lower sets of  $P$  is denoted  $\overline{\text{LowSets } P}$ .



## 24.8. ○ Upper and lower closure

### Code exercise G.4 (TestFinitePosetClosures)

Write functions to compute the upper and lower closure of subsets.

Implement the interface in Listing 89.

Test using:

```
act4e-check TestFinitePosetClosures
```

**Listing 89:** The `FinitePosetClosures` interface.

```
class FinitePosetClosures(ABC):
    @abstractmethod
    def upper_closure(self, fp: FinitePoset[E], s: List[E]) -> List[E]: ...

    @abstractmethod
    def lower_closure(self, fp: FinitePoset[E], s: List[E]) -> List[E]: ...
```



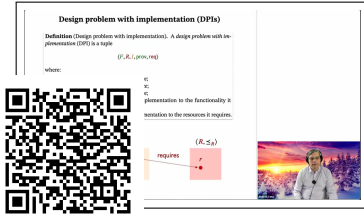
## 25. Monotone Co-Design Theory

This chapter introduces *Monotone Co-Design Theory*, a formalization for computational design theory. It is a compositional theory of which the primitive elements are *design problems* (DPI), formalized as relations among functionality, resources, and implementations.

We show that DPIs can capture design problems across diverse fields.

25.1 DPIs . . . . .	358
25.2 Examples . . . . .	361
25.3 Queries . . . . .	367
25.4 Co-design problems . . . . .	369
25.5 The semicategory DPI . . . . .	373
25.6 Sum and intersection of DPIs . . . . .	375

Watch: *Design problems with implementation (DPIs)* (7 minutes).



## 25.1. Design Problems with Implementation

We start by defining a “design problem with implementation”, which is a tuple of “**functionality** space”, “**implementation** space”, and “**resources** space”, together with two maps that describe the feasibility relations between these three spaces (Fig. 1).

**Definition 25.1** (Design problem with implementation)

A design problem with implementation (DPI) is a tuple

$$\langle \mathbf{F}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle, \quad (1)$$

where:

- ▷  $\mathbf{F}$  is a poset, called *functionality space*;
- ▷  $\mathbf{R}$  is a poset, called *requirements space*;
- ▷  $\mathbf{I}$  is a set, called *implementation space*;
- ▷ the map  $\text{prov} : \mathbf{I} \rightarrow \mathbf{F}$  maps an implementation to the functionality it provides;
- ▷ the map  $\text{req} : \mathbf{I} \rightarrow \mathbf{R}$  maps an implementation to the resources it requires.

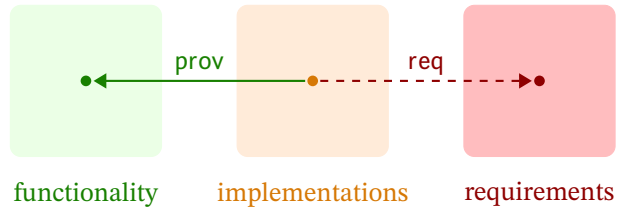


Figure 1.

**Example 25.2** (Motor design). Suppose we need to choose a motor for a robot from a given set. The **functionality** of a motor could be parametrized by **torque** and **speed**. The **resources** to consider could include the **cost [USD]**, the **mass [g]**, the input **voltage [V]**, and the input **current [A]**. The map  $\text{prov} : \mathbf{I} \rightarrow \mathbf{F}$  assigns to each motor its functionality, and the map  $\text{req} : \mathbf{I} \rightarrow \mathbf{R}$  assigns to each motor the resources it needs (Fig. 2).

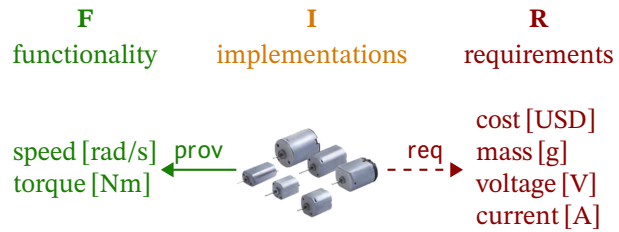


Figure 2.

**Example 25.3** (Chassis design). Suppose we need to choose a chassis for a robot. The implementation space  $\mathbf{I}$  (Fig. 3) could be the set of all chassis that could ever be designed (in case of a theoretical analysis), or just the set of chassis available in the catalogue at hand (in case of a practical design decision). The functionality of a chassis could be formalized as “the ability to transport a certain **payload [g]**” and “at a given **speed [m/s]**”. More refined functional requirements would include maneuverability, the cargo volume, *etc.* The resources to consider could be the **cost [USD]** of the chassis; the total mass; and, for each motor to be placed in the chassis, the required **speed [rad/s]** and **torque [Nm]**.



Figure 3.

**Example 25.4.** We revisit the leading example of Section 35.1 with the newly introduced co-design perspective. Consider a list of electrical motors as in Table 25.1.

**Table 25.1.:** A simplified catalogue of motors.

Motor ID	Company	Torque [kg · cm]	Weight [g]	Max Power [W]	Cost [USD]
1204	SOYO	0.18	60.0	2.34	19.95
1206	SOYO	0.95	140.0	3.00	19.95
1207	SOYO	0.65	130.0	2.07	12.95
2267	SOYO	3.7	285.0	4.76	16.95
2279	Sanyo Denki	1.9	165.0	5.40	164.95
1478	SOYO	19.0	1,000	8.96	49.95
2299	Sanyo Denki	2.2	150.0	5.90	59.95

We can think of this as a catalogue of electric motors  $\langle \mathbf{I}_{\text{EM}}, \text{prov}_{\text{EM}}, \text{req}_{\text{EM}} \rangle$ . In particular, the set of implementations collects all the motor models, which we can specify using the motor IDs:

$$\mathbf{I}_{\text{EM}} = \{1204, 1206, 1207, 2267, 2279, 1478, 2299\}. \quad (2)$$

We now have to think about **resources** and **functionalities**. Each motor **requires** some **weight** (in g), **power** (in W), and has some **cost** (in USD), and **provides** some **torque** (in kg · cm). Thus, we can identify

$$\mathbf{F} = \mathbb{R}^{\text{kg} \cdot \text{cm}}, \quad \mathbf{R} = \mathbb{R}^{\text{g}} \times \mathbb{R}^{\text{W}} \times \mathbb{R}^{\text{USD}}, \quad (3)$$

by considering the units as discussed in Section 15.3. The correspondences are given by the details in Table 25.1. For instance, we have

$$\text{prov}_{\text{EM}}(1204) = 0.18 \text{ kg} \cdot \text{cm}, \quad (4)$$

$$\text{req}_{\text{EM}}(1204) = \langle 60 \text{ g}, 2.34 \text{ W}, 19.95 \text{ USD} \rangle. \quad (5)$$

## Graphical notation

A graphical notation will help reasoning about composition. A DPI is represented as a box with  $n_f$  green edges and  $n_r$  red edges (Fig. 4).

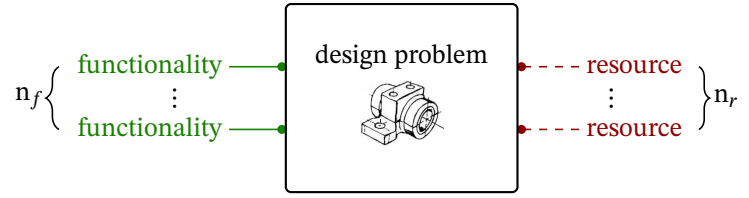


Figure 4.

This means that the functionality and resources spaces can be factorized in  $n_f$  and  $n_r$  components:

$$\mathbf{F} = \prod_{i=1}^{n_f} \text{pr}_i \mathbf{F}_i, \quad \mathbf{R} = \prod_{j=1}^{n_r} \text{pr}_j \mathbf{R}_j, \quad (6)$$

where “ $\text{pr}_i$ ” represents the projection to the  $i$ -th component. If there are no green (respectively, red) edges, then  $n_f$  (respectively,  $n_r$ ) is zero, and  $\mathbf{F}$  (respectively,  $\mathbf{R}$ ) is equal to  $\mathbf{1} = \{\langle \rangle\}$ , the set containing one element, the empty tuple  $\langle \rangle$ .

These *co-design diagrams* are not to be confused with signal flow diagrams, in which the boxes represent oriented systems and the edges represent signals.

## 25.2. Examples

We now present a list of design problems for different disciplines, to showcase the universality of the approach.

### Mechatronics

Many mechanisms can be readily modeled as relations between a provided functionality and required resources.

**Example 25.5.** A gearbox (Fig. 5) provides a certain **output torque**  $\tau_o$  and **speed**  $\omega_o$ , given a certain **input torque**  $\tau_i$  and **speed**  $\omega_i$ . For an ideal gearbox with a reduction ratio  $r \in \mathbb{Q}_+$  and efficiency ratio  $\gamma$ ,  $0 < \gamma < 1$ , the constraints among those quantities are  $\omega_i \geq r \omega_o$  and  $\tau_i \omega_i \geq \gamma \tau_o \omega_o$ . With this simple model, the set of implementations are given by the possible values of reduction and efficiency ratio.



Figure 5.

**Example 25.6.** *Propellers* (Fig. 6) generate **thrust** given a certain **torque** and **speed**.



Figure 6.

**Example 25.7.** A four-bar *crank-rocker* (Fig. 7) converts **rotational motion** into a **rocking motion**. The parametrization depends on the length of the four linkages.

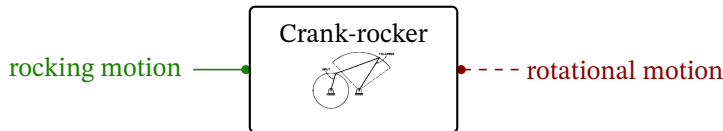


Figure 7.

### Geometrical constraints

Geometrical constraints are examples of constraints that are easily recognized as monotone, but possibly hard to write down in closed form.

**Example 25.8** (Bin packing). Suppose that each internal component occupies a volume bounded by a parallelepiped, and that we must choose the minimal enclosure in which to place all components (Fig. 8). What is the minimal size of the enclosure? This is a variation of the *bin packing* problem, which is in NP for both 2D and 3D [18]. It is easy to see that the problem is monotone, by noticing that, if one the components shapes increases, then the size of the enclosure cannot shrink. The implementations, in this case, are the configurations which one can choose to place all components in the container (one of the possible configurations is shown in the picture).



Figure 8.

### Inference

Many inference problems have a monotone formalization, taking the **accuracy** or **robustness** as functionality, and **computation** or **sensing** as resources. Typically, these bounds are known in a closed form only for restricted classes of systems, such as the linear/Gaussian setting.

**Example 25.9 (SLAM).** One issue with particle-filter-based estimation procedures, such as the ones used in the popular GMapping [10] suite, is that the filter might diverge if there aren't enough particles. Although the relation might be hard to characterize, there is a monotone relation between the **robustness** (1 - probability of failure), the **accuracy**, and the **number of particles** (Fig. 9). Here, the implementation space contains the other choices of parameters for the filter: fixed the number of particles, by changing the tuning of the filter, we can explore the trade-off of accuracy and robustness.

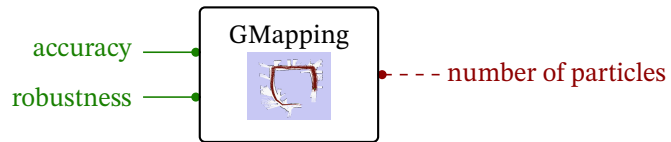


Figure 9.

**Example 25.10 (Stereo reconstruction).** Progressive reconstruction systems ([17]), which start with a coarse approximation of the solution that is progressively refined, are described by a smooth relation between the **resolution** and the **latency** to obtain the answer (Fig. 10). A similar relation characterizes any anytime algorithms in other domains, such as robot motion planning.

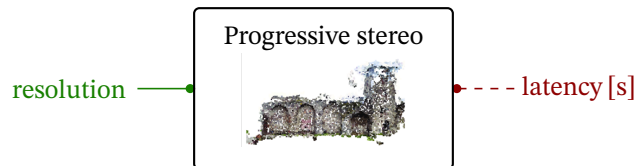


Figure 10.

**Example 25.11.** The empirical characterization of the monotone relation between the **accuracy of a visual SLAM solution** and the **power consumption** is the goal of recent work by Davison and colleagues [23, 36].

### Communication

**Example 25.12 (Transducers).** Any type of “transducer” that bridges between different mediums can be modeled as a DP. For example, an access point (Fig. 11) provides the **wireless access** functionality, and requires that the infrastructure provides the **Ethernet access** resource.

**Example 25.13 (Wireless link).** The basic functionality of a wireless link is to provide a certain **bandwidth** (Fig. 12). Further refinements could include bounds on the latency or the probability that a packet drop is dropped. Given the



Figure 11.

established convention about the preference relations for functionality, in which a *lower* functionality is “easier” to achieve, one needs to choose “*minus the latency*” and “*minus the packet drop probability*” for them to count as functionality. As for the resources, apart from the **transmission power [W]**, one should consider at least **the spectrum occupation**, which could be described as an interval  $[f_0, f_1]$  of the frequency axis  $\mathbb{R}^{[\text{Hz}]}$ . Thus, the resources space is  $\mathbf{R} = \mathbb{R}^{[\text{W}]} \times \text{intervals}(\mathbb{R}^{[\text{Hz}]})$ .

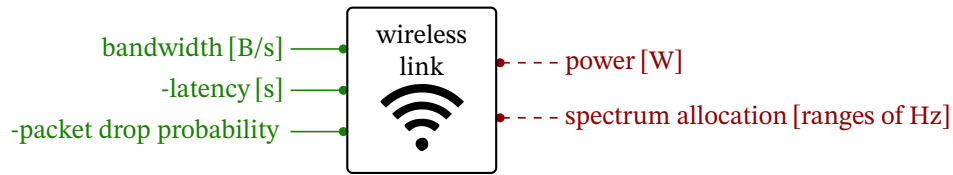


Figure 12.

## Multi-robot systems

In a multi-robot system there is always a trade-off between the number of robots and the capabilities of the single robot.

**Example 25.14.** Suppose we need to create a swarm of agents whose functionality is **to sweep an area**. If the functionality is fixed, one expects a three-way trade-off between the three resources: number of agents, the speed of a single agent, and the execution time. For example, if the time available decreases, we have to increase either the speed of an agent or the number of agents (Fig. 13b).

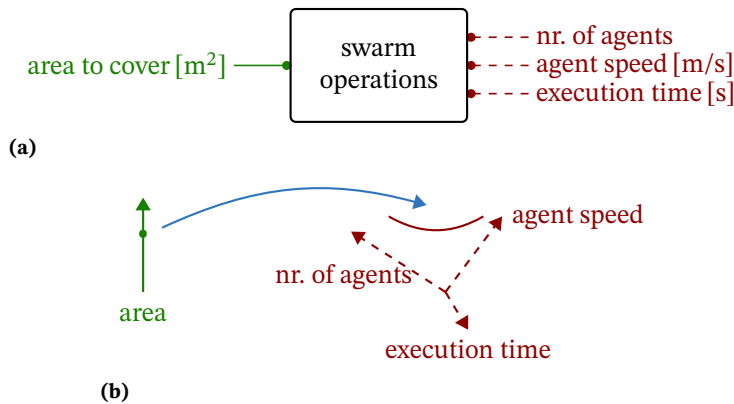


Figure 13.

## Computation graphs

The trivial model of a CPU is as a device that provides **computation, measured in flops**, and requires **power [W]**. Clearly there is a monotone relation between the two.



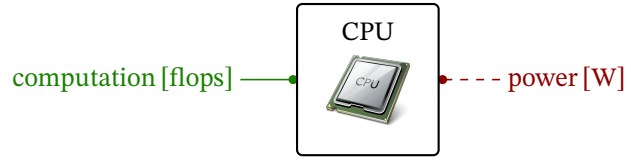


Figure 14.

A similar monotone relation between application requirements and computation resources holds in a much more general setting, where both application and computation resources are represented by graphs. This will be an example of a monotone relation between nontrivial partial orders.

In the Static Data Flow (SDF) model of computation [32, 15, Chapter 3], the application is represented as a graph of procedures that need to be allocated on a network of processors.

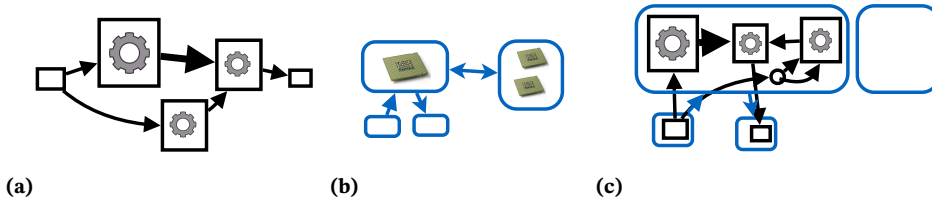


Figure 15.

Define the *application graph* (sometimes called “computation graph”) as a graph where each node is a procedure (or “actor”) and each edge is a message that needs to be passed between procedures. Each node is labeled by the number of ops necessary to run the procedure. Each edge is labeled by the size of the message. There is a partial order  $\leq$  on application graphs. In this order, it holds that  $A_1 \leq A_2$  if the application graph  $A_2$  needs more computation or bandwidth for its execution than  $A_1$ . Formally, it holds that  $A_1 \leq A_2$  if there is a homomorphism  $\varphi : A_1 \Rightarrow A_2$ ; and, for each node  $n \in A_1$ , the node  $\varphi(n)$  has equal or larger computational requirements than  $n$ ; and for each edge  $\langle n_1, n_2 \rangle$  in  $A_2$ , the edge  $\langle \varphi(n_1), \varphi(n_2) \rangle$  has equal or larger message size.

Define a *resource graph* as a graph where each node represents a processor, and each edge represents a network link. Each node is labeled by the processor capacity [flops]. Each edge is labeled by latency [s] and bandwidth [B/s]. There is a partial order on resources graph as well: it holds that  $R_1 \leq R_2$  if the resource graph  $R_2$  has more computation or network available than  $R_1$ . The definition is similar to the case of the application graph: there must exist a graph homomorphism  $\varphi : R_1 \Rightarrow R_2$  and the corresponding nodes (edges) of  $R_2$  must have larger or equal computation (bandwidth) than those of  $R_1$ .

Given an application graph  $A$  and a resource graph  $R$ , a typical resource allocation problem consists in choosing in which processor each actor must be scheduled to maximize the throughput  $T$  [Hz]. This is equivalent to the problem of finding a graph homomorphism  $\Psi : A \Rightarrow R$ . Let  $T^*$  be the optimal throughput, and write it as a function of the two graphs:

$$T^* = T^*(A, R). \quad (7)$$

Then the optimal throughput  $T^*$  is decreasing in  $A$  (a more computationally demanding application graph decreases the throughput) and increasing in  $R$  (more available computation/bandwidth increase the throughput).

Therefore, we can formalize this as a design problem where the two functionalities are the throughput  $T$  [Hz] and the application graph  $A$ , and the resource graph  $R$

is the resource.

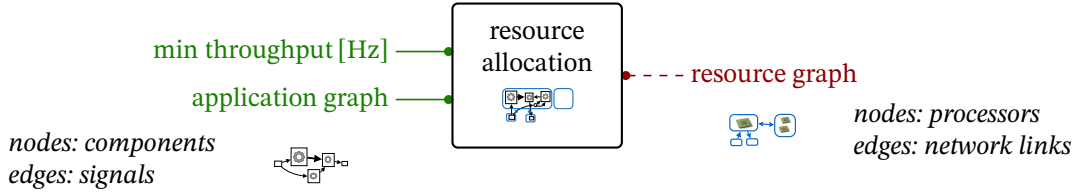


Figure 16.

**Example 25.15.** Svorenova *et al.* [33] consider a joint sensor scheduling and control synthesis problem, in which a robot can decide to not perform sensing to save power, given performance objectives on the probability of reaching the target and the probability of collision. The method outputs a Pareto frontier of all possible operating points. This can be cast as a design problem with functionality equal to the **probability of reaching the target** and (the inverse of) **the collision probability**, and with resources equal to the **actuation power**, **sensing power**, and **sensor accuracy**.

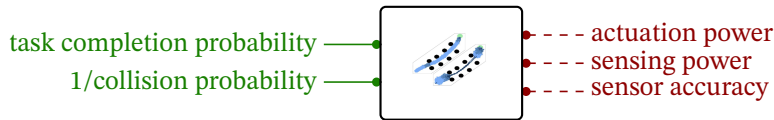


Figure 17.

**Example 25.16.** Nardi *et al.* [36] describe a benchmarking system for visual SLAM that provides the empirical characterization of the monotone relation between **the accuracy** of the visual SLAM solution, the **throughput [frames/s]** and **the energy for computation [J/frame]**. The implementation space is the product of algorithmic parameters, compiler flags, and architecture choices, such as the number of GPU cores active. This is an example of a design problem whose functionality-resources map needs to be experimentally evaluated.



Figure 18.

### Other examples in minimal robotics

Many works have sought to find “minimal” designs for robots, and can be understood as characterizing the relation between the poset of **tasks** and the poset of physical resources, which is the product of **sensing**, **actuation**, and **computation** resources, plus other non-physical resources, such as **prior knowledge** (Fig. 19). Given a task, there is a minimal antichain in the resources poset that describes the possible trade-offs (for instance, compensating lousier sensors with more computation).

The poset structure arises naturally: for example, in the *sensor lattice* [14], a sensor dominates another if it induces a finer partition of the state space. Similar dominance relations can be defined for actuation and computation. O’Kane and Lavelle [24] define a robot as a union of “robotic primitives”, where each primitive is an abstraction for a set of sensors, actuators, and control strategies that can be used together (for instance, a compass plus a contact sensor allow to “drive North until a wall is hit”). The effect of each primitive is modeled as an operator on the

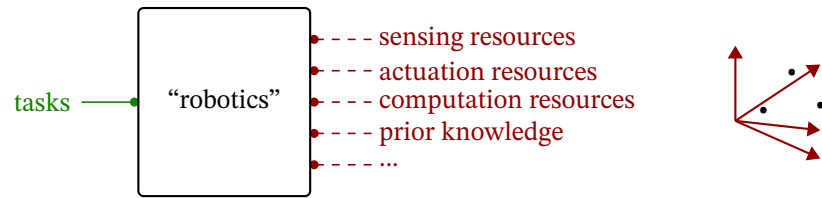


Figure 19.

robot's information space. It is possible to work out what are the minimal combinations of robotic primitives (minimal antichain) that are sufficient to perform a task (for instance, global localization), and describe a dominance relation (partial order) of primitives. Other works have focused on minimizing the complexity of the controller. Egerstedt [5] studies the relation between the **complexity of the environment** and a notion of **minimum description length of control strategies**, which can be taken as a proxy for the computation necessary to perform the task. Soatto [29] studies the relation between the **performance of a visual task**, and the **minimal representation** that is needed to perform that task.

## 25.3. Queries

A DPI is a model to which we can associate a family of optimization problems. While in previous examples we covered the problem “feasibility”, we still miss **FixFunMinRes**, **FixResMaxFun**, and **FeasibleImp**.

The first can be translated to “Given a lower bound on the functionality  $f$ , what are the implementations that have minimal resource usage?” (Fig. 20).

**Problem (FixFunMinRes).** Given  $f \in \mathbf{F}$ , find the implementations in  $\mathbf{I}$  that realize the functionality  $f$  (or higher) with minimal resources, or provide a proof that there are none:

$$\left\{ \begin{array}{ll} \text{using} & i \in \mathbf{I}, \\ \text{Min}_{\leq_{\mathbf{R}}} & r, \\ \text{s.t.} & r = \text{req}(i), \\ & f \leq_{\mathbf{F}} \text{prov}(i). \end{array} \right. \quad (8)$$

**Remark 25.17** (Minimal vs least solutions). Note the use of  $\text{Min}_{\leq_{\mathbf{R}}}$  in (8), which indicates the set of minimal (non-dominated) elements according to  $\leq_{\mathbf{R}}$ , rather than  $\min_{\leq_{\mathbf{R}}}$ , which would presume the existence of the least element. In all problems in this paper, the goal is to find the optimal trade-off of resources (“Pareto front”). So, for each  $f$ , we expect to find an antichain  $R \in \text{Anti } \mathbf{R}$ . We will see that this formalization allows an elegant way to treat multi-objective optimization problems. The algorithm to be developed will directly solve for the set  $R$ , without resorting to techniques such as *scalarization*, and therefore is able to work with arbitrary posets, possibly discrete.

In an entirely symmetric fashion, we could fix an upper bound on the resource usage, and then maximize the functionality provided (Fig. 21). The formulation is entirely dual, in the sense that it is obtained from (8) by swapping Min with Max,  $\mathbf{F}$  with  $\mathbf{R}$ , and  $\text{prov}$  with  $\text{req}$ .

**Problem (FixResMaxFun).** Given  $r \in \mathbf{R}$ , find the implementations in  $\mathbf{I}$  that requires  $r$  (or lower) and provide the maximal functionality, or provide a proof that there are none:

$$\left\{ \begin{array}{ll} \text{using} & i \in \mathbf{I}, \\ \text{Max}_{\leq_{\mathbf{F}}} & f, \\ \text{s.t.} & f = \text{prov}(i), \\ & r \geq_{\mathbf{R}} \text{req}(i). \end{array} \right. \quad (9)$$

Another type of query is: “Given a lower bound on the functionality  $f$  and an upper bound on the costs  $f$ , what are the feasible implementations?”

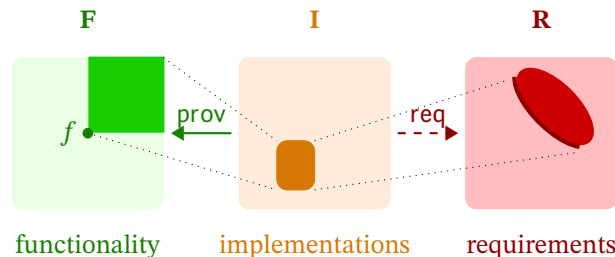
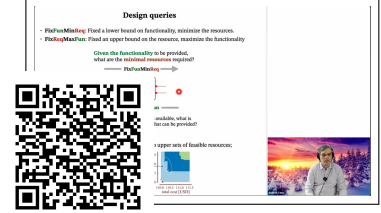


Figure 20.

Watch: *Design queries* (3 minutes).



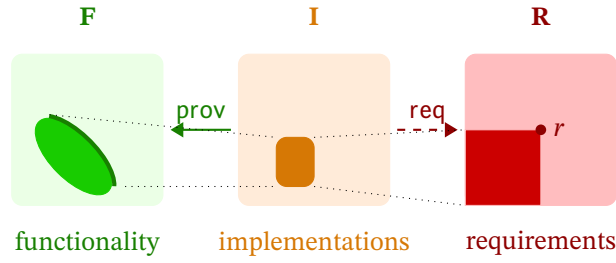


Figure 21.

**Problem** (FeasibleImp). Given  $f \in \mathbf{F}$  and  $r \in \mathbf{R}$ , find the implementations in  $\mathbf{I}$  that requires  $r$  (or lower) and provide  $f$  (or higher)

$$\begin{cases} \text{using } i \in \mathbf{I}, \\ \text{s.t. } f \leq_{\mathbf{F}} \text{prov}(i), \\ \text{s.t. } \text{prov}(i) \leq_{\mathbf{R}} \text{req}(i), \end{cases} \quad (10)$$

Another variation is to find only whether there are feasible solutions or not.

**Problem** (Feasibility). Given  $f \in \mathbf{F}$  and  $r \in \mathbf{R}$ , find if (10) is feasible.

## 25.4. Co-design problems

A “co-design problem” will be defined as a *multigraph* of design problems.

**Definition 25.18** (Co-design problem with implementation)

A *co-design problem with implementation* (CDPI) is a tuple  $\langle \mathbf{F}, \mathbf{R}, \langle \mathcal{V}, \mathcal{E} \rangle \rangle$ , where  $\mathbf{F}$  and  $\mathbf{R}$  are two posets, and  $\langle \mathcal{V}, \mathcal{E} \rangle$  is a multigraph of DPIs. Each node  $\mathbf{d} \in \mathcal{V}$  is a DPI  $\mathbf{d} = \langle \mathbf{F}_{\mathbf{d}}, \mathbf{R}_{\mathbf{d}}, \mathbf{I}_{\mathbf{d}}, \text{prov}_{\mathbf{d}}, \text{req}_{\mathbf{d}} \rangle$ . An edge  $e \in \mathcal{E}$  is a tuple  $e = \langle \langle \mathbf{d}_1, i_1 \rangle, \langle \mathbf{d}_2, j_2 \rangle \rangle$ , where  $\mathbf{d}_1, \mathbf{d}_2 \in \mathcal{V}$  are two nodes and  $i_1$  and  $j_2$  are the indices of the components of the functionality and resources to be connected, and it holds that  $\pi_{i_1} \mathbf{R}_{\mathbf{d}_1} = \pi_{j_2} \mathbf{F}_{\mathbf{d}_2}$  (Fig. 22).

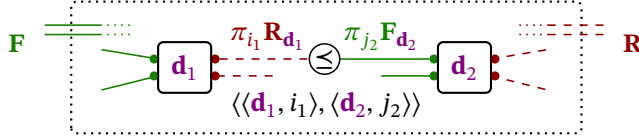


Figure 22.

A CDPI is equivalent to a DPI with an implementation space  $\mathbf{I}$  that is a subset of the product  $\prod_{\mathbf{d} \in \mathcal{V}} \mathbf{I}_{\mathbf{d}}$ , and contains only the tuples that satisfy the co-design constraints. An implementation tuple  $i \in \prod_{\mathbf{d} \in \mathcal{V}} \mathbf{I}_{\mathbf{d}}$  belongs to  $\mathbf{I}$  iff it respects all functionality–resources constraints on the edges, in the sense that, for all edges  $\langle \langle \mathbf{d}_1, i_1 \rangle, \langle \mathbf{d}_2, j_2 \rangle \rangle$  in  $\mathcal{E}$ , it holds that

$$\pi_{i_1} \text{req}_{\mathbf{d}_1}(\pi_{\mathbf{d}_1} i) \leq \pi_{j_2} \text{prov}_{\mathbf{d}_2}(\pi_{\mathbf{d}_2} i). \quad (11)$$

The posets  $\mathbf{F}, \mathbf{R}$  for the entire CDPI are the products of the functionality and resources of the nodes that remain *unconnected*. For a node  $\mathbf{d}$ , let  $\text{UF}_{\mathbf{d}}$  and  $\text{UR}_{\mathbf{d}}$  be the set of unconnected functionalities and resources. Then  $\mathbf{F}$  and  $\mathbf{R}$  for the CDPI are defined as the product of the unconnected functionality and resources of all DPIs:  $\mathbf{F} = \prod_{\mathbf{d} \in \mathcal{V}} \prod_{j \in \text{UF}_{\mathbf{d}}} \pi_j \mathbf{F}_{\mathbf{d}}$  and  $\mathbf{R} = \prod_{\mathbf{d} \in \mathcal{V}} \prod_{i \in \text{UR}_{\mathbf{d}}} \pi_i \mathbf{R}_{\mathbf{d}}$ . The maps  $\text{prov}$  and  $\text{req}$  return the values of the unconnected functionality and resources:

$$\begin{aligned} \text{prov} : i &\mapsto \prod_{\mathbf{d} \in \mathcal{V}} \prod_{j \in \text{UF}_{\mathbf{d}}} \pi_j \text{prov}_{\mathbf{d}}(\pi_{\mathbf{d}} i), \\ \text{req} : i &\mapsto \prod_{\mathbf{d} \in \mathcal{V}} \prod_{i \in \text{UR}_{\mathbf{d}}} \pi_i \text{req}_{\mathbf{d}}(\pi_{\mathbf{d}} i). \end{aligned} \quad (12)$$

**Example 25.19.** The CDPI in Fig. 23 is the interconnection of 3 DPs  $\mathbf{d}, \mathbf{e}, \mathbf{g}$ . The implementation space is a subset of the product

$$\mathbf{I}_{\mathbf{d}} \times \mathbf{I}_{\mathbf{e}} \times \mathbf{I}_{\mathbf{g}}. \quad (13)$$

The elements  $\langle i_{\mathbf{d}}, i_{\mathbf{e}}, i_{\mathbf{g}} \rangle$  that are feasible are the ones that respect the following constraints:

1. Functionality and resources of each DPI are given by their implementation:

$$r_{\mathbf{d}} = \text{req}(i_{\mathbf{d}}), \quad (14)$$

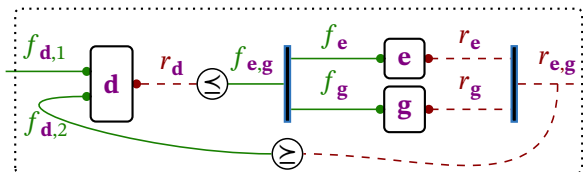


Figure 23.: Example of interconnection of 3 DPs

$$r_e = \text{req}(i_e), \quad (15)$$

$$r_g = \text{req}(i_g), \quad (16)$$

$$f_d = \text{prov}(i_d), \quad (17)$$

$$f_e = \text{prov}(i_e), \quad (18)$$

$$f_g = \text{prov}(i_g). \quad (19)$$

2. Wiring constraints:

$$\langle f_{d_1}, f_{d_2} \rangle = f_d, \quad (20)$$

$$r_{e,g} = \langle r_e, r_g \rangle, \quad (21)$$

$$f_{e,g} = \langle f_e, f_g \rangle. \quad (22)$$

3. Co-design constraints:

$$r_{e,g} \leq f_{d_2}, \quad (23)$$

$$r_d \leq f_{e,g}. \quad (24)$$

## Recursive constraints

**Example 25.20.** Consider the co-design of chassis (Example 25.3) plus motor (Example 25.2). The design problem for a motor has **speed** and **torque** as the provided functionality (what the motor must provide), and **cost**, **mass**, **voltage**, and **current** as the required resources (Fig. 24).

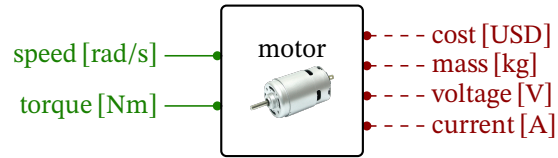


Figure 24.

For the chassis (Fig. 25), the provided functionality is parameterized by the **mass** of the payload and the **cost**, **total mass**, and what the chassis needs from its motor(s), such as **speed** and **torque**.

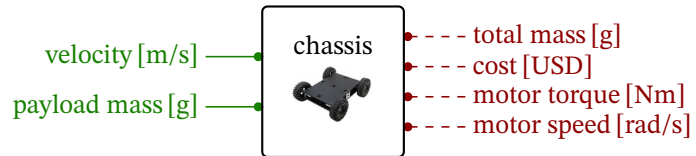


Figure 25.

The two design problem can be connected at the edges for torque and speed, as in Fig. 26. The semantics is that the motor needs to have *at least* the given torque and speed.

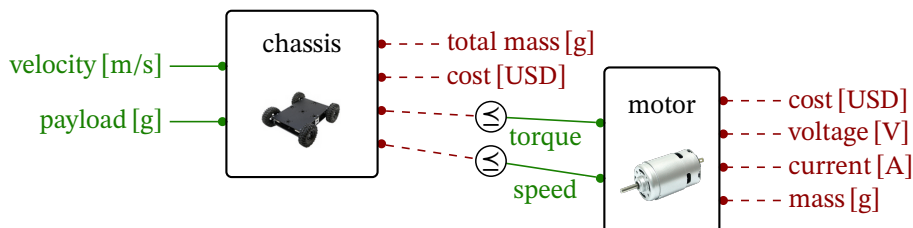


Figure 26.

Resources can be summed together using a trivial DP corresponding to the relation:

$$f_1 + f_2 \leq r. \quad (25)$$

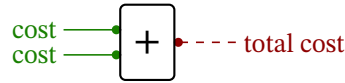


Figure 27.

A co-design problem might contain recursive co-design constraints. For example, if we set the payload to be transported to be the sum of the motor mass plus some extra payload, a cycle appears in the graph (Fig. 28).

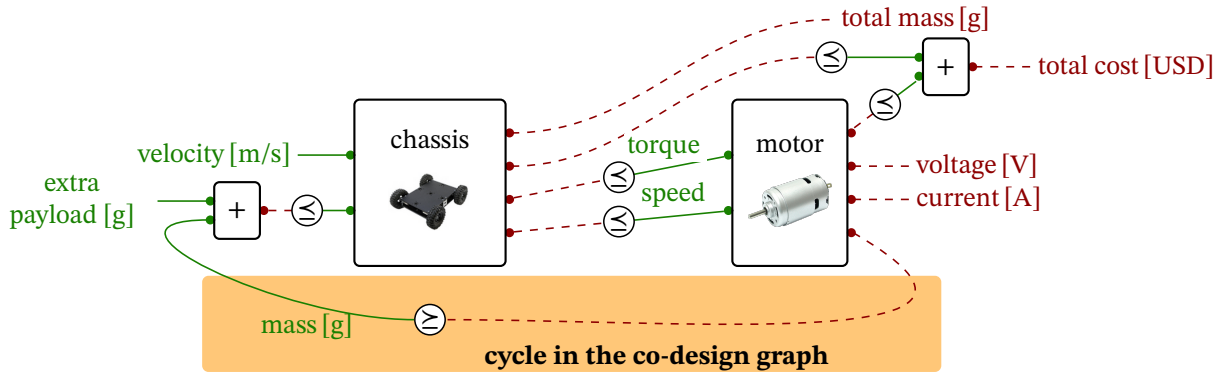


Figure 28.

## Abstraction

This formalism makes it easy to abstract away the details in which we are not interested. Once a diagram like Fig. 28 is obtained, we can draw a box around it and consider the abstracted problem (Fig. 29).

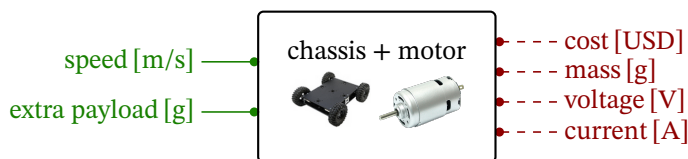


Figure 29.

Let us finish assembling our robot. A motor needs a motor control board. The functional requirements are the (peak) **output current** and the **output voltage range** (Fig. 30).

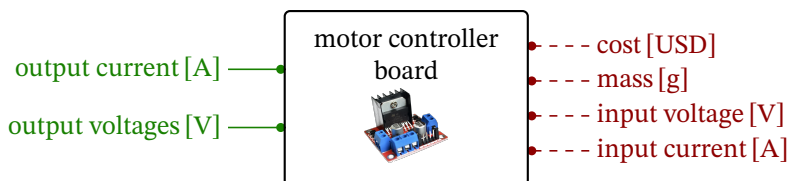


Figure 30.

The functionality for a power supply could be parameterized by the **output current**, the **output voltages**, and the **capacity**. The resources could include **cost** and **mass** (Fig. 31).

Relations such as **current** × **voltage** ≤ **power required** and **power** × **endurance** ≤ **energy required** can be modeled by a trivial “multiplication” DPI (Fig. 32).



Figure 31.

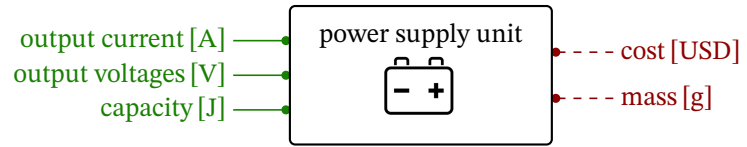


Figure 32.

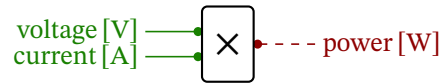
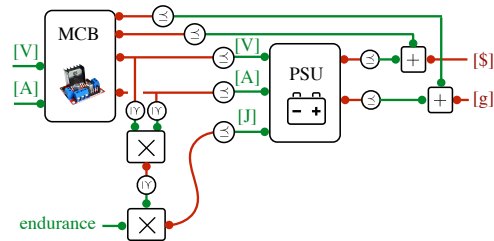


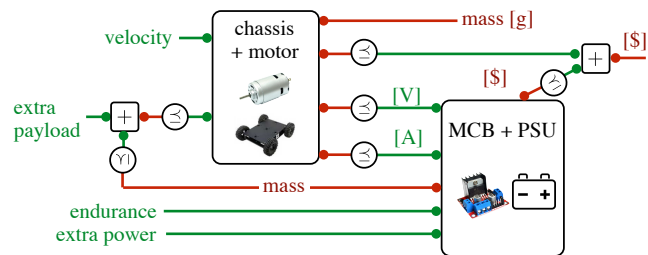
Figure 33.



We can connect these DPs to obtain a co-design problem with functionality **voltage**, **current**, **endurance**, and resources **mass** and **cost** (Fig. 33).

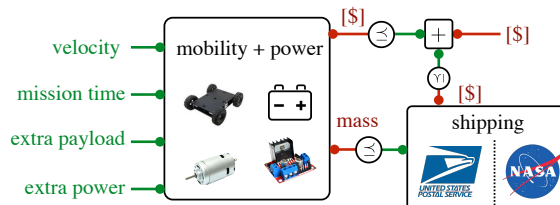
Draw a box around the diagram, and call it “MCB+PSU”; then interconnect it with the “chassis+motor” diagram in Fig. 34.

Figure 34.



We can further abstract away the diagram in Fig. 34 as a “mobility+power” CDPI, as in Fig. 35. The formalism allows considering **mass** and **cost** as independent resources, meaning that we wish to obtain the Pareto frontier for the minimal resources. Of course, we can always reduce everything to a scalar objective. For example, a conversion from mass to cost exists, and it is called “shipping”. Depending on the destination, the conversion factor is between \$0.5/lbs, using USPS, to \$10k/lbs for sending your robot to low Earth orbit.

Figure 35.



## 25.5. The semicategory **DPI**

### Definition 25.21 (Semicategory **DPI**)

There is a semicategory **DPI** where

- ▷ The objects are posets, objects of  $\langle \mathbf{Pos} \rangle$ .
- ▷ The morphisms from **F** to **R** are DPIs  $\langle \mathbf{F}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle$ , with **I** an object of  $\langle \mathbf{Set} \rangle$ .
- ▷ Given two morphisms  $\mathbf{d}_1 : \mathbf{A} \leftrightarrow \mathbf{B}$  and  $\mathbf{d}_2 : \mathbf{B} \leftrightarrow \mathbf{C}$ , described by

$$\mathbf{d}_1 = \langle \mathbf{A}, \mathbf{B}, \mathbf{I}_1, \text{prov}_1, \text{req}_1 \rangle, \quad (26)$$

$$\mathbf{d}_2 = \langle \mathbf{B}, \mathbf{C}, \mathbf{I}_2, \text{prov}_2, \text{req}_2 \rangle, \quad (27)$$

their composition  $\mathbf{d}_1 \circ \mathbf{d}_2$  is given by

$$\mathbf{d}_1 \circ \mathbf{d}_2 = \langle \mathbf{A}, \mathbf{C}, \mathbf{I}, \text{prov}, \text{req} \rangle, \quad (28)$$

where

$$\mathbf{I} = \{i_1 \circ i_2 \in \mathbf{I}_1 \circ \mathbf{I}_2 \mid \text{req}_1(i_1) \leq_{\mathbf{B}} \text{prov}_2(i_2)\}, \quad (29)$$

$$\text{prov} : i_1 \circ i_2 \mapsto \text{prov}_1(i_1), \quad (30)$$

$$\text{req} : i_1 \circ i_2 \mapsto \text{req}_2(i_2). \quad (31)$$

The semantics of the interconnection is that the second DPI provides the resources required by the first DPI. This is a partial order inequality constraint of the type  $r_1 \leq f_2$ .

**Lemma 25.22.** Series composition is associative.

*Proof.* Consider

$$\begin{aligned} \mathbf{d}_1 &= \langle \mathbf{A}, \mathbf{B}, \mathbf{I}_1, \text{prov}_1, \text{req}_1 \rangle, \\ \mathbf{d}_2 &= \langle \mathbf{B}, \mathbf{C}, \mathbf{I}_2, \text{prov}_2, \text{req}_2 \rangle, \\ \mathbf{d}_3 &= \langle \mathbf{C}, \mathbf{D}, \mathbf{I}_3, \text{prov}_3, \text{req}_3 \rangle. \end{aligned} \quad (32)$$

We want to show that

$$(\mathbf{d}_1 \circ \mathbf{d}_2) \circ \mathbf{d}_3 = \mathbf{d}_1 \circ (\mathbf{d}_2 \circ \mathbf{d}_3). \quad (33)$$

We know that the first part of the left term of (33) gives

$$(\mathbf{d}_1 \circ \mathbf{d}_2) = \langle \mathbf{A}, \mathbf{C}, \mathbf{I}_{1,2}, \text{prov}_{1,2}, \text{req}_{1,2} \rangle, \quad (34)$$

where

$$\mathbf{I}_{1,2} = \{i_1 \circ i_2 \in \mathbf{I}_1 \circ \mathbf{I}_2 \mid \text{req}_1(i_1) \leq_{\mathbf{B}} \text{prov}_2(i_2)\} \quad (35)$$

$$\text{prov}_{1,2} : i_1 \circ i_2 \mapsto \text{prov}_1(i_1) \quad (36)$$

$$\text{req}_{1,2} : i_1 \circ i_2 \mapsto \text{req}_2(i_2) \quad (37)$$

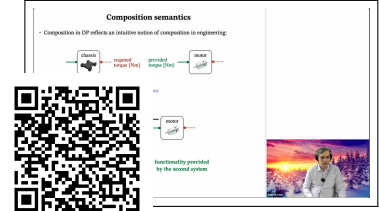
Therefore, the full left term of (33) reads

$$(\mathbf{d}_1 \circ \mathbf{d}_2) \circ \mathbf{d}_3 = \langle \mathbf{A}, \mathbf{D}, \mathbf{I}_{1,3}, \text{prov}_{1,3}, \text{req}_{1,3} \rangle, \quad (38)$$

where

$$\mathbf{I}_{1,3} = \{i_1 \circ i_2 \circ i_3 \in \mathbf{I} \mid \text{req}_{1,2}(i_1 \circ i_2) \leq_{\mathbf{C}} \text{prov}_3(i_3) \wedge \text{req}_1(i_1) \leq_{\mathbf{B}} \text{prov}_2(i_2)\}, \quad (39)$$

Watch: *Composing DPIs* (4 minutes).



$$= \{i_1 \circ i_2 \circ i_3 \in \mathbf{I} \mid \text{req}_2(i_2) \leq_{\mathbf{C}} \text{prov}_3(i_3) \wedge \text{req}_1(i_1) \leq_{\mathbf{B}} \text{prov}_2(i_2)\}, \quad (40)$$

$$\text{prov}_{1,3} : i_1 \circ i_2 \circ i_3 \mapsto \text{prov}_{1,2}(i_1 \circ i_2) = \text{prov}_1(i_1), \quad (41)$$

$$\text{req}_{1,3} : i_1 \circ i_2 \circ i_3 \mapsto \text{req}_3(i_3), \quad (42)$$

where  $\mathbf{I} = \mathbf{I}_1 \circ \mathbf{I}_2 \circ \mathbf{I}_3$ .

By expanding the second part of the second term of (33), we have:

$$(\mathbf{d}_2 \circ \mathbf{d}_3) = \langle \mathbf{B}, \mathbf{D}, \mathbf{I}_{2,3}, \text{prov}_{2,3}, \text{req}_{2,3} \rangle, \quad (43)$$

where

$$\mathbf{I}_{2,3} = \{i_2 \circ i_3 \in \mathbf{I}_2 \circ \mathbf{I}_3 \mid \text{req}_2(i_2) \leq_{\mathbf{C}} \text{prov}_3(i_3)\} \quad (44)$$

$$\text{prov}_{2,3} : i_2 \circ i_3 \mapsto \text{prov}_2(i_2) \quad (45)$$

$$\text{req}_{2,3} : i_2 \circ i_3 \mapsto \text{req}_3(i_3) \quad (46)$$

Therefore, the full right term of (33) reads

$$\mathbf{d}_1 \circ (\mathbf{d}_2 \circ \mathbf{d}_3) = \langle \mathbf{A}, \mathbf{D}, \mathbf{I}_{1,3}', \text{prov}_{1,3}', \text{req}_{1,3}' \rangle, \quad (47)$$

where

$$\mathbf{I}_{1,3}' = \{i_1 \circ i_2 \circ i_3 \in \mathbf{I} \mid \text{req}_1(i_1) \leq_{\mathbf{B}} \text{prov}_{2,3}(i_2 \circ i_3) \wedge \text{req}_2(i_2) \leq_{\mathbf{C}} \text{prov}_3(i_3)\}, \quad (48)$$

$$= \{i_1 \circ i_2 \circ i_3 \in \mathbf{I} \mid \text{req}_1(i_1) \leq_{\mathbf{B}} \text{prov}_2(i_2) \wedge \text{req}_2(i_2) \leq_{\mathbf{C}} \text{prov}_3(i_3)\}, \quad (49)$$

$$\text{prov}_{1,3}' : i_1 \circ i_2 \circ i_3 \mapsto \text{prov}_1(i_1), \quad (50)$$

$$\text{req}_{1,3}' : i_1 \circ i_2 \circ i_3 \mapsto \text{req}_{2,3}(i_2 \circ i_3) = \text{req}_3(i_3). \quad (51)$$

It is clear that  $\mathbf{I}_{1,3} = \mathbf{I}_{1,3}'$ ,  $\text{prov}_{1,3} = \text{prov}_{1,3}'$ , and  $\text{req}_{1,3} = \text{req}_{1,3}'$ . This, together with (38) and (47) shows associativity.  $\square$

These two properties are sufficient to conclude that there exists a semicategory of design problems.

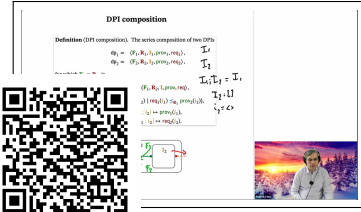
**Lemma 25.23.** DPI is not a category, because one cannot find identities.

*Proof.* We prove this by contradiction. Suppose we can find a DPI that works as an identity for interconnection for any other DPI with implementation space  $\mathbf{I}_{\text{id}}$ . Therefore, when postponed to a DPI with implementation space  $\mathbf{I}_1$  we should have that

$$\mathbf{I}_1 \circ \mathbf{I}_{\text{id}} = \mathbf{I}_1. \quad (52)$$

This implies that  $\mathbf{I}_{\text{id}}$  is an empty list of sets, that is inhabited by only one element, the empty tuple. Therefore, the map  $\text{req}_{\text{id}}$  of the identity is necessarily a constant, because there is only one element in the domain. Therefore, it is impossible to preserve  $\text{req}_1$ .  $\square$

Watch: *Semi-category DPI* (5 minutes).



## 25.6. Sum and intersection of DPIs

### Sum of DPIs

The sum of two design problems with implementation is a design problem with the implementation space  $\mathbf{I} = \mathbf{I}_1 + \mathbf{I}_2$ , and it represents the exclusive choice between two possible alternative families of designs.

**Definition 25.24** (Sum of DPIs)

Given two DPIs with same functionality and resources  $\mathbf{d} = \langle \mathbf{F}, \mathbf{R}, \mathbf{I}_1, \text{prov}_1, \text{req}_1 \rangle$  and  $\mathbf{e} = \langle \mathbf{F}, \mathbf{R}, \mathbf{I}_2, \text{prov}_2, \text{req}_2 \rangle$ , define their sum as

$$\mathbf{d} \vee \mathbf{e} := \langle \mathbf{F}, \mathbf{R}, \mathbf{I}_1 + \mathbf{I}_2, \text{prov}, \text{req} \rangle, \quad (53)$$

where

$$\begin{aligned} \text{prov} &= \text{prov}_1 + \text{prov}_2, \\ \text{req} &= \text{req}_1 + \text{req}_2. \end{aligned} \quad (54)$$

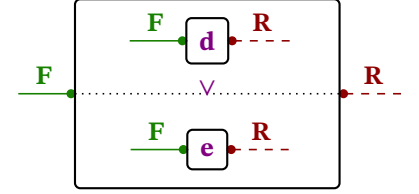


Figure 36.

### Intersection of DPIs

**Definition 25.25** (Intersection of DPIs)

Given two DPIs with same functionality and resources  $\mathbf{d} = \langle \mathbf{F}, \mathbf{R}, \mathbf{I}_1, \text{prov}_1, \text{req}_1 \rangle$  and  $\mathbf{e} = \langle \mathbf{F}, \mathbf{R}, \mathbf{I}_2, \text{prov}_2, \text{req}_2 \rangle$ , define their intersection as

$$\mathbf{d} \wedge \mathbf{e} := \langle \mathbf{F}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle, \quad (55)$$

where

$$\begin{aligned} \mathbf{I} = \{ & (i_1 \circledast i_2 \circledast f \circledast r) \in (\mathbf{I}_1 \circledast \mathbf{I}_2 \circledast \mathbf{F} \circledast \mathbf{R}) \mid \\ & (f \leq_{\mathbf{F}} \text{prov}_1(i_1)) \wedge (f \leq_{\mathbf{F}} \text{prov}_2(i_2)) \wedge \\ & (\text{req}_1(i_1) \leq_{\mathbf{R}} r) \wedge (\text{req}_2(i_2) \leq_{\mathbf{R}} r) \} \end{aligned} \quad (56)$$

and the maps  $\text{prov}$  and  $\text{req}$  are the projections of the third and fourth component.

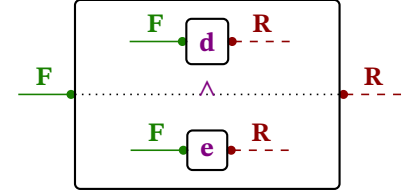


Figure 37.





# 26. Feasibility

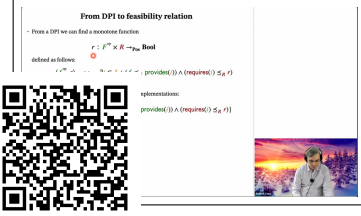
In the previous chapter we have introduced the *design problems with implementations*. Those describe relations among 3 sets: functionality, resources, and implementations.

If we are not interested in the implementations, we can simplify the model and obtain a relation directly between functionality and resources. We obtain in this way a category **DP**.

26.1 DPs as monotone maps . . . . .	378
26.2 DPs as upper sets . . . . .	380
26.3 Querying design problems . . . . .	382
26.4 Series composition . . . . .	384
26.5 Identity for DP . . . . .	387
26.6 The category of design problems DP . . . . .	391

*Steinstossen* is a sport in which the competitors need to throw a heavy stone as far away as possible. It was practiced among the alpine population since prehistoric times.

Watch: *From DPI to DP* (2 minutes).



## 26.1. Design problems as monotone maps

A DPI (Def. 25.1) describes a relation between three sets:  $\mathbf{F}$ ,  $\mathbf{R}$ ,  $\mathbf{I}$ . If we are not interested in the implementations, but just in the relation between  $\mathbf{F}$  and  $\mathbf{R}$ , then we can describe a DPI more compactly as a “DP”.

Recall how the problem Feasibility was defined in Section 25.3. Given a particular functionality  $f$  and resource  $r$ , we would like to know if they are feasible.

This is a function from  $\mathbf{F} \times \mathbf{R}$  to  $\mathbf{Bool}$ :

$$g : \mathbf{F} \times \mathbf{R} \rightarrow \mathbf{Bool}. \quad (1)$$

The value  $g(f, r)$  is the answer to the question “is the functionality  $f$  feasible with resources  $r$ ?”. Due to how the problem is defined, we know that

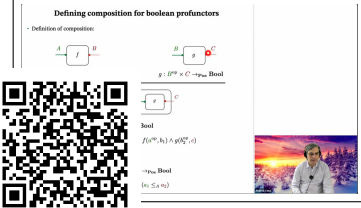
1. If  $f$  is feasible with  $r$ , then any  $f_2 \leq_{\mathbf{F}} f$  is feasible with  $r$ .
2. If  $f$  is feasible with  $r$ , then  $f$  is feasible with any  $r_2 \geq_{\mathbf{R}} r$ .

Therefore, we can conclude that  $g$  is monotone in the second argument  $r$ , and antitone in the first argument  $f$ .

It is going to be convenient to have functions that are monotone in both arguments, and not mixed monotone/antitone. Instead of considering a map from  $\mathbf{F} \times \mathbf{R}$  to  $\mathbf{Bool}$ , we can turn things around and look at a map  $\mathbf{d}$  from  $\mathbf{F}^{\text{op}} \times \mathbf{R}$  to  $\mathbf{Bool}$ , defined as  $\mathbf{d}(f, r) = g(f^*, r)$ . Because we use  $\mathbf{F}^{\text{op}}$  rather than  $\mathbf{F}$ , the map  $\mathbf{d}$  is monotone.

The feasibility map  $\mathbf{d}$  has now forgotten everything about the implementations; however, it does contain all the information we need to solve co-design feasibility problems.

Watch: *Boolean profunctors* (11 minutes).



### Definition 26.1 (Design Problem)

A design problem (DP) is a tuple  $\langle \mathbf{F}, \mathbf{R}, \mathbf{d} \rangle$ , where  $\mathbf{F}, \mathbf{R}$  are posets and  $\mathbf{d}$  is a monotone map of the form

$$\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}. \quad (2)$$

We will also use the notation  $\mathbf{d} : \mathbf{F} \multimap \mathbf{R}$  for design problems, in order to emphasize how we think of them as morphisms. This will be explained below.

**Remark 26.2.** Given a DPI  $\langle \mathbf{F}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle$  it is always possible to obtain from it the following DP

$$\begin{aligned} \mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} &\rightarrow_{\text{Pos}} \mathbf{Bool}, \\ \langle f^*, r \rangle &\mapsto \exists i \in \mathbf{I} : (f \leq_{\mathbf{F}} \text{prov}(i)) \wedge (\text{req}(i) \leq_{\mathbf{R}} r). \end{aligned} \quad (3)$$

Evaluating this DP is the same as asking whether the set

$$\{i \in \mathbf{I} : (f \leq_{\mathbf{F}} \text{prov}(i)) \wedge (\text{req}(i) \leq_{\mathbf{R}} r)\} \quad (4)$$

is empty or not.

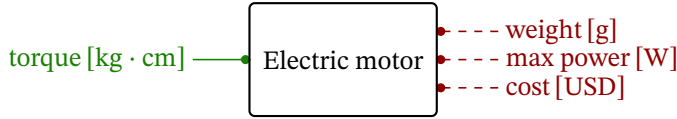
**Example 26.3.** Recall Example 25.4, with the catalogue of electric motors in Table 26.1.

**Table 26.1.:** A simplified catalogue of motors.

Torque [kg · cm]	Motor ID	Weight [g]	Max Power [W]	Cost [USD]
0.18	1204	60.0	2.34	19.95
0.95	1206	140.0	3.00	19.95
0.65	1207	130.0	2.07	12.95
3.7	2267	285.0	4.76	16.95
1.9	2279	165.0	5.40	164.95
19.0	1478	1,000	8.96	49.95
2.2	2299	150.0	5.90	59.95

**Table 26.2.:** Feasibility relations for the design problem of motors.

Torque [kg · cm]	Weight [g]	Max Power [W]	Cost [USD]
0.18	60.0	2.34	19.95
0.95	140.0	3.00	19.95
0.65	130.0	2.07	12.95
3.7	285.0	4.76	16.95
1.9	165.0	5.40	164.95
19.0	1,000	8.96	49.95
2.2	150.0	5.90	59.95



**Figure 1.:** Electric motor design problem.

The catalogue induces a design problem  $\mathbf{d}_{\text{EM}}$ , where each feasibility relation between functionality and resources is reported in Table 26.2. with diagrammatic form as in Fig. 1. In particular, we can query the design problem for combinations of **functionalities** and **resources**. For instance:

$$\mathbf{d}_{\text{EM}}(0.2 \text{ kg} \cdot \text{cm}, \langle 50.0 \text{ g}, 2.0 \text{ W}, 15.0 \text{ USD} \rangle) = \perp, \quad (5)$$

since no listed model can provide  $0.2 \text{ kg} \cdot \text{cm}$  torque by requiring the set of resources  $\langle 50.0 \text{ g}, 2.0 \text{ W}, 15.0 \text{ USD} \rangle$  or less.

#### Graded exercise G.2 (DPSFromMonotoneMaps)

Given any monotone map  $g : \mathbf{F} \rightarrow_{\text{Pos}} \mathbf{R}$ , we can turn it into a design problem

$$\mathbf{d}_g : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$$

via the following recipe. Set

$$\mathbf{d}_g(f^*, r) = \top \quad \text{if and only if} \quad g(f) \leq r.$$

Prove that  $\mathbf{d}_g$ , as defined above, is indeed a design problem when  $g$  is a monotone map.



## 26.2. DPs as upper sets

**Definition 26.4** (Feasible set of a design problem)

We define the *feasible set*  $\mathbf{K}_d$  of a design problem

$$d : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool} \quad (6)$$

as the subset of  $\mathbf{F}^{\text{op}} \times \mathbf{R}$  for which  $d$  is the *indicator function*, that is

$$\mathbf{K}_d = \{\langle f^*, r \rangle \in \mathbf{F}^{\text{op}} \times \mathbf{R} \mid d(f^*, r) = \top\}. \quad (7)$$

Note that the feasibility set  $\mathbf{K}_d$  of a design problem  $d : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$  is a binary relation  $\mathbf{K}_d \subseteq \mathbf{F}^{\text{op}} \times \mathbf{R}$ . We saw in Section 5.6 that there is a one-to-one correspondence between functions  $g : \mathbf{A} \times \mathbf{B} \rightarrow \mathbf{Bool}$  and binary relations  $R : \mathbf{A} \rightarrow \mathbf{B}$ .

An analogous correspondence holds in the context of design problems:

### Graded exercise G.3

Prove that there is a one-to-one correspondence between design problems

$$d : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool} \text{ and upper sets } \mathbf{K}_d \subseteq \mathbf{F}^{\text{op}} \times \mathbf{R}.$$

Recall that when working with a relation  $R : \mathbf{A} \rightarrow \mathbf{B}$  between sets, if the sets in question were finite, then we could conveniently draw the relation  $R$  using arrows to connect elements of  $\mathbf{A}$  to those elements of  $\mathbf{B}$  to which they are related via  $R$ . Given a design problem  $d : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$  involving finite posets, we can visualize it in a similar fashion. We use Hasse diagrams to visualize the posets involved, and we use dashed arrows to connect those elements which are related via the feasibility set  $\mathbf{K}_d$ .

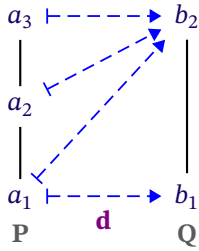


Figure 2.

**Example 26.5.** In Fig. 2 we have illustrated this kind of visualization in the case of a design problem of the type

$$d : \mathbf{P}^{\text{op}} \times \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{Bool}, \quad (8)$$

where  $\mathbf{P} = \langle \mathbf{P}, \leq \rangle$  and  $\mathbf{Q} = \langle \mathbf{Q}, \leq \rangle$  are finite posets, with

$$\mathbf{P} = \{a_1, a_2, a_3\} \quad \text{and} \quad \mathbf{Q} = \{b_1, b_2\}, \quad (9)$$

and ordered as shown in the figure.

The relation described by the design problem is marked with the dashed arrows; The feasibility set

$$\mathbf{K}_d = \{\langle a_1, b_1 \rangle, \langle a_1, b_2 \rangle, \langle a_2, b_2 \rangle, \langle a_3, b_2 \rangle\}, \quad (10)$$

is reported in Fig. 3.

**Remark 26.6.** Given any monotone map  $g : \mathbf{F} \rightarrow_{\text{Pos}} \mathbf{R}$ , we can turn it into a design problem

$$d_g : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool} \quad (11)$$

via the following recipe. Set

$$d_g(f^*, r) = \top \quad \text{if and only if} \quad g(f) \leq r. \quad (12)$$

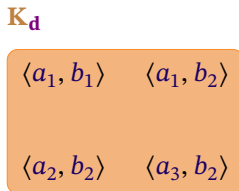


Figure 3.

**Graded exercise G.4**

Prove that  $\mathbf{d}_g$  as defined in Remark 26.6 is indeed a design problem when  $g$  is a monotone map.

The Boolean-valued design problems we are considering in this section do not distinguish between particular implementations: they only tell us if *any* implementation or solution exists for given functionality and resources.

**Graded exercise G.5** (DPsAsUpperSets)

In this exercise, your task is to prove that there is a one-to-one correspondence between design problems  $\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$  and upper sets  $\mathbf{K} \subseteq \mathbf{F}^{\text{op}} \times \mathbf{R}$ .

In more detail:

Let  $\mathbf{A}$  denote the set of all design problems  $\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$  and let  $\mathbf{B}$  denote that set of all upper sets  $\mathbf{K} \subseteq \mathbf{F}^{\text{op}} \times \mathbf{R}$ .

1. Define a function

$$f : \mathbf{A} \rightarrow \mathbf{B}$$

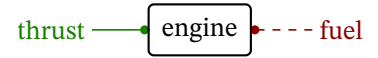
which to any design problem in  $\mathbf{A}$  assigns a corresponding upper set in  $\mathbf{B}$ .

2. Define a function

$$g : \mathbf{B} \rightarrow \mathbf{A}$$

which maps any upper set in  $\mathbf{B}$  to a corresponding design problem in  $\mathbf{A}$ .

3. Prove that  $f$  and  $g$  are inverses to one another.



**Figure 4.:** Diagram of the engine design problem.

### 26.3. Querying design problems

Equation (3) on the one hand, and (4) on the other hand, give two perspectives on the mathematical definition of what we are calling a *design problem*. These two perspectives are analogous to something we already discussed in Section 5.6, when talking about binary relations. There, we said that a binary relation from a set  $\mathbf{A}$  to a set  $\mathbf{B}$  is a subset  $R \subseteq \mathbf{A} \times \mathbf{B}$ , but that such a relation  $R$  can also, equivalently, be viewed as a function  $\phi_R : \mathbf{A} \times \mathbf{B} \rightarrow \{\perp, \top\}$ . The subset  $R \subseteq \mathbf{A} \times \mathbf{B}$  corresponded to the set

$$\{\langle x, y \rangle \in \mathbf{A} \times \mathbf{B} \mid \phi_R(x, y) = \top\}. \quad (13)$$

To make the analogy with (7) more precise, note that  $\mathbf{A}, \mathbf{B}, \{\perp, \top\}$ , and  $\phi_R : \mathbf{A} \times \mathbf{B} \rightarrow \{\perp, \top\}$  live in the category of *sets*, and that  $\mathbf{F}, \mathbf{R}, \mathbf{Bool}$  and  $\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$  live in the category of *posets*.

In Section 5.6, we also discussed two further ways to describe a relation  $R \subseteq \mathbf{A} \times \mathbf{B}$ : namely, we can transform the function  $\phi_R : \mathbf{A} \times \mathbf{B} \rightarrow \{\perp, \top\}$  either into a function

$$\begin{aligned} \hat{\phi}_R : \mathbf{A} &\rightarrow \text{Pow}(\mathbf{B}), \\ x &\mapsto \{y \in \mathbf{B} \mid x R y\}. \end{aligned} \quad (14)$$

or a function

$$\begin{aligned} \check{\phi}_R : \mathbf{B} &\rightarrow \text{Pow}(\mathbf{A}), \\ y &\mapsto \{x \in \mathbf{A} \mid x R y\}. \end{aligned} \quad (15)$$

There are analogous transformations for a design problem  $\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$ . Can you guess what they would be?

In order to use our “sets to posets” analogy and find an answer, it is useful to express the constructions we used in the setting of sets and relations entirely in terms of constructions from the category of sets, if possible. Then the strategy is to identify what are the analogous constructions in the category of posets, and this will allow us to make analogous definitions for design problems.

The functions  $\hat{\phi}_R$  and  $\check{\phi}_R$  above have powersets as their target objects. What is the analogue of the powerset operation in the category of posets?

The answer that we will use goes like this. Given a set  $\mathbf{A}$ , there is a 1-to-1 correspondence between subsets of  $\mathbf{A}$  and functions  $\mathbf{A} \rightarrow \{\perp, \top\}$  (similar to above, a set corresponds here to its indicator function). Thus,  $\text{Pow}(\mathbf{A})$  can be seen to correspond to  $\text{Hom}_{\text{Set}}(\mathbf{A}; \{\perp, \top\})$ . The latter is definitely an expression we can transfer, by analogy, to the category of posets, namely we can consider  $\text{Hom}_{\text{Pos}}(\mathbf{P}; \mathbf{Bool})$ . And from Graded Exercise F.7 we know that monotone maps  $\mathbf{P} \rightarrow \mathbf{Bool}$  correspond to *upper* subsets of  $\mathbf{P}$ . So  $\text{Hom}_{\text{Pos}}(\mathbf{P}; \mathbf{Bool})$  corresponds to set  $\text{USets}(\mathbf{P})$  of upper subsets of  $\mathbf{P}$  (c.f. Section 24.5 for the definitions of upper and lower sets).

We now can write down the “poset” analogues of the functions  $\hat{\phi}_R$  and  $\check{\phi}_R$ . Namely, given a design problem (6), we have associated functions

$$\hat{\mathbf{d}} : \mathbf{F}^{\text{op}} \rightarrow \text{USets}(\mathbf{R}) \quad (16)$$

and

$$\check{\mathbf{d}} : \mathbf{R} \rightarrow \text{USets}(\mathbf{F}^{\text{op}}). \quad (17)$$

However, we are not quite finished: are these monotone functions? Which poset structure can we choose on  $\text{USets}(\mathbf{R})$  and  $\text{USets}(\mathbf{F}^{\text{op}})$ , respectively, so that  $\hat{\mathbf{d}}$  and  $\check{\mathbf{d}}$  are monotone?

#### Graded exercise G.6 (CurryingDesignProblems)

Let  $\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$  be a design problem. In this exercise we will

show that (16) corresponds to a monotone function

$$\mathbf{F} \rightarrow \langle \mathbf{USets}(\mathbf{R}), \supseteq \rangle, \quad (18)$$

and that (17) corresponds to a monotone function

$$\mathbf{R} \rightarrow \langle \mathbf{LSets}(\mathbf{F}), \subseteq \rangle. \quad (19)$$

Here  $\mathbf{USets}(\mathbf{R})$  denotes the set of upper sets of  $\mathbf{R}$  and  $\mathbf{LSets}(\mathbf{F})$  denotes the set of lower sets of  $\mathbf{F}$ .

1. Show that  $\hat{\mathbf{d}}: \mathbf{F}^{\text{op}} \rightarrow \mathbf{USets}(\mathbf{R})$  and  $\check{\mathbf{d}}: \mathbf{R} \rightarrow \mathbf{USets}(\mathbf{F}^{\text{op}})$  are monotone maps when we consider  $\mathbf{USets}(\mathbf{R})$  and  $\mathbf{USets}(\mathbf{F}^{\text{op}})$  to have the partial order corresponding to the inclusion of subsets.
2. Show that the poset  $\langle \mathbf{USets}(\mathbf{F}^{\text{op}}), \subseteq \rangle$  and the poset  $\langle \mathbf{LSets}(\mathbf{F}), \subseteq \rangle$  are isomorphic.
3. Show that there is a 1-to-1 correspondence between monotone functions

$$\mathbf{F}^{\text{op}} \rightarrow \langle \mathbf{USets}(\mathbf{R}), \subseteq \rangle \quad (20)$$

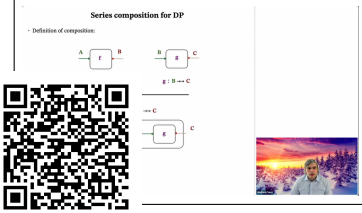
and monotone functions

$$\mathbf{F} \rightarrow \langle \mathbf{USets}(\mathbf{R}), \supseteq \rangle, \quad (21)$$

where in the latter poset, the order is given by the relation of “containment” (as opposed to “inclusion”).

4. Explain, in a few words, why the above steps prove the stated goal of this exercise.

Watch: *Series composition* (8 minutes).



## 26.4. Series composition of design problems

We will define several ways to connect design problems together. The first and most basic way is series composition, or just “composition”.

### Definition 26.7 (Series composition)

Let  $\mathbf{d} : \mathbf{P} \rightarrow \mathbf{Q}$  and  $\mathbf{e} : \mathbf{Q} \rightarrow \mathbf{R}$  be design problems. We define their *series composition*  $(\mathbf{d} \circ \mathbf{e}) : \mathbf{P} \rightarrow \mathbf{R}$  as:

$$(\mathbf{d} \circ \mathbf{e}) : \mathbf{P}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool},$$

$$\langle p^*, r \rangle \mapsto \bigvee_{q \in \mathbf{Q}} \mathbf{d}(p^*, q) \wedge \mathbf{e}(q^*, r). \quad (22)$$

The series composition  $(\mathbf{d} \circ \mathbf{e})$  judges a pair  $\langle p^*, r \rangle$  as feasible if and only if there exists a  $q \in \mathbf{Q}$  such that  $\mathbf{d}(p^*, q)$  and  $\mathbf{e}(q^*, r)$  are feasible.

Given a set  $\mathbf{A}$  and a map  $s : \mathbf{A} \rightarrow \mathbf{Bool}$ , we can define the boolean  $\bigvee_{a \in \mathbf{A}} s(a)$  by

$$\bigvee_{a \in \mathbf{A}} s(a) := \begin{cases} \top & \text{if there exists } a \in \mathbf{A} \text{ for which } s(a) = \top, \\ \perp & \text{if there exists no } a \in \mathbf{A} \text{ for which } s(a) = \top. \end{cases} \quad (23)$$

In (22) we could have written “ $\exists_{q \in \mathbf{Q}}$ ” instead of “ $\bigvee_{q \in \mathbf{Q}}$ ”:

$$\exists_{q \in \mathbf{Q}} \mathbf{d}(p^*, q) \wedge \mathbf{e}(q^*, r). \quad (24)$$

Using  $\bigvee$  form highlights the connection with an integration operation  $\int_q$ .

We use the same diagrammatic notation for DPs as for DPIs. We represent series composition as

$$\text{---} \bullet \boxed{\mathbf{d}} \text{---} \circlearrowleft \text{---} \bullet \boxed{\mathbf{e}} \text{---} \equiv \text{---} \bullet \boxed{\mathbf{d} \circ \mathbf{e}} \text{---} \quad (25)$$

One can notice the “co-design constraint”  $\circlearrowleft$ , which can be interpreted as follows. The **resource** required by a component is limited by the **functionality** produced by another component.

When viewing compositions (and larger diagrams) formed from these boxes, it is tempting to interpret the boxes as input-output processes. However, that would be misleading. The arrows do not represent information flow, materials flow, or energy flow. Design problems do not represent input-output processes but rather a static calculus of requirements—a requirements flow.

When the posets involved are finite, the series composition of design problems can be calculated visually, using the kind of representation discussed in Example 26.5.

To explain how this works, consider the design problem

$$\mathbf{d} : \mathbf{P}^{\text{op}} \times \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{Bool}, \quad (26)$$

from Example 26.5, visualized again for convenience in the first row of Fig. 5.

And consider another design problem of the type

$$\mathbf{e} : \mathbf{Q}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}, \quad (27)$$

as given by the visualization in the first row of Fig. 5.

We can calculate the series composition  $\mathbf{d} \circ \mathbf{e}$  by tracing paths in the “composite” visualization given in the second row of Fig. 5. Namely, a pair  $\langle a, c \rangle$  is in the feasibility set of  $\mathbf{d} \circ \mathbf{e}$  if and only if we can trace a path from  $a$  to  $c$  by only moving

upwards in the posets  $\mathbf{P}$ ,  $\mathbf{Q}$ , and  $\mathbf{R}$ , or crossing from one poset to another following dashed arrows in the direction they are pointing. Thus, the visualization of the composite  $\mathbf{d} \circledast \mathbf{e}$  is as in the third row of Fig. 5.

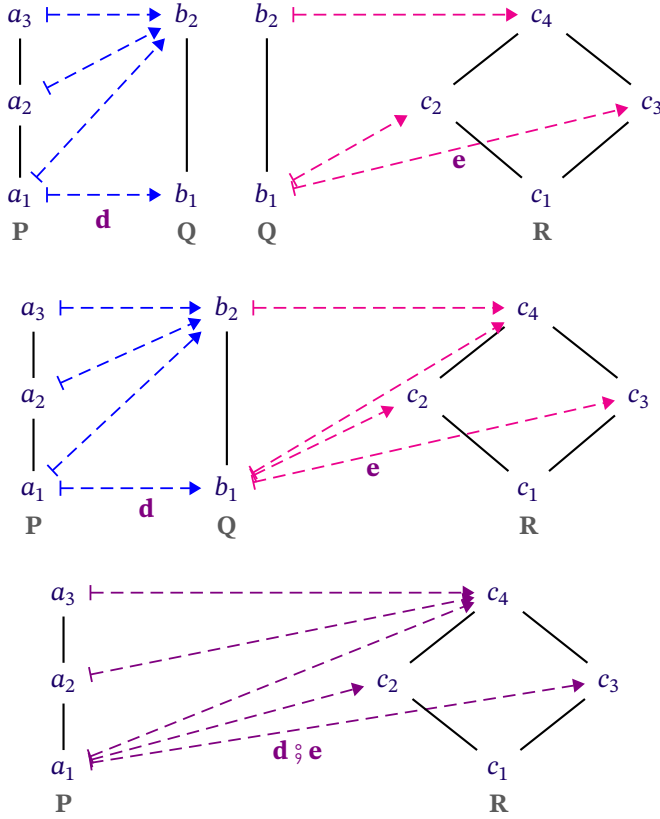


Figure 5.

Let us check that, given design problems  $\mathbf{d}$  and  $\mathbf{e}$ , their series composition  $(\mathbf{d} \circledast \mathbf{e})$  is in fact a design problem.

**Lemma 26.8.** Series composition as in (22) is monotone in  $p$  and  $r$ .

*Proof.* We need to show that  $(\mathbf{d} \circledast \mathbf{e})(p^*, r)$  is monotone in  $p^*$  and  $r$ . Because  $\mathbf{d}$  represents a design problem,  $\mathbf{d}(p^*, q)$  is monotone in  $p^*$ , and similarly  $\mathbf{e}(q^*, r)$  is monotone in  $r$ . The conjunction “ $\wedge$ ” is monotone in both variables, and likewise the “ $\vee$ ” operation.  $\square$

We can show two important properties for the “ $\circledast$ ” operation: associativity and unitality.

**Lemma 26.9.** The series composition operation as in (22) is associative:

$$(\mathbf{d} \circledast \mathbf{e}) \circledast \mathbf{g} = \mathbf{d} \circledast (\mathbf{e} \circledast \mathbf{g}). \quad (28)$$

*Proof.* Consider  $\mathbf{d} : \mathbf{P} \rightarrow \mathbf{Q}$ ,  $\mathbf{e} : \mathbf{Q} \rightarrow \mathbf{R}$ ,  $\mathbf{g} : \mathbf{R} \rightarrow \mathbf{S}$ . To show that the

operation is associative, we can use distributivity and commutativity in **Bool**:

$$\begin{aligned}
 ((\mathbf{d} \circledast \mathbf{e}) \circledast \mathbf{g})(p^*, s) &= \bigvee_{r \in \mathbf{R}} \left( \bigvee_{q \in \mathbf{Q}} \mathbf{d}(p^*, q) \wedge \mathbf{e}(q^*, r) \right) \wedge \mathbf{g}(r^*, s) \\
 &= \bigvee_{r \in \mathbf{R}} \left( \bigvee_{q \in \mathbf{Q}} \mathbf{d}(p^*, q) \wedge \mathbf{e}(q^*, r) \wedge \mathbf{g}(r^*, s) \right) \quad (29) \\
 &= \bigvee_{q \in \mathbf{Q}} \mathbf{d}(p^*, q) \wedge \left( \bigvee_{r \in \mathbf{R}} \mathbf{e}(q^*, r) \wedge \mathbf{g}(r^*, s) \right) \\
 &= (\mathbf{d} \circledast (\mathbf{e} \circledast \mathbf{g}))(p^*, s).
 \end{aligned}$$

□

Because of associativity, we can write  $\mathbf{d} \circledast \mathbf{e} \circledast \mathbf{g}$  without ambiguity. Associativity of composition is represented as:

$$\begin{aligned}
 \text{---} \bullet \boxed{\mathbf{d}} \bullet \leq \bullet \boxed{\mathbf{e}} \bullet \leq \bullet \boxed{\mathbf{g}} \bullet \text{---} &\equiv \text{---} \bullet \boxed{(\mathbf{d} \circledast \mathbf{e})} \bullet \leq \bullet \boxed{\mathbf{g}} \bullet \text{---} \\
 &\equiv \text{---} \bullet \boxed{\mathbf{d}} \bullet \leq \bullet \boxed{(\mathbf{e} \circledast \mathbf{g})} \bullet \text{---} \quad (30)
 \end{aligned}$$

## 26.5. Identity for DP

There exists an identity for the “ $\circ$ ” operation. We define the identity  $\text{id}_P : P \rightarrow P$  as follows.

**Definition 26.10** (Identity design problem)

For any poset  $P$ , the *identity design problem*  $\text{id}_P : P \rightarrow P$  is a monotone map

$$\begin{aligned} \text{id}_P : P^{\text{op}} \times P &\rightarrow_{\text{Pos}} \text{Bool}, \\ \langle p_1^*, p_2 \rangle &\mapsto p_1 \leq_P p_2. \end{aligned} \quad (31)$$

**Remark 26.11** (Monotonicity of the identity). Let’s consider  $p_1' \leq_P p_1$ . If it holds  $p_1 \leq_P p_2$ , then it also holds  $p_1' \leq_P p_2$ . Similarly, now consider  $p_2 \leq_P p_2'$ . If it holds  $p_1 \leq_P p_2$ , then it also holds  $p_1 \leq_P p_2'$ .

In the diagrammatic notation, we represent  $\text{id}_P$  as:

$$P \xrightarrow{\text{id}_P} P \quad (32)$$

**Lemma 26.12.** The series composition operation as in (22) satisfies the left and right unit laws ((33)).

$$\xrightarrow{\text{id}_P} \leq \text{id}_Q \equiv \xrightarrow{\text{id}_Q} \quad (33)$$

*Proof.* Given  $d : P \rightarrow Q$ , we need to show:

$$\text{id}_P \circ d = d = d \circ \text{id}_Q. \quad (34)$$

In the following, we prove  $\text{id}_P \circ d = d$ . Proving  $d \circ \text{id}_Q = d$  is similar. Consider the poset **Bool**. Since for  $x, y \in \text{Bool}$ ,

$$\frac{x \cong y}{x = y}, \quad (35)$$

(also referred to as skeletality [6]), we just need to show that  $d \leq \text{id}_P \circ d$  and  $\text{id}_P \circ d \leq d$ . Here,  $d \leq e$  means  $d(p^*, q) \leq_{\text{Bool}} e(p^*, q)$  for all  $p \in P, q \in Q$ . We have

$$\begin{aligned} d(p^*, q) &= \top \wedge d(p^*, q) \\ &= \text{id}_P(p^*, p) \wedge d(p^*, q) \\ &\leq \bigvee_{p' \in P} \text{id}_P(p^*, p') \wedge d(p^*, q) \\ &= (\text{id}_P \circ d)(p^*, q). \end{aligned} \quad (36)$$

For the other direction, we need to show that  $\text{id}_P \circ d \leq d$ :

$$\bigvee_{p' \in P} \text{id}_P(p^*, p') \wedge d(p^*, q) \leq d(p^*, q). \quad (37)$$

This holds if and only if  $\text{id}_P(p^*, p') \wedge d(p^*, q) \leq d(p^*, q)$  for some  $p' \in P$ . If there is no such  $p'$ , then the inequality holds ( $\perp \leq \perp$  and  $\perp \leq \top$ ). If there is such an element  $p'$ , it means that  $\text{id}_P(p^*, p') = \top$  and  $d(p^*, q) = \top$ . We



know that

$$\frac{\text{id}_P(p^*, p') = \top}{p \leq p'} \quad (38)$$

and hence  $\mathbf{d}(p^*, q) = \top$ . □

**Graded exercise G.7** (ComposingDesignProblems)

Consider the following posets, given in terms of Hasse diagrams:

▷ **Speed:**

fast  
|  
slow

▷ **Size:**

large  
|  
small

▷ **Money:**

120K  
|  
40K  
|  
20K

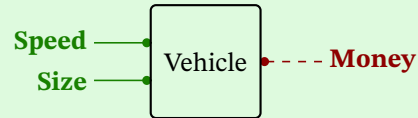
▷ **Time:**

long  
|  
short

Furthermore let the poset **Speed** × **Size** be equipped with the standard product partial ordering.

Consider the design problem given by the monotone function

$$f : (\mathbf{Speed} \times \mathbf{Size})^{\text{op}} \times \mathbf{Money} \rightarrow \mathbf{Bool} \quad (39)$$

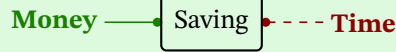


with

$$\begin{aligned} f^{-1}(\top) = \{ & \langle \langle \text{slow}, \text{small} \rangle, 20\text{K} \rangle, \langle \langle \text{slow}, \text{small} \rangle, 40\text{K} \rangle, \langle \langle \text{slow}, \text{small} \rangle, 120\text{K} \rangle, \\ & \langle \langle \text{slow}, \text{large} \rangle, 40\text{K} \rangle, \langle \langle \text{slow}, \text{large} \rangle, 120\text{K} \rangle \\ & \langle \langle \text{fast}, \text{small} \rangle, 120\text{K} \rangle \} \end{aligned}$$

along with the design problem given by the monotone function

$$g : \mathbf{Money}^{\text{op}} \times \mathbf{Time} \rightarrow \mathbf{Bool} \quad (40)$$



with

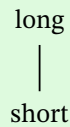
$$g^{-1}(T) = \{\langle 20K, \text{short} \rangle, \langle 20K, \text{long} \rangle, \langle 40K, \text{long} \rangle\}.$$

1. Compute the series composition  $f \circ_{\text{DP}} g$  in the category of design problems. The result should be a design problem described in terms of a monotone function  $(\text{Speed} \times \text{Size})^{\text{op}} \times \text{Time} \rightarrow \text{Bool}$ .
2. We interpret elements of  $\text{Speed} \times \text{Size}$  as properties of cars that Alice considers buying. **Money** represents the amounts of money that she would need to buy one of said cars, and **Time** represents the amounts of time Alice could spend saving money. The feasibility relations  $f$  and  $g$  describe what is possible for Alice. According to  $f \circ_{\text{DP}} g$ , is it feasible for Alice to buy a fast small car? If yes, will she have to work, at minimum, a long or a short amount of time in order to save enough money to buy it?

### Graded exercise G.8 (DPComposition)

Consider the following posets, given in terms of Hasse diagrams:

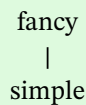
▷ **Time:**



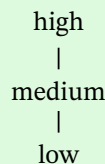
▷ **Money:**



▷ **Tech complexity:**



▷ **Task complexity:**



Let the poset  $\text{Time} \times \text{Money}$  be equipped with the standard product partial ordering.

You are in charge of an engineering team that should develop a robotic system. It is not yet clear what strategy you wish follow for realizing the project, and you wish to do a preliminary feasibility study. You will potentially need to make some tradeoffs between the *time* that your team works on the project and the *money* that you invest. Furthermore, you will need to decide between implementing either a cutting edge *fancy* robotic system, or a *simple* one. This will have an impact on the *complexity of the tasks* that the system will be able to handle.

Design problems capture feasibility relations. In your case, the feasibility relation between **Task complexity** and **Tech complexity** is given by the

design problem

$$f : \text{Task complexity}^{\text{op}} \times \text{Tech complexity} \rightarrow \text{Bool} \quad (41)$$

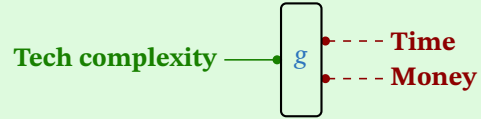


with

$$f^{-1}(\text{T}) = \{ \langle \text{high}, \text{fancy} \rangle, \langle \text{medium}, \text{fancy} \rangle, \langle \text{low}, \text{fancy} \rangle, \langle \text{low}, \text{simple} \rangle \}$$

and the feasibility relation between **Tech complexity** and **Time**  $\times$  **Money** is given by the design problem

$$g : \text{Tech complexity}^{\text{op}} \times (\text{Time} \times \text{Money}) \rightarrow \text{Bool} \quad (42)$$



with

$$g^{-1}(\text{T}) = \{ \langle \text{fancy}, \langle \text{long}, 100\text{K} \rangle \rangle, \langle \text{simple}, \langle \text{long}, 100\text{K} \rangle \rangle, \langle \text{simple}, \langle \text{long}, 20\text{K} \rangle \rangle, \langle \text{simple}, \langle \text{short}, 100\text{K} \rangle \rangle \}.$$

Your tasks in this exercise:

1. Compute the series composition  $f \circ_{\text{DP}} g$  in the category of design problems.
2. Based on the previous calculation, would it be feasible to build a robotic system that can handle medium-complexity tasks if your team works only a short time on the project but invests 100K?

## 26.6. The category of design problems **DP**

Finally, we can declare that the design problems so defined are morphisms in a category that we call **DP**.

### Definition 26.13 (Category of design problems **DP**)

The category of design problems **DP** consists of the following constituents:

1. *Objects*: The objects of **DP** are posets.
2. *Morphisms*: The morphisms of **DP** are design problems (Def. 26.1).
3. *Identity morphism*: The identity morphism  $\text{id}_{\mathbf{P}} : \mathbf{P} \leftrightarrow \mathbf{P}$  is given by Def. 26.10.
4. *Composition operation*: Given morphisms  $\mathbf{d} : \mathbf{P} \leftrightarrow \mathbf{Q}$  and  $\mathbf{e} : \mathbf{Q} \leftrightarrow \mathbf{R}$ , their composition  $\mathbf{d} \circ \mathbf{e} : \mathbf{P} \leftrightarrow \mathbf{R}$  is given by Def. 26.7.

We have already shown that the composition operator “ $\circ$ ” is associative and unital, and that the composition of two design problems is a design problem (closure). Therefore, **DP** is a category.

**DP** is called **Feas** or  $\mathbb{P}\text{rof}_{\text{Bool}}$  in [6].

### Relation between **DPI** and **DP**

We have already seen in Remark 26.2 that we can obtain a **DP** from a **DPI**. We can make this more formal and say that there exists a forgetful semifunctor from **DPI** to **DP**.

### Definition 26.14

The forgetful semifunctor  $F : \mathbf{DPI} \rightarrow \mathbf{DP}$  is given by:

1. Identity on the objects:  $F_*(\mathbf{P}) = \mathbf{P}$ .
2. Given  $\mathbf{d} = \langle \mathbf{F}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle$ , the action on morphisms is given by

$$F_*(\mathbf{d}) : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}, \quad (43)$$

$$\langle f^*, r \rangle \mapsto \exists i \in \mathbf{I} : (f \leq_{\mathbf{F}} \text{prov}(i)) \wedge (\text{req}(i) \leq_{\mathbf{R}} r).$$

**Lemma 26.15.** Def. 26.14 indeed defines a semifunctor.

*Proof.* Consider

$$\begin{aligned} \mathbf{d} &= \langle \mathbf{P}, \mathbf{Q}, \mathbf{I}_1, \text{prov}_1, \text{req}_1 \rangle, \\ \mathbf{e} &= \langle \mathbf{Q}, \mathbf{R}, \mathbf{I}_2, \text{prov}_2, \text{req}_2 \rangle, \end{aligned} \quad (44)$$

We need to show

$$F_*(\mathbf{d} \circ_{\mathbf{DPI}} \mathbf{e}) = F_*(\mathbf{d}) \circ_{\mathbf{DP}} F_*(\mathbf{e}). \quad (45)$$

Let's start with the left term. One has

$$F_*(\mathbf{d} \circ_{\mathbf{DPI}} \mathbf{e})(p^*, r) = \exists i \in \mathbf{I} : (p \leq_{\mathbf{P}} \text{prov}_1(i_1)) \wedge (\text{req}_2(i_2) \leq_{\mathbf{R}} r), \quad (46)$$

where  $\mathbf{I} = \{i_1 \circ i_2 \in \mathbf{I}_1 \circ \mathbf{I}_2 \mid \text{req}_1(i_1) \leq_{\mathbf{Q}} \text{prov}_2(i_2)\}$ .

On the other hand,

$$\begin{aligned}
 & (F_{\rightarrow}(\mathbf{d}) \mathbin{\text{\textcircled{\tiny DP}}} F_{\rightarrow}(\mathbf{e}))(p^*, r) \\
 &= \bigvee_{q \in Q} F_{\rightarrow}(\mathbf{d})(p^*, q) \wedge F_{\rightarrow}(\mathbf{e})(q^*, r) \\
 &= \bigvee_{q \in Q} (\exists i_1 \in \mathbf{I}_1 : (p \leq_P \text{prov}_1(i_1)) \wedge (\text{req}_1(i_1) \leq_Q q)) \\
 &\quad \wedge (\exists i_2 \in \mathbf{I}_2 : (q \leq_Q \text{prov}_2(i_2)) \wedge (\text{req}_2(i_2) \leq_R r))
 \end{aligned} \tag{47}$$

Consider the following cases:

▷ If  $F_{\rightarrow}(\mathbf{d} \mathbin{\text{\textcircled{\tiny DPI}}} \mathbf{e})(p^*, r) = \top$ , there exist  $i_1 \in \mathbf{I}_1, i_2 \in \mathbf{I}_2$  for which

$$\begin{aligned}
 & p \leq_P \text{prov}_1(i_1), \\
 & \text{req}_2(i_2) \leq_R r, \\
 & \text{req}_1(i_1) \leq_Q \text{prov}_2(i_2).
 \end{aligned} \tag{48}$$

The first two terms are clear, and the last term implies that there exists a  $q \in Q$  such that

$$(q \leq_Q \text{prov}_2(i_2)) \wedge (\text{req}_1(i_1) \leq_Q q), \tag{49}$$

implying  $(F_{\rightarrow}(\mathbf{d}) \mathbin{\text{\textcircled{\tiny DP}}} F_{\rightarrow}(\mathbf{e}))(p^*, r) = \top$ .

▷ The case

$$\frac{F_{\rightarrow}(\mathbf{d} \mathbin{\text{\textcircled{\tiny DPI}}} \mathbf{e})(p^*, r) = \perp}{(F_{\rightarrow}(\mathbf{d}) \mathbin{\text{\textcircled{\tiny DP}}} F_{\rightarrow}(\mathbf{e}))(p^*, r) = \perp} \tag{50}$$

follows analogously.

▷ The other direction is easier to show, since clearly

$$\frac{(F_{\rightarrow}(\mathbf{d}) \mathbin{\text{\textcircled{\tiny DP}}} F_{\rightarrow}(\mathbf{e}))(p^*, r) = \top}{F_{\rightarrow}(\mathbf{d} \mathbin{\text{\textcircled{\tiny DPI}}} \mathbf{e})(p^*, r) = \top} \tag{51}$$

and

$$\frac{(F_{\rightarrow}(\mathbf{d}) \mathbin{\text{\textcircled{\tiny DP}}} F_{\rightarrow}(\mathbf{e}))(p^*, r) = \perp}{F_{\rightarrow}(\mathbf{d} \mathbin{\text{\textcircled{\tiny DPI}}} \mathbf{e})(p^*, r) = \perp} \tag{52}$$

by inspecting (46) and (47).

□

In the other direction, we can take a DP and find a corresponding DPI. We obtain another semifunctor.

### Definition 26.16

The semifunctor  $G : \mathbf{DP} \rightarrow \mathbf{DPI}$  is given by:

1. Identity on the objects:  $F_{\bullet}(\mathbf{P}) = \mathbf{P}$ .
2. Given  $\mathbf{d} : \mathbf{F}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$ , the action on morphisms is given by

$$G_{\rightarrow}(\mathbf{d}) = \langle \mathbf{F}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle, \tag{53}$$

where

$$\begin{aligned}
 \mathbf{I} &= \{ \langle f, r \rangle \in \mathbf{F} \times \mathbf{R} : \mathbf{d}(f^*, r) \}, \\
 \text{prov} &: \langle f, r \rangle \mapsto f, \\
 \text{req} &: \langle f, r \rangle \mapsto r,
 \end{aligned} \tag{54}$$

**Lemma 26.17.** Def. 26.16 indeed defines a semifunctor.

*Proof.* Consider  $\mathbf{d} : \mathbf{P} \times \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{Bool}$  and  $\mathbf{e} : \mathbf{Q} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool}$ . We need to show

$$G_{\rightarrow}(\mathbf{d} \circ_{\text{DP}} \mathbf{e}) = G_{\rightarrow}(\mathbf{d}) \circ_{\text{DPI}} G_{\rightarrow}(\mathbf{e}). \quad (55)$$

Let's start with the left term. One has

$$G_{\rightarrow}(\mathbf{d} \circ_{\text{DP}} \mathbf{e}) = \langle \mathbf{P}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle, \quad (56)$$

where

$$\begin{aligned} \mathbf{I} &= \{ \langle p, r \rangle \in \mathbf{P} \times \mathbf{R} : (\mathbf{d} \circ_{\text{DP}} \mathbf{e})(p, r) \}, \\ &= \{ \langle p, r \rangle \in \mathbf{P} \times \mathbf{R} : \bigvee_{q \in \mathbf{Q}} \mathbf{d}(p^*, q) \wedge \mathbf{e}(q^*, r) \} \end{aligned} \quad (57)$$

$$\text{prov} = \text{pr}_s,$$

$$\text{req} = \text{pr}_e,$$

where  $\text{pr}_s$  is the projection mapping returning the first element of a tuple, and  $\text{pr}_e$  is the projection mapping returning the last element of a tuple. For the right-hand side, instead, one has

$$G_{\rightarrow}(\mathbf{d}) \circ_{\text{DPI}} G_{\rightarrow}(\mathbf{e}) = \langle \mathbf{P}, \mathbf{R}, \mathbf{I}', \text{prov}', \text{req}' \rangle, \quad (58)$$

where

$$\begin{aligned} \mathbf{I}' &= \{ i_1 \circ_{\mathbf{Q}} i_2 \in \mathbf{I}_1 \circ_{\mathbf{Q}} \mathbf{I}_2 \mid \text{req}_1(i_1) \leq_{\mathbf{Q}} \text{prov}_2(i_2) \} \\ \mathbf{I}_1 &= \{ \langle p, q \rangle \in \mathbf{P} \times \mathbf{Q} : \mathbf{d}(p, q) \}, \\ \mathbf{I}_2 &= \{ \langle q, r \rangle \in \mathbf{Q} \times \mathbf{R} : \mathbf{e}(q, r) \}, \\ \text{prov}_1 &= \text{pr}_s \\ \text{prov}_2 &= \text{pr}_s \\ \text{req}_1 &= \text{pr}_e \\ \text{req}_2 &= \text{pr}_e \\ \text{prov}' &= \text{pr}_s \\ \text{req}' &= \text{pr}_e. \end{aligned} \quad (59)$$

One can already notice that  $\text{prov} = \text{prov}'$  and  $\text{req} = \text{req}'$ . To finish the proof, we can massage  $\mathbf{I}'$ :

$$\begin{aligned} \mathbf{I}' &= \{ i_1 \circ_{\mathbf{Q}} i_2 \in \mathbf{I}_1 \circ_{\mathbf{Q}} \mathbf{I}_2 \mid \text{req}_1(i_1) \leq_{\mathbf{Q}} \text{prov}_2(i_2) \} \\ &= \{ \langle p, q, q', r \rangle \in \mathbf{P} \times \mathbf{Q} \times \mathbf{Q} \times \mathbf{R} : \mathbf{d}(p, q) \wedge \mathbf{e}(q', r) \wedge q \leq_{\mathbf{Q}} q' \} \\ &= \{ \langle p, r \rangle \in \mathbf{P} \times \mathbf{R} : \bigvee_{q \in \mathbf{Q}} \mathbf{d}(p, q) \wedge \mathbf{e}(q, r) \} \\ &= \mathbf{I}. \end{aligned} \quad (60)$$

□





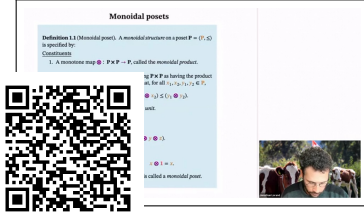


## 27. Lattices

27.1 Monoidal posets . . . . .	396
27.2 Monoidal-time procedures . . . .	398
27.3 Lattices . . . . .	400
27.4 Lattice homomorphisms . . . . .	403
27.5 Categories Lat and BoundedLat	404



Watch: *Monoidal posets* (5 minutes).



## 27.1. Monoidal posets

A monoidal poset is a poset that is also a monoid, and in which the monoidal product is a monotone map that is compatible with the order.

### Definition 27.1 (Monoidal poset)

A *monoidal structure* on a poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  is specified by:

#### Constituents

1. A monotone map  $\otimes : \mathbf{P} \times \mathbf{P} \rightarrow_{\text{pos}} \mathbf{P}$ , called the *monoidal product*.
2. An element  $\mathbf{1} \in \mathbf{P}$ , called the *monoidal unit*.

#### Conditions

1. Associativity: for all  $x, y, z \in \mathbf{P}$ :

$$(x \otimes y) \otimes z = x \otimes (y \otimes z). \quad (1)$$

2. Left and right unitality: for all  $x \in \mathbf{P}$ :

$$\mathbf{1} \otimes x = x \quad \text{and} \quad x \otimes \mathbf{1} = x. \quad (2)$$

A poset equipped with a monoidal structure is called a *monoidal poset*.

Note that here we are implicitly assuming  $\mathbf{P} \times \mathbf{P}$  as having the product order (Def. 10.1). In detail, monotonicity means that, for all  $x_1, x_2, y_1, y_2 \in \mathbf{P}$ :

$$\frac{x_1 \leq_{\mathbf{P}} y_1 \quad x_2 \leq_{\mathbf{P}} y_2}{(x_1 \otimes x_2) \leq_{\mathbf{P}} (y_1 \otimes y_2)}. \quad (3)$$

### Definition 27.2 (Symmetric monoidal poset)

A *symmetric monoidal poset* is a monoidal poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}}, \otimes, \mathbf{1} \rangle$  such that, for all  $x, y \in \mathbf{P}$ ,

$$x \otimes y = y \otimes x. \quad (4)$$

**Example 27.3** (Reals with addition). Consider the real numbers  $\mathbb{R}$  with the poset structure given the usual ordering. Consider 0 as the monoidal unit and the operation  $+$  :  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  as monoidal product. It is easy to see that the conditions of Def. 27.1 are satisfied:

- (a) Given  $p_1, p_2, q_1, q_2 \in \mathbb{R}$ , we know:

$$\frac{p_1 \leq p_2 \quad q_1 \leq q_2}{(p_1 + p_2) \leq (q_1 + q_1)}. \quad (5)$$

- (b)  $0 + p = p + 0 = 0, \forall p \in \mathbb{R}$ .

- (c)  $(p + q) + r = p + (q + r), \forall p, q, r \in \mathbb{R}$ .

**Counter-example 27.4.** Someone proposes now to substitute the monoidal unit in Example 27.3 with 1 and the monoidal product with multiplication “ $\cdot$ ”. This does not form a monoidal poset anymore. To see a simple counterexample, consider the fact that  $-5 \leq 0$  and  $-4 \leq 3$ . However,  $(-5) \cdot (-4) \not\leq 0 \cdot 3$ .

**Example 27.5** (Boolean monoid). The booleans form a monoidal poset  $\langle \mathbf{Bool}, \leq_{\mathbf{Bool}}, \top, \wedge \rangle$  with the unit being  $\top$  and the product being  $\wedge$ . The action of the monoidal product “ $\wedge$ ” can be summarized in the table on the side. From this table,

$\wedge$	$\perp$	$\top$
$\perp$	$\perp$	$\perp$
$\top$	$\perp$	$\top$

it is clear that given  $x_1 \leq_{\text{Bool}} y_1$  and  $x_2 \leq_{\text{Bool}} y_2$ , we have  $x_1 \wedge x_2 \leq_{\text{Bool}} y_1 \wedge y_2$  (if you do not believe it, try all possible combinations). Furthermore,  $x \wedge \top = x = \top \wedge x$ .

**Graded exercise G.9** (HwkMonoidalPosets)

Prove or disprove that the following are monoidal posets:

1. The set  $\mathbb{R}$  equipped with the usual ordering, addition as monoidal product, and  $0 \in \mathbb{R}$  as monoidal unit.
2. The set  $\mathbb{R}$  equipped with the usual ordering, multiplication as monoidal product, and  $1 \in \mathbb{R}$  as monoidal unit.

**Graded exercise G.10** (HwkInternalHomCancelling)

Let  $\mathbf{P} = \langle \mathbf{P}, \leq, \otimes, \mathbf{1}, \backslash \rangle$  be a closed monoidal poset. Prove that for any  $x, y, z \in \mathbf{P}$ , we have

$$(x \backslash y) \otimes (y \backslash z) \leq x \backslash z. \quad (6)$$

**Graded exercise G.11** (HwkInternalLeastUpperBounds) 1. Let  $\mathbf{P} = \langle \mathbf{P}, \leq \rangle$  be a poset, and consider a subset  $\mathbf{S} \subseteq \mathbf{P}$ . A *least upper bound*, or *join*, for  $\mathbf{S}$  is an element  $x \in \mathbf{P}$  which satisfies the conditions

- a)  $y \leq x \ \forall y \in \mathbf{S}$ ;
- b) if  $x' \in \mathbf{P}$  is such that  $y \leq x' \ \forall y \in \mathbf{S}$ , then  $x \leq x'$  must hold.

Prove that, if a least upper bound of a subset  $\mathbf{S}$  exists, then it is unique. In this case we use the notation  $\bigvee \mathbf{S}$  to denote it.

2. Suppose that we have a Galois connection

$$\begin{array}{ccc} & f & \\ \text{P} & \xrightarrow{\quad} & \text{Q} \\ & g & \\ & \perp & \end{array} \quad (7)$$

between posets  $\mathbf{P}$  and  $\mathbf{Q}$ . Furthermore, assume that  $\mathbf{P}$  has all joins, meaning that for any subset  $\mathbf{S} \subseteq \mathbf{P}$ , the least upper bound  $\bigvee \mathbf{S}$  exists and is an element of  $\mathbf{P}$ .

Prove that for any subset  $\mathbf{S} \subseteq \mathbf{P}$  it holds that

$$f(\bigvee \mathbf{S}) = \bigvee \{f(x) \mid x \in \mathbf{S}\}. \quad (8)$$

## 27.2. Monoidal-time procedures

Before, we thought of sized sets (Def. 15.7) as a datatype that can be measured with integer sizes. However, this does not capture some important cases. For example, if we are dealing with *trees*, from the point of view of computation it could be important to think about multidimensional sizes: for example, we might want to account for number of nodes, number of edges, maximum branching factor, and so on. Still, we want to know when an instance is bigger than another: this is a perfect job for posets.

### Definition 27.6 (Poset-sized sets)

A *poset-sized set* is a tuple  $\langle \mathbf{A}, \Sigma_{\mathbf{A}}, \text{size} \rangle$ , where  $\mathbf{A}$  is a set,  $\Sigma_{\mathbf{A}}$  is a poset, and  $\text{size} : \mathbf{A} \rightarrow \Sigma_{\mathbf{A}}$  is the size function.

In **ProcSizeTime**, we assumed that time was measured using real numbers. We can generalize this to an arbitrary poset, for example allowing counting “number of operations”. We need an additional structure: in (34) on page 255 we needed a “+” to sum the time of successive procedures. Therefore, we assume that we have a time monoidal poset  $\mathbf{T}$ .

### Definition 27.7 (Semicategory **ProcSizeTime** <sub>$\mathbf{T}$</sub> )

For a given monoidal poset  $\langle \mathbf{T}, \otimes_{\mathbf{T}} \rangle$ , the semicategory **ProcSizeTime** <sub>$\mathbf{T}$</sub>  consists of the following constituents:

1. *Objects*: The objects are poset-sized sets.
2. *Morphisms*: A morphism

$$f : X \rightarrow_{\text{ProcSize}} X \quad (9)$$

between the two objects

$$X = \langle \mathbf{A}, \Sigma_{\mathbf{A}}, \text{size}_{\mathbf{A}} \rangle \quad \text{and} \quad Y = \langle \mathbf{B}, \Sigma_{\mathbf{B}}, \text{size}_{\mathbf{B}} \rangle \quad (10)$$

is a tuple

$$\langle f_e, \sigma, \text{dur} \rangle, \quad (11)$$

where:

- a)  $f_e : \mathbf{A} \rightarrow \mathbf{B}$  is the function computed;
  - b)  $\sigma : \Sigma_{\mathbf{A}} \rightarrow_{\text{Pos}} \Sigma_{\mathbf{B}}$  is a monotone function that keeps track of how the size changes.
  - c)  $\text{dur} : \Sigma_{\mathbf{A}} \rightarrow_{\text{Pos}} \mathbf{T}$  is a monotone function that gives computation time as a function of instance size;
3. *Composition*: The composition of

$$\langle f_1, \sigma_1, \text{dur}_1 \rangle \quad \text{and} \quad \langle g_2, \sigma_2, \text{dur}_2 \rangle \quad (12)$$

is given by

$$\langle f_{1;2}, \sigma_{1;2}, \text{dur}_{1;2} \rangle, \quad (13)$$

where

$$f_{1;2} = f_1 \circ g_2, \quad (14)$$

$$\sigma_{1;2} = \sigma_1 \circ \sigma_2, \quad (15)$$

and  $\text{dur}_{1,2}$  is defined as

$$\begin{aligned} \text{dur}_{1,2} : \Sigma_{\mathbf{A}} &\rightarrow \mathbf{T}, \\ \sigma_{\mathbf{A}} &\mapsto \text{dur}_1(\sigma_{\mathbf{A}}) \otimes_{\mathbf{T}} \text{dur}_2(\sigma_1(\sigma_{\mathbf{A}})). \end{aligned} \tag{16}$$

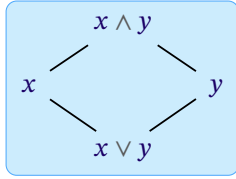
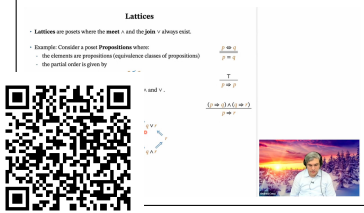
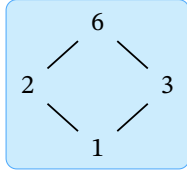
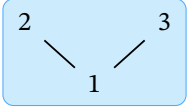
Watch: *Lattices* (4 minutes).

Figure 1.



(a) A lattice



(b) Not a lattice

Figure 2.: Examples of a lattice and a non-lattice.

$a$	$b$	$a \leq b$	$a \wedge b$	$a \vee b$
$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\perp$	$\perp$	$\perp$	$\top$
$\perp$	$\top$	$\top$	$\perp$	$\top$
$\perp$	$\perp$	$\top$	$\perp$	$\perp$

Table 27.1.: Properties of the **Bool**poset. Note that  $\leq \Leftrightarrow \Rightarrow$ .

## 27.3. Lattices

### Definition 27.8 (Lattice)

A *lattice* is a poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  with the additional property that, for any two-element subset  $\{x, y\} \subseteq \mathbf{P}$ , both the join  $\vee\{x, y\}$  and the meet  $\wedge\{x, y\}$  exist. Usually these are written using infix notation as  $x \vee y$  and  $x \wedge y$ , respectively.

### Definition 27.9 (Bounded lattices)

If both a top and a bottom exist, we call the lattice *bounded*, and denote it by  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}}, \vee, \wedge, \perp, \top \rangle$ .

**Example 27.10.** In Def. 9.12 we presented the poset arising from the power set  $\text{Pow } \mathbf{A}$  of a set  $\mathbf{A}$  and ordered via subset inclusion. This is a lattice, bounded by  $\mathbf{A}$  and by the empty set  $\emptyset$ . Note that this lattice possesses two (dual) monoidal structures  $\langle \text{Pow } \mathbf{A}, \subseteq, \emptyset, \cup \rangle$  and  $\langle \text{Pow } \mathbf{A}, \subseteq, \mathbf{A}, \cap \rangle$ .

**Example 27.11.** Consider the poset **Bool**, in which  $b_1 \leq_{\text{Bool}} b_2$  iff  $b_1 \Rightarrow b_2$ , that is, in addition to the operation

$$\Rightarrow : \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}, \quad (17)$$

called *implication*, there are also the familiar *and* ( $\wedge$ ) and *or* ( $\vee$ ) operations. Note that  $\wedge$  and  $\vee$  are commutative ( $b \wedge c = c \wedge b$ ,  $b \vee c = c \vee b$ ), whereas  $\Rightarrow$  is not. Furthermore,  $\wedge$  and  $\vee$  correspond to the meet and the join, respectively.

**Example 27.12.** Consider the set  $\{1, 2, 3, 6\}$  ordered by divisibility. For instance, since 2 divides 6, we have  $2 \leq 6$ . This is a lattice. However, the set  $\{1, 2, 3\}$  ordered by divisibility is not, since 2 and 3 lack a meet (Fig. 2).

**Lemma 27.13.**  $\mathbf{UP}$  is a bounded lattice (Def. 27.8) with

$$\begin{aligned} \leq_{\mathbf{UP}} &:= \supseteq, \\ \perp_{\mathbf{UP}} &:= \mathbf{P}, \\ \top_{\mathbf{UP}} &:= \emptyset, \\ \wedge_{\mathbf{UP}} &:= \cap, \\ \vee_{\mathbf{UP}} &:= \cup. \end{aligned} \quad (18)$$

*Proof.* Consider the poset  $\mathbf{UP} = \langle \mathbf{USets } \mathbf{P}, \leq_{\mathbf{UP}} \rangle$  and  $\mathbf{A}, \mathbf{B} \in \mathbf{USets } \mathbf{P}$ .

First, we need to show that  $\mathbf{A} \cap \mathbf{B} \in \mathbf{USets } \mathbf{P}$ . To this extent, we need to show that, for all  $a \in \mathbf{A} \cap \mathbf{B}$  and for all  $a \leq_{\mathbf{P}} b$ , it holds  $b \in \mathbf{A} \cap \mathbf{B}$ . We have  $\mathbf{A} \in \mathbf{USets } \mathbf{P}$  and  $\mathbf{B} \in \mathbf{USets } \mathbf{P}$ , meaning that by definition, if  $a \in \mathbf{A} \cap \mathbf{B}$ , we have  $a \in \mathbf{A} \wedge a \in \mathbf{B}$ . It follows that  $b \in \mathbf{USets } \mathbf{P}$  and  $b \in \mathbf{USets } \mathbf{Q}$ . Therefore,  $b \in \mathbf{USets } \mathbf{P} \cap \mathbf{B}$  and, thus,  $\mathbf{A} \cap \mathbf{B} \in \mathbf{USets } \mathbf{P}$ . Furthermore, we need to show that  $\mathbf{A} \cap \mathbf{B}$  is the least upper bound of  $\mathbf{A}$  and  $\mathbf{B}$ . Let  $\mathbf{C} \in \mathbf{USets } \mathbf{P}$  such that

$$\mathbf{A} \leq_{\mathbf{UP}} \mathbf{C} \leq_{\mathbf{UP}} (\mathbf{A} \cap \mathbf{B}) \iff \mathbf{A} \supseteq \mathbf{C} \supseteq (\mathbf{A} \cap \mathbf{B})$$

and

$$\mathbf{B} \leq_{\mathbf{UP}} \mathbf{C} \leq_{\mathbf{UP}} (\mathbf{A} \cap \mathbf{B}) \iff \mathbf{B} \supseteq \mathbf{C} \supseteq (\mathbf{A} \cap \mathbf{B}).$$

Using the fact that intersection preserves inclusions, we have

$$\begin{aligned} (\mathbf{A} \cap \mathbf{B}) &\supseteq (\mathbf{C} \cap \mathbf{C}) \supseteq (\mathbf{A} \cap \mathbf{B}) \\ \Leftrightarrow (\mathbf{A} \cap \mathbf{B}) &\supseteq \mathbf{C} \supseteq (\mathbf{A} \cap \mathbf{B}) \\ \Leftrightarrow \mathbf{C} &= (\mathbf{A} \cap \mathbf{B}). \end{aligned}$$

Therefore,  $\mathbf{A} \cap \mathbf{B}$  is the least upper bound of  $\mathbf{A}$  and  $\mathbf{B}$ .

Second, we need to show that  $\mathbf{A} \cup \mathbf{B} \in \mathbf{USets} \mathbf{P}$ , meaning that for all  $a \in \mathbf{A} \cup \mathbf{B}$ ,  $a \leq_{\mathbf{P}} b$  implies  $b \in \mathbf{A} \cup \mathbf{B}$ . We have  $\mathbf{A} \in \mathbf{USets} \mathbf{P}$  and  $\mathbf{B} \in \mathbf{USets} \mathbf{P}$ , meaning that by definition, if  $a \in \mathbf{A} \cup \mathbf{B}$ , we have either  $a \in \mathbf{A}$  or  $a \in \mathbf{B}$ . If  $a \in \mathbf{A}$ , then  $b \in \mathbf{USets} \mathbf{A}$ . If  $a \in \mathbf{B}$ , then  $b \in \mathbf{USets} \mathbf{B}$ . Either way,  $b \in \mathbf{A} \cup \mathbf{B}$  and, thus,  $\mathbf{A} \cup \mathbf{B} \in \mathbf{USets} \mathbf{P}$ . Furthermore, we need to show that  $\mathbf{A} \cup \mathbf{B}$  is the greatest lower bound of  $\mathbf{A}$  and  $\mathbf{B}$ . Let  $\mathbf{C} \in \mathbf{USets} \mathbf{P}$  such that

$$\mathbf{A} \cup \mathbf{B} \leq_{\mathbf{UP}} \mathbf{C} \leq_{\mathbf{UP}} \mathbf{A} \Leftrightarrow \mathbf{A} \cup \mathbf{B} \supseteq \mathbf{C} \supseteq \mathbf{A}$$

and

$$\mathbf{A} \cup \mathbf{B} \leq_{\mathbf{UP}} \mathbf{C} \leq_{\mathbf{UP}} \mathbf{B} \Leftrightarrow \mathbf{A} \cup \mathbf{B} \supseteq \mathbf{C} \supseteq \mathbf{B}.$$

Using the fact that union preserves inclusions, we have

$$\begin{aligned} (\mathbf{A} \cup \mathbf{B}) \cup (\mathbf{A} \cup \mathbf{B}) &\supseteq (\mathbf{C} \cup \mathbf{C}) \supseteq (\mathbf{A} \cup \mathbf{B}) \\ \Leftrightarrow \mathbf{A} \cup \mathbf{B} &\supseteq \mathbf{C} \supseteq (\mathbf{A} \cup \mathbf{B}) \\ \Leftrightarrow \mathbf{C} &= (\mathbf{A} \cup \mathbf{B}). \end{aligned}$$

Therefore,  $\mathbf{A} \cup \mathbf{B}$  is the greatest lower bound of  $\mathbf{A}$  and  $\mathbf{B}$ .

We have therefore proved that  $\mathbf{UP} = \langle \mathbf{USets} \mathbf{P}, \leq_{\mathbf{UP}} \rangle$  is a lattice. To show that it is bounded, we notice that  $\emptyset \subseteq \mathbf{C}$  for any  $\mathbf{C} \in \mathbf{USets} \mathbf{P}$ , meaning that  $\emptyset$  is the top. Furthermore, we notice that  $\mathbf{C} \subseteq \mathbf{P}$  for any  $\mathbf{C} \in \mathbf{USets} \mathbf{P}$ , meaning that  $\mathbf{P}$  is a bottom. Therefore, the lattice is bounded.  $\square$

**Lemma 27.14.**  $\mathbf{LP}$  is a bounded lattice (Def. 27.8) with:

$$\begin{aligned} \leq_{\mathbf{LP}} &:= \subseteq, \\ \perp_{\mathbf{LP}} &:= \emptyset, \\ \top_{\mathbf{LP}} &:= \mathbf{P}, \\ \wedge_{\mathbf{LP}} &:= \cup, \\ \vee_{\mathbf{LP}} &:= \cap. \end{aligned}$$

*Proof.* Consider the poset  $\mathbf{LP} = \langle \mathbf{LSets} \mathbf{P}, \leq_{\mathbf{LP}} \rangle$  and  $\mathbf{A}, \mathbf{B} \in \mathbf{LSets} \mathbf{P}$ .

First, we need to show that  $\mathbf{A} \cup \mathbf{B} \in \mathbf{LSets} \mathbf{P}$ . That is,  $b \leq_{\mathbf{P}} a$  implies  $b \in \mathbf{A} \cup \mathbf{B}$ . We have  $\mathbf{A} \in \mathbf{LSets} \mathbf{P}$  and  $\mathbf{B} \in \mathbf{LSets} \mathbf{P}$ , meaning that by definition, if  $a \in \mathbf{A} \cup \mathbf{B}$ , either  $a \in \mathbf{A}$  or  $a \in \mathbf{B}$ . If  $a \in \mathbf{A}$ , then  $b \in \mathbf{A}$ . If  $a \in \mathbf{B}$ , then  $b \in \mathbf{B}$ . It follows that  $b \in \mathbf{A} \cup \mathbf{B}$  and, thus, all  $\mathbf{A} \cup \mathbf{B} \in \mathbf{LSets} \mathbf{P}$ . Furthermore, we need to show that  $\mathbf{A} \cup \mathbf{B}$  is the least upper bound of  $\mathbf{A}$  and  $\mathbf{B}$ . Consider  $\mathbf{C} \in \mathbf{LSets} \mathbf{P}$  such that

$$\mathbf{A} \leq_{\mathbf{LP}} \mathbf{C} \leq_{\mathbf{LP}} \mathbf{A} \cup \mathbf{B} \Leftrightarrow \mathbf{A} \subseteq \mathbf{C} \subseteq \mathbf{A} \cup \mathbf{B}$$

and

$$\mathbf{B} \leq_{\mathbf{LP}} \mathbf{C} \leq_{\mathbf{LP}} \mathbf{A} \cup \mathbf{B} \Leftrightarrow \mathbf{B} \subseteq \mathbf{C} \subseteq \mathbf{A} \cup \mathbf{B}.$$

Using the fact that union preserves inclusions, we have

$$\begin{aligned} & (\mathbf{A} \cup \mathbf{B}) \subseteq (\mathbf{C} \cup \mathbf{C}) \subseteq (\mathbf{A} \cup \mathbf{B}) \\ \Leftrightarrow & (\mathbf{A} \cup \mathbf{B}) \subseteq \mathbf{C} \subseteq (\mathbf{A} \cup \mathbf{B}) \\ \Leftrightarrow & \mathbf{C} = (\mathbf{A} \cup \mathbf{B}). \end{aligned}$$

Therefore,  $\mathbf{A} \cup \mathbf{B}$  is the least upper bound of  $\mathbf{A}$  and  $\mathbf{B}$ .

Second, we need to show that  $\mathbf{A} \cap \mathbf{B} \in \mathbf{LSets} \mathbf{P}$ . That is,  $b \leq_{\mathbf{P}} a$  implies  $b \in \mathbf{A} \cap \mathbf{B}$ . We have  $\mathbf{A} \in \mathbf{LSets} \mathbf{P}$  and  $\mathbf{B} \in \mathbf{LSets} \mathbf{P}$ , meaning that by definition, if  $a \in \mathbf{A} \cap \mathbf{B}$ , we have  $a \in \mathbf{A} \wedge a \in \mathbf{B}$ . Since  $\mathbf{A}, \mathbf{B} \in \mathbf{LSets} \mathbf{P}$ , this implies  $b \in \mathbf{A} \wedge b \in \mathbf{B}$  and, thus,  $b \in \mathbf{A} \cap \mathbf{B}$ . Consider  $\mathbf{C} \in \mathbf{LSets} \mathbf{P}$  such that

$$\mathbf{A} \cap \mathbf{B} \leq_{LP} \mathbf{C} \leq_{LP} \mathbf{A} \Leftrightarrow \mathbf{A} \cap \mathbf{B} \subseteq \mathbf{C} \subseteq \mathbf{A}$$

and

$$\mathbf{A} \cap \mathbf{B} \leq_{LP} \mathbf{C} \leq_{LP} \mathbf{B} \Leftrightarrow \mathbf{A} \cap \mathbf{B} \subseteq \mathbf{C} \subseteq \mathbf{B}.$$

Using the fact that intersection preserves inclusions, we have

$$\begin{aligned} & (\mathbf{A} \cap \mathbf{B}) \cap (\mathbf{A} \cap \mathbf{B}) \subseteq (\mathbf{C} \cap \mathbf{C}) \subseteq (\mathbf{A} \cap \mathbf{B}) \\ \Leftrightarrow & \mathbf{A} \cap \mathbf{B} \subseteq \mathbf{C} \subseteq (\mathbf{A} \cap \mathbf{B}) \\ \Leftrightarrow & \mathbf{C} = (\mathbf{A} \cap \mathbf{B}). \end{aligned} \tag{19}$$

Therefore,  $\mathbf{A} \cap \mathbf{B}$  is the greatest lower bound of  $\mathbf{A}$  and  $\mathbf{B}$ .

We have therefore proved that  $LP = \langle \mathbf{LSets} \mathbf{P}, \leq_{LP} \rangle$  is a lattice. To show that it is bounded, we notice that  $\emptyset \subseteq \mathbf{C}$  for any  $\mathbf{C} \in \mathbf{LSets} \mathbf{P}$ , meaning that  $\emptyset$  is the bottom. Furthermore, we notice that  $\mathbf{C} \subseteq \mathbf{P}$  for any  $\mathbf{C} \in \mathbf{LSets} \mathbf{P}$ , meaning that  $\mathbf{P}$  is a top. Therefore, the lattice is bounded.  $\square$

### Graded exercise G.12 (UpperLowerBounds)

Let  $\mathbf{A} = \{a, b, c, d, e\}$ . Give examples of the following situations using Hasse diagrams. In each case, provide a poset structure on  $\mathbf{A}$  and a subset  $\mathbf{B} \subseteq \mathbf{A}$  such that:

1.  $\mathbf{B}$  has a least upper bound;
2.  $\mathbf{B}$  has a greatest lower bound;
3.  $\mathbf{B}$  has no least upper bound;
4.  $\mathbf{B}$  has no greatest lower bound.

## 27.4. Lattice homomorphisms

In this section, we want to abstract the concept of lattice and describe a category in which the objects are lattices themselves, and the morphisms are lattice homomorphisms. We call this category **Lat**.

### Definition 27.15 (Lattice homomorphism)

Given two lattices  $\mathbf{P}, \mathbf{Q}$ , a *lattice homomorphism* is a map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  which preserves meets and joins:

$$\begin{aligned} f(p \wedge_{\mathbf{P}} q) &= f(p) \wedge_{\mathbf{Q}} f(q), \\ f(p \vee_{\mathbf{P}} q) &= f(p) \vee_{\mathbf{Q}} f(q). \end{aligned} \quad (20)$$

**Example 27.16.** We consider the lattices  $\mathbf{P} = \langle \text{Pow}\{\text{🍷}, \text{🍺}\}, \cap, \cup \rangle$ , and  $\mathbf{Q} = \langle \{\text{🍺}, \text{🍷}\}, \max, \min \rangle$ , where min (max) refer to the minimum (maximum) alcoholic content of the beverage (assuming Swiss beers, which have alcohol content lower than wine). Furthermore, consider

$$\begin{aligned} f : \text{Pow}\{\text{🍷}, \text{🍺}\} &\rightarrow \{\text{🍷}, \text{🍺}\}, \\ \mathbf{A} &\mapsto \begin{cases} \text{🍺}, & \text{🍷} \in \mathbf{A}, \\ \text{🍷}, & \text{otherwise} \end{cases}. \end{aligned} \quad (21)$$

The explicit evaluations of  $f$  are reported in Table 27.2.

Is  $f$  a lattice homomorphism? Yes. We can check it explicitly. Consider  $\mathbf{A}, \mathbf{B} \subseteq \{\text{🍷}, \text{🍺}\}$ . We need to show that

$$f(\mathbf{A} \cap \mathbf{B}) = \max \{f(\mathbf{A}), f(\mathbf{B})\} \quad (22)$$

and

$$f(\mathbf{A} \cup \mathbf{B}) = \min \{f(\mathbf{A}), f(\mathbf{B})\}. \quad (23)$$

Technically, we can check every possible pair of  $\mathbf{A}, \mathbf{B}$  (only 16 for this case), but that's not efficient. First, consider  $f(\mathbf{A} \cap \mathbf{B}) = \text{🍷}$ . Following (21), this means  $\text{🍷} \notin \mathbf{A} \cap \mathbf{B}$  (in other words, either  $\text{🍷} \notin \mathbf{A}$ ,  $\text{🍷} \notin \mathbf{B}$ , or both). At least one of  $f(\mathbf{A})$  and  $f(\mathbf{B})$  is  $\text{🍷}$ , because

$$\frac{\text{🍷} \notin \mathbf{A}}{f(\mathbf{A}) = \text{🍷}} \quad \text{and} \quad \frac{\text{🍷} \notin \mathbf{B}}{f(\mathbf{B}) = \text{🍷}}. \quad (24)$$

This implies  $\max \{f(\mathbf{A}), f(\mathbf{B})\} = \text{🍷}$ , which verifies (22).

If instead, we have  $f(\mathbf{A} \cap \mathbf{B}) = \text{🍺}$ , then  $\text{🍷} \in \mathbf{A} \cap \mathbf{B}$ , meaning that  $\text{🍷} \in \mathbf{A}$  and  $\text{🍷} \in \mathbf{B}$ . Therefore,  $\max \{f(\mathbf{A}), f(\mathbf{B})\} = \text{🍺}$ , which verifies (22).

Condition (23) can be verified analogously.

The notion of lattice homomorphism can be extended to bounded lattices.

### Definition 27.17 (Bounded lattice homomorphism)

Given two bounded lattices  $\mathbf{P}, \mathbf{Q}$ , a *bounded lattice homomorphism* is a lattice homomorphism  $f : \mathbf{P} \rightarrow \mathbf{Q}$  which also preserves top and bottom:

$$\begin{aligned} f(\perp_{\mathbf{P}}) &= \perp_{\mathbf{Q}}, \\ f(\top_{\mathbf{P}}) &= \top_{\mathbf{Q}}. \end{aligned} \quad (25)$$

Note that (bounded) lattice homomorphisms are necessarily monotone.

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

$\mathbf{A}$	$f(\mathbf{A})$
$\emptyset$	🍷
$\{\text{🍷}\}$	🍺
$\{\text{🍺}\}$	🍷
$\{\text{🍷}, \text{🍺}\}$	🍺

Table 27.2.



## 27.5. Categories **Lat** and **BoundedLat**

### Definition 27.18 (Category **Lat**)

The category **Lat** is defined by:

1. *Objects*: The objects of this category are all lattices.
2. *Morphisms*: The morphisms from a lattice  $X$  to a lattice  $Y$  are the lattice homomorphisms from  $X$  to  $Y$ .
3. *Identity morphism*: The identity morphism for the lattice  $X$  is the identity map  $\text{id}_X$ .
4. *Composition operation*: The composition operation is composition of maps.

### Definition 27.19 (Category **BoundedLat**)

The category **BoundedLat** is defined by:

1. *Objects*: The objects of this category are all bounded lattices.
2. *Morphisms*: The morphisms from a lattice  $X$  to a lattice  $Y$  are the lattice homomorphisms from  $X$  to  $Y$ .
3. *Identity morphism*: The identity morphism for the bounded lattice  $X$  is the identity map  $\text{id}_X$ .
4. *Composition operation*: The composition operation is composition of maps.

**Exercise 43.** Show that **Lat** is a category.

*See solution on page 421.*

**Exercise 44.** Show that **BoundedLat** is a category.

*See solution on page 421.*



## 28. Lattice structure of DPs

In the previous chapter we have talked about the posetal structure of hom-sets. In **DP** hom-sets are also posets: morphisms can be ordered, and this order is preserved by composition. Moreover, it also has a lattice structure that is preserved by composition.

28.1 Ordering DPs . . . . .	406
28.2 Interaction with series composition . . . . .	407
28.3 Union of Design Problems . . . . .	408
28.4 Intersection of Design Problems . . . . .	409
28.5 Lattice structure of DP hom-sets . . . . .	410
28.6 Interaction with composition . . . . .	413

“Krampus” is a horned figure, which, in Alpine folklore, during the Christmas season, scares children who have misbehaved, helping Saint Nicholas. More recently, the character has been imported in Hollywood horror films, and has become part of the American popular culture.

## 28.1. Ordering DPs

### Definition 28.1 (Order on DP)

Suppose that  $\mathbf{P}$  and  $\mathbf{Q}$  are posets, and that  $\mathbf{d}, \mathbf{e} : \mathbf{P} \leftrightarrow \mathbf{Q}$  are design problems. We define the order as follows:

$$\frac{\mathbf{d} \leq_{\text{DP}} \mathbf{e}}{\mathbf{d}(p^*, q) \leq_{\text{Bool}} \mathbf{e}(p^*, q) \text{ for all } p \in \mathbf{P}, q \in \mathbf{Q}.}, \quad (1)$$

**Remark 28.2.** Recall that design problems are monotone functions, and note that the order defined in Def. 28.1 is just the usual order on monotone functions.

We diagrammatically represent the relation  $\mathbf{d} \leq_{\text{DP}} \mathbf{e}$  as in Fig. 1.



**Figure 1.:** The design problem  $\mathbf{d}$  implies the design problem  $\mathbf{e}$ .

## 28.2. Interaction with series composition

In the previous section, we introduced the concept of order in  $\mathbf{DP}$ , and proved that the hom-sets of  $\mathbf{DP}$  form a bounded lattice. In this section, we show that composition (Def. 26.7) of design problems is an order-preserving operation.

**Lemma 28.3.** Given  $\mathbf{d}, \mathbf{e} \in \text{Hom}_{\mathbf{DP}}(\mathbf{P}; \mathbf{Q})$  and  $\mathbf{g}, \mathbf{h} \in \text{Hom}_{\mathbf{DP}}(\mathbf{Q}; \mathbf{R})$  we have:

$$\frac{\mathbf{d} \leq_{\mathbf{DP}} \mathbf{e} \quad \mathbf{g} \leq_{\mathbf{DP}} \mathbf{h}}{(\mathbf{d} \circledast \mathbf{g}) \leq_{\mathbf{DP}} (\mathbf{e} \circledast \mathbf{h})}. \quad (2)$$

In other words, series composition is order-preserving on  $\mathbf{DP}$ .

*Proof.* We have

$$\begin{aligned} & (\mathbf{d} \circledast \mathbf{g})(p^*, r) \\ &= \bigvee_{q \in \mathbf{Q}} \mathbf{d}(p^*, q) \wedge \mathbf{g}(q, r) \\ &\leq_{\mathbf{DP}} \bigvee_{q \in \mathbf{Q}} \mathbf{e}(p^*, q) \wedge \mathbf{h}(q, r) \\ &= (\mathbf{e} \circledast \mathbf{h})(p^*, r). \end{aligned} \quad (3)$$

Therefore,  $\circledast$  is order-preserving on  $\mathbf{DP}$ .  $\square$



## 28.4. Intersection of Design Problems

Given two design problems  $\mathbf{d}, \mathbf{e} : \mathbf{P} \rightarrow \mathbf{Q}$ , we can define a design problem  $\mathbf{d} \wedge \mathbf{e}$  that is feasible if and only if  $\mathbf{d}$  and  $\mathbf{e}$  are both feasible. We call  $\mathbf{d} \wedge \mathbf{e}$  the *intersection* of  $\mathbf{d}$  and  $\mathbf{e}$ . One interpretation of  $\mathbf{d} \wedge \mathbf{e}$  is that  $\mathbf{d}$  and  $\mathbf{e}$  are two slightly different models of the same process, and we want to make sure that the design is conservatively feasible for both models.

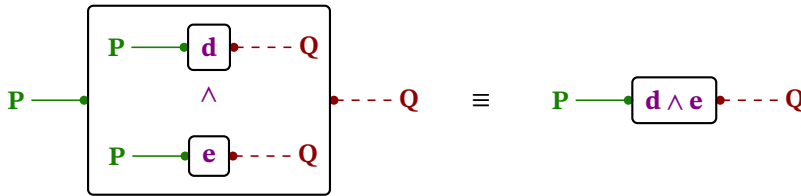
**Definition 28.5** (Intersection of design problems )

Given design problems  $\mathbf{d} : \mathbf{P} \rightarrow \mathbf{Q}$  and  $\mathbf{e} : \mathbf{P} \rightarrow \mathbf{Q}$ , their *intersection* is denoted  $(\mathbf{d} \wedge \mathbf{e}) : \mathbf{P} \rightarrow \mathbf{Q}$ , defined by:

$$(\mathbf{d} \wedge \mathbf{e}) : \mathbf{P}^{\text{op}} \times \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{Bool}, \quad (5)$$

$$\langle p^*, q \rangle \mapsto \mathbf{d}(p^*, q) \wedge \mathbf{e}(p^*, q).$$

The intersection of design problems is represented as in Fig. 3.



**Figure 3.:** Diagrammatic representation of the intersection of design problems.

We can directly generalize the intersection  $\mathbf{d} \wedge \mathbf{e}$  by allowing  $\mathbf{d}$  and  $\mathbf{e}$  to have different domain and codomains,  $\mathbf{d} : \mathbf{P} \rightarrow \mathbf{Q}$  and  $\mathbf{e} : \mathbf{R} \rightarrow \mathbf{S}$ . We call this putting two design problems “in parallel”.

## 28.5. Lattice structure of DP hom-sets

Given the definitions of  $\wedge$  and  $\vee$  in the previous sections, we can prove that every DP hom-sets have a lattice structure.

This lattice is bounded by a “true” and a “false” DP.

### Definition 28.6 (False and true DP)

Given any two partial orders  $\mathbf{P}, \mathbf{Q}$ , we can define a *false* DP as

$$\begin{aligned} \perp_{\mathbf{P}, \mathbf{Q}} : \mathbf{P}^{\text{op}} \times \mathbf{Q} &\rightarrow_{\text{Pos}} \mathbf{Bool}, \\ \langle p^*, q \rangle &\mapsto \perp. \end{aligned}$$

We can define a *true* DP as

$$\begin{aligned} \top_{\mathbf{P}, \mathbf{Q}} : \mathbf{P}^{\text{op}} \times \mathbf{Q} &\rightarrow_{\text{Pos}} \mathbf{Bool}, \\ \langle p^*, q \rangle &\mapsto \top. \end{aligned}$$

For any functionality-resource pair  $\mathbf{P}, \mathbf{Q}$ , these represent the design problem which is never (respectively always) feasible.

**Lemma 28.7.**  $\text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$  is a bounded lattice with union  $\vee$  as join, intersection  $\wedge$  as meet, top  $\top_{\mathbf{P}, \mathbf{Q}}$  and bottom  $\perp_{\mathbf{P}, \mathbf{Q}}$ .

*Proof.* First, we need to prove that  $\text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$  is a poset. To prove this, we check the following:

▷ *Reflexivity:* Given  $\mathbf{d} \in \text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$ :

$$\frac{\top}{\mathbf{d} \leq_{\text{DP}} \mathbf{d}}; \quad (6)$$

▷ *Antisymmetry:* Given  $\mathbf{d}, \mathbf{e} \in \text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$ :

$$\frac{\mathbf{d} \leq_{\text{DP}} \mathbf{e} \quad \mathbf{e} \leq_{\text{DP}} \mathbf{d}}{\mathbf{d} = \mathbf{e}}; \quad (7)$$

▷ *Transitivity:* Given  $\mathbf{d}, \mathbf{e}, \mathbf{g} \in \text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$ :

$$\frac{\mathbf{d} \leq_{\text{DP}} \mathbf{e} \quad \mathbf{e} \leq_{\text{DP}} \mathbf{g}}{\mathbf{d} \leq_{\text{DP}} \mathbf{g}}. \quad (8)$$

Therefore,  $\text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$  is a poset. Furthermore, consider two design problems  $\mathbf{d}, \mathbf{e} \in \text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$ . Their greatest lower bound (meet) is  $\mathbf{d} \wedge \mathbf{e}$ , since it is the greatest design problem implying both  $\mathbf{d}$  and  $\mathbf{e}$ . Their least upper bound (join), instead, is  $\mathbf{d} \vee \mathbf{e}$ , since it is the least design problem implied by both  $\mathbf{d}$  and  $\mathbf{e}$ . This proves that  $\text{Hom}_{\text{DP}}$  is a lattice. To prove that it is bounded, we identify the top element as  $\top_{\mathbf{P}, \mathbf{Q}}$  (it is implied by all other design problems) and the bottom element as  $\perp_{\mathbf{P}, \mathbf{Q}}$  (it implies by all the other design problems).  $\square$

We show that a DP hom-set is a *complete lattice*.

### Definition 28.8 (Complete Lattice)

A poset  $\mathbf{P} = \langle \mathbf{P}, \leq_{\mathbf{P}} \rangle$  is a *complete lattice* if every subset  $\mathbf{S}$  of  $\mathbf{P}$  has both a *greatest lower bound* (often referred to as the *infimum*, *meet*) and a *least upper*

*bound* (often referred to as the *supremum*, *join*).

**Example 28.9.** Consider the power set of any given set, ordered by inclusion. The supremum of any two subsets is given by their union. The infimum of any two subsets is given by their intersection.

**Lemma 28.10** (**DP** hom-sets are complete lattices). Hom-sets of **DP** are complete lattices.

*Proof.* Consider any  $\mathbf{P}, \mathbf{Q} \in \mathbf{Ob}_{\mathbf{DP}}$  and  $\mathbf{Hom}_{\mathbf{DP}}(\mathbf{P}; \mathbf{Q})$ . We have already shown that  $\mathbf{A} = \mathbf{Hom}_{\mathbf{DP}}(\mathbf{P}; \mathbf{Q})$  is a bounded lattice (Lemma 28.7). Now, take any subset  $\mathbf{B}$  of  $\mathbf{A}$ . We define the following two design problems:

$$\begin{aligned} \bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d} : \mathbf{P}^{\text{op}} \times \mathbf{Q} &\rightarrow_{\mathbf{Pos}} \mathbf{Bool}, \\ \langle p, q \rangle &\mapsto \exists \mathbf{d} \in \mathbf{B} : \mathbf{d}(p^*, q), \end{aligned} \quad (9)$$

and

$$\begin{aligned} \bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d} : \mathbf{P}^{\text{op}} \times \mathbf{Q} &\rightarrow_{\mathbf{Pos}} \mathbf{Bool}, \\ \langle p^*, q \rangle &\mapsto \forall \mathbf{d} \in \mathbf{B} : \mathbf{d}(p^*, q). \end{aligned} \quad (10)$$

These are clearly design problems (given that  $\mathbf{d}$  is a design problem) and given their signature they belong to  $\mathbf{A}$ . We will now argue that  $\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is the supremum of  $\mathbf{B}$  and  $\bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is the infimum of  $\mathbf{B}$ .

**$\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is the supremum of  $\mathbf{B}$ :** First, for any  $\mathbf{d} \in \mathbf{B}$ , we know that  $\mathbf{d} \leq_{\mathbf{DP}} \bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$ , proving that  $\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is an upper bound of  $\mathbf{B}$ . We now want to show that  $\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is the least upper bound of  $\mathbf{B}$ : for any upper bound  $\mathbf{e}$  of  $\mathbf{B}$ , we need to show  $\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d} \leq_{\mathbf{DP}} \mathbf{e}$ . In other words, for any pair  $\langle p^*, q \rangle \in \mathbf{P}^{\text{op}} \times \mathbf{Q}$ , we need to show  $(\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d})(p^*, q) \leq_{\mathbf{Bool}} \mathbf{e}(p^*, q)$ . Fix any  $\langle p^*, q \rangle$ . If  $(\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d})(p^*, q) = \perp$ , the condition is trivially satisfied.

If, instead,  $(\bigvee_{\mathbf{d} \in \mathbf{B}} \mathbf{d})(p^*, q) = \top$ , there exists a  $\mathbf{d} \in \mathbf{B}$  such that  $\mathbf{d}(p^*, q) = \top$ . Given that  $\mathbf{e}$  is an upper bound of  $\mathbf{B}$ , this implies  $\top = \mathbf{d}(p^*, q) \leq_{\mathbf{Bool}} \mathbf{e}(p^*, q) = \top$ , proving the condition.

**$\bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is the infimum of  $\mathbf{B}$ :** First, for any  $\mathbf{d} \in \mathbf{B}$ , we know that  $\mathbf{d} \wedge \bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d} = \bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d} \leq_{\mathbf{DP}} \mathbf{d}$ , proving that  $\bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is a lower bound of  $\mathbf{B}$ . We now want to show that  $\bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$  is the greatest lower bound of  $\mathbf{B}$ : for any lower bound  $\mathbf{e}$  of  $\mathbf{B}$ , we need to show  $\mathbf{e} \leq_{\mathbf{DP}} \bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d}$ . In other words, for any pair  $\langle p^*, q \rangle \in \mathbf{P}^{\text{op}} \times \mathbf{Q}$ , we need to show  $\mathbf{e}(p^*, q) \leq_{\mathbf{Bool}} (\bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d})(p^*, q)$ . Fix any  $\langle p^*, q \rangle$ . If  $(\bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d})(p^*, q) = \top$ , the condition is trivially satisfied. If, instead,  $(\bigwedge_{\mathbf{d} \in \mathbf{B}} \mathbf{d})(p^*, q) = \perp$ , there is at least one  $\mathbf{d} \in \mathbf{B}$  for which  $\mathbf{d}(p^*, q) = \perp$ . Given that  $\mathbf{e}$  is a lower bound of  $\mathbf{B}$ , this implies  $\perp = \mathbf{e}(p^*, q) \leq_{\mathbf{Bool}} \mathbf{d}(p^*, q) = \perp$ , proving the condition.  $\square$

**Definition 28.11** (Distributive Lattice)

A lattice  $\mathbf{P} = \langle \mathbf{P}, \wedge, \vee \rangle$  is a *distributive lattice* if for all  $x, y, z \in \mathbf{P}$ :

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z). \quad (11)$$

**Remark 28.12.** Note that condition (11) is equivalent to its dual:

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z), \quad (12)$$



for all  $x, y, z \in \mathbf{P}$ .

**Lemma 28.13.** Consider  $\mathbf{d}, \mathbf{e}, \mathbf{g} \in \text{Hom}_{\mathbf{DP}}(\mathbf{P}; \mathbf{Q})$ . We have

$$(\mathbf{d} \wedge \mathbf{e}) \vee \mathbf{g} = (\mathbf{d} \vee \mathbf{g}) \wedge (\mathbf{e} \vee \mathbf{g}). \quad (13)$$

*Proof.* We have:

$$\begin{aligned} & ((\mathbf{d} \wedge \mathbf{e}) \vee \mathbf{g})(p^*, q) \\ &= (\mathbf{d} \wedge \mathbf{e})(p^*, q) \vee \mathbf{g}(p^*, q) \\ &= (\mathbf{d}(p^*, q) \wedge \mathbf{e}(p^*, q)) \vee \mathbf{g}(p^*, q) \\ &= (\mathbf{d}(p^*, q) \vee \mathbf{g}(p^*, q)) \wedge (\mathbf{e}(p^*, q) \vee \mathbf{g}(p^*, q)) \\ &= ((\mathbf{d} \vee \mathbf{g}) \wedge (\mathbf{e} \vee \mathbf{g}))(p^*, q). \end{aligned} \quad (14)$$

□

**Lemma 28.14.** Consider  $\mathbf{d}, \mathbf{e}, \mathbf{g} \in \text{Hom}_{\mathbf{DP}}(\mathbf{P}; \mathbf{Q})$ . We have

$$(\mathbf{d} \vee \mathbf{e}) \wedge \mathbf{g} = (\mathbf{d} \wedge \mathbf{g}) \vee (\mathbf{e} \wedge \mathbf{g}). \quad (15)$$

*Proof.* We have:

$$\begin{aligned} & ((\mathbf{d} \vee \mathbf{e}) \wedge \mathbf{g})(p^*, q) \\ &= (\mathbf{d} \vee \mathbf{e})(p^*, q) \wedge \mathbf{g}(p^*, q) \\ &= (\mathbf{d}(p^*, q) \vee \mathbf{e}(p^*, q)) \wedge \mathbf{g}(p^*, q) \\ &= (\mathbf{d}(p^*, q) \wedge \mathbf{g}(p^*, q)) \vee (\mathbf{e}(p^*, q) \wedge \mathbf{g}(p^*, q)) \\ &= ((\mathbf{d} \wedge \mathbf{g}) \vee (\mathbf{e} \wedge \mathbf{g}))(p^*, q). \end{aligned} \quad (16)$$

□

**Lemma 28.15** (**DP** hom-sets are distributive lattices). Hom-sets of **DP** are distributive lattices.

*Proof.* Either Lemma 28.13 or Lemma 28.14 prove the statement. □

## 28.6. Interaction with composition

Furthermore, we show that all composition operations preserve joins, and all composition operations except trace preserve meets.

### Series composition

**Lemma 28.16.** Consider  $\mathbf{d}, \mathbf{e} \in \text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$  and  $\mathbf{g} \in \text{Hom}_{\text{DP}}(\mathbf{Q}; \mathbf{R})$ . We have

$$(\mathbf{d} \vee \mathbf{e}) \circledast \mathbf{g} = (\mathbf{d} \circledast \mathbf{g}) \vee (\mathbf{e} \circledast \mathbf{g}). \quad (17)$$

This is diagrammatically represented in Fig. 4.



Figure 4.

*Proof.* We have:

$$\begin{aligned} & ((\mathbf{d} \vee \mathbf{e}) \circledast \mathbf{g})(p^*, r) \\ &= \bigvee_{q \in Q} (\mathbf{d} \vee \mathbf{e})(p^*, q) \wedge \mathbf{g}(q^*, r) \\ &= \bigvee_{q \in Q} (\mathbf{d}(p^*, q) \vee \mathbf{e}(p^*, q)) \wedge \mathbf{g}(q^*, r) \\ &= \bigvee_{q \in Q} (\mathbf{d}(p^*, q) \wedge \mathbf{g}(q^*, r)) \vee (\mathbf{e}(p^*, q) \wedge \mathbf{g}(q^*, r)) \\ &= ((\mathbf{d} \circledast \mathbf{g}) \vee (\mathbf{e} \circledast \mathbf{g}))(p^*, r). \end{aligned} \quad (18)$$

□

**Remark 28.17.** Consider  $\mathbf{d}, \mathbf{e} \in \text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$  and  $\mathbf{g}, \mathbf{h} \in \text{Hom}_{\text{DP}}(\mathbf{Q}; \mathbf{R})$ . In general, we have:

$$(\mathbf{d} \vee \mathbf{e}) \circledast (\mathbf{g} \vee \mathbf{h}) \neq (\mathbf{d} \circledast \mathbf{g}) \vee (\mathbf{e} \circledast \mathbf{h}). \quad (19)$$

Indeed, consider  $\mathbf{d} = \top_{\mathbf{P}, \mathbf{Q}}$ ,  $\mathbf{e} = \perp_{\mathbf{P}, \mathbf{Q}}$ ,  $\mathbf{g} = \perp_{\mathbf{Q}, \mathbf{R}}$ , and  $\mathbf{h} = \top_{\mathbf{Q}, \mathbf{R}}$ . Clearly:

$$\begin{aligned} ((\mathbf{d} \vee \mathbf{e}) \circledast (\mathbf{g} \vee \mathbf{h}))(p^*, r) &= \bigvee_{q \in Q} (\mathbf{d} \vee \mathbf{e})(p^*, r) \wedge (\mathbf{g} \vee \mathbf{h})(q^*, r) \\ &= \top, \end{aligned} \quad (20)$$

but

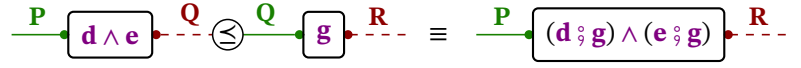
$$\begin{aligned} & ((\mathbf{d} \circledast \mathbf{g}) \vee (\mathbf{e} \circledast \mathbf{h}))(p^*, r) \\ &= \left( \bigvee_{q \in Q} \mathbf{d}(p^*, q) \wedge \mathbf{g}(q^*, r) \right) \vee \left( \bigvee_{q \in Q} \mathbf{e}(p^*, q) \wedge \mathbf{h}(q^*, r) \right) \\ &= \perp \vee \perp \\ &= \perp. \end{aligned} \quad (21)$$

**Lemma 28.18.** Consider  $\mathbf{d}, \mathbf{e} \in \text{Hom}_{\text{DP}}(\mathbf{P}; \mathbf{Q})$  and  $\mathbf{g} \in \text{Hom}_{\text{DP}}(\mathbf{Q}; \mathbf{R})$ . We have

$$(\mathbf{d} \wedge \mathbf{e}) \circledast \mathbf{g} = (\mathbf{d} \circledast \mathbf{g}) \wedge (\mathbf{e} \circledast \mathbf{g}). \quad (22)$$

This is diagrammatically represented in Fig. 5.

Figure 5.



*Proof.* We have:

$$\begin{aligned}
 & ((d \wedge e) \circ g)(p^*, r) \\
 &= \bigvee_{q \in Q} (d \wedge e)(p^*, q) \wedge g(q^*, r) \\
 &= \bigvee_{q \in Q} (d(p^*, q) \wedge e(p^*, q)) \wedge g(q^*, r) \\
 &= \bigvee_{q \in Q} (d(p^*, q) \wedge g(q^*, r)) \wedge (e(p^*, q) \wedge g(q^*, r)) \\
 &= ((d \circ g) \wedge (e \circ g))(p^*, r).
 \end{aligned} \tag{23}$$

□



## 29. Constructing design problems

We have previously seen ways to construct posets (Chapter 10) and categories (Chapter 17) from simpler pieces. Analogously, this chapter describes some ways to construct design problems.

<b>29.1 Companion and conjoint use . .</b>	<b>416</b>
<b>29.2 Companions and conjoint . . . .</b>	<b>418</b>
<b>29.3 Monoidal DPs . . . . .</b>	<b>420</b>

## 29.1. Union and intersection with companions and conjoins

We can also re-define the sum  $\vee$  and intersection  $\wedge$  using companions and conjoins, which allows us to introduce some useful constructions.

### Definition 29.1 (Diagonal function)

Define the *diagonal function*  $\text{diag}_P : P \rightarrow P \times P$ :

$$\begin{aligned} \text{diag}_P : P &\rightarrow P \times P, \\ p &\mapsto \langle p, p \rangle. \end{aligned} \tag{1}$$

### Definition 29.2 (Codiagonal function)

Define the *codiagonal function*  $\text{codiag}_P : P + P \rightarrow P$ :

$$\begin{aligned} \text{codiag}_P : P + P &\rightarrow P, \\ \langle 1, p \rangle &\mapsto p, \\ \langle 2, p \rangle &\mapsto p. \end{aligned} \tag{2}$$

Using the diagonal function, (4) can be rewritten as the following lemma.

**Lemma 29.3.** Given  $d, e : P \rightarrow Q$ , we have:

$$d \vee e = \text{conj}(\text{diag}_P) \circ (d + e) \circ \text{comp}(\text{diag}_Q). \tag{3}$$

*Proof.* First, note that

$$\begin{aligned} \text{conj}(\text{diag}_P) : P &\rightarrow P + P \\ \langle p_1^*, \langle 1, p_2 \rangle \rangle &\mapsto p_1 \leq p_2 \\ \langle p_1^*, \langle 1, p_3 \rangle \rangle &\mapsto p_1 \leq p_3 \end{aligned} \tag{4}$$

and

$$\begin{aligned} \text{comp}(\text{diag}_Q) : Q + Q &\rightarrow Q \\ \langle \langle 1, q_1^*, q_3 \rangle \rangle &\mapsto q_1 \leq q_3 \\ \langle \langle 2, q_2^*, q_3 \rangle \rangle &\mapsto q_2 \leq q_3. \end{aligned} \tag{5}$$

We start by looking at  $\underbrace{\text{conj}(\text{diag}_P) \circ (d + e)}_{\star} : P \rightarrow Q + Q$ .

$$\begin{aligned} &\star (\langle p^*, q \rangle) \\ &= \bigvee_{p' \in P+P} \text{conj}(\text{diag}_P)(\langle p^*, p' \rangle) \wedge (d + e)(\langle p^*, q \rangle) \\ &= \left( \bigvee_{\langle 1, p' \rangle \in P+P} (p \leq p') \wedge d(p^*, q) \right) \vee \left( \bigvee_{\langle 2, p' \rangle \in P+P} (p \leq p') \wedge e(p^*, q) \right) \\ &= d(p^*, q) \vee e(p^*, q). \end{aligned} \tag{6}$$

We now look at  $\star \circ \text{comp}(\text{diag}_Q) : \mathbf{P} \rightarrow \mathbf{Q}$ :

$$\begin{aligned}
 & (\star \circ \text{comp}(\text{diag}_Q))(p^*, q') \\
 &= \bigvee_{q \in Q+Q} \star(p^*, q) \wedge \text{comp}(\text{diag}_Q)(q, q') \\
 &= \left( \bigvee_{\langle 1, q \rangle \in Q+Q} \mathbf{d}(p^*, q) \wedge (q \leq q') \right) \vee \left( \bigvee_{\langle 2, q \rangle \in Q+Q} \mathbf{e}(p^*, q) \wedge (q \leq q') \right) \quad (7) \\
 &= \mathbf{d}(p^*, q') \vee \mathbf{e}(p^*, q').
 \end{aligned}$$

□

## 29.2. Companions and conjoint

We round out our discussion of **DP** by introducing two formulas for transforming monotone maps in **Pos** into design problems in **DP**. Each monotone map  $f$  can be transformed into two design problems, called its *companion*  $\text{comp}(f)$  and *conjoint*  $\text{conj}(f)$ . Many of the design problems that we have introduced can be realized as companions and conjoints of appropriate monotone maps.

### Definition 29.4 (Companion and conjoint)

Let  $\mathbf{P}$  and  $\mathbf{Q}$  be posets, and suppose that  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$  is a monotone map. We define its *companion* in **DP**, denoted  $\text{comp}(f) : \mathbf{P} \leftrightarrow \mathbf{Q}$ , and its *conjoint*, denoted  $\text{conj}(f) : \mathbf{Q} \leftrightarrow \mathbf{P}$  as

$$\text{comp}(f)(p^*, q) := f(p) \leq_Q q \quad \text{and} \quad \text{conj}(f)(q^*, p) := q \leq_P f(p). \quad (8)$$

**Lemma 29.5.** Both the companion and conjoint constructions from Def. 29.4 are functorial from **Pos** to **DP**: they preserve identities and composition.

*Proof.* We will show that the companion and conjoint are functors of the following forms:

$$\text{comp} : \mathbf{Pos} \rightarrow \mathbf{DP} \quad \text{and} \quad \text{conj} : \mathbf{Pos} \rightarrow \mathbf{DP}^{\text{op}}. \quad (9)$$

First, we see that they send the identity monotone maps  $\text{id}_{\mathbf{P}}$  to the identity design problem  $\text{id}_{\mathbf{P}}$  for any poset  $\mathbf{P}$ , because

$$\text{comp}(\text{id}_{\mathbf{P}})(p_1^*, p_2) = (p_1 \leq_P p_2) = \text{conj}(\text{id}_{\mathbf{P}})(p_1^*, p_2). \quad (10)$$

Now suppose that  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow_{\mathbf{Pos}} \mathbf{R}$  are given. We first show that  $\text{conj}(g) \circ \text{conj}(f) = \text{conj}(f \circ g)$ . For any  $p \in \mathbf{P}$  and  $r \in \mathbf{R}$ , we have

$$\begin{aligned} & (\text{conj}(g) \circ \text{conj}(f))(p^*, r) \\ &= \bigvee_{q \in \mathbf{Q}} \text{conj}(g)(r^*, q) \wedge \text{conj}(f)(q^*, p) \\ &= \bigvee_{q \in \mathbf{Q}} (r \leq_{\mathbf{R}} g(q)) \wedge (q \leq_{\mathbf{Q}} f(p)) \\ &= r \leq_{\mathbf{R}} g(f(p)) \\ &= (\text{conj}(f \circ g))(r^*, p). \end{aligned} \quad (11)$$

Similarly, we can prove that  $\text{comp}(f) \circ \text{comp}(g) = \text{comp}(f \circ g)$ :

$$\begin{aligned} & (\text{comp}(f) \circ \text{comp}(g))(p^*, r) \\ &= \bigvee_{q \in \mathbf{Q}} \text{comp}(f)(p^*, q) \wedge \text{comp}(g)(q^*, r) \\ &= \bigvee_{q \in \mathbf{Q}} (g(p) \leq_{\mathbf{Q}} q) \wedge (g(q) \leq_{\mathbf{R}} r) \\ &= g(f(p)) \leq_{\mathbf{R}} r \\ &= (\text{comp}(f \circ g))(p^*, r). \end{aligned} \quad (12)$$

□

**Example 29.6.** The identity design problem  $\text{id}_{\mathbf{P}} : \mathbf{P} \leftrightarrow \mathbf{P}$  is the companion (and

the conjoint) of the identity map  $\text{id}_{\mathbf{P}}^{\text{Pos}} : \mathbf{P} \rightarrow_{\text{Pos}} \mathbf{P}$ . This is easy to check, as

$$\begin{aligned}
 & \text{comp}(\text{id}_{\mathbf{P}})(p_1^*, p_2) \\
 &= \text{id}_{\mathbf{P}}(p_1) \leq p_2 \\
 &= p_1 \leq p_2 \\
 &= \text{id}_{\mathbf{P}}(p_1^*, p_2).
 \end{aligned} \tag{13}$$

**Example 29.7.** The coproduct injections  $\iota_{\mathbf{P}}, \iota_{\mathbf{Q}}$  for design problems are the companions of the coproduct injections for the disjoint union.

**Example 29.8.** The product projections  $\pi_{\mathbf{P}}, \pi_{\mathbf{Q}}$  for design problems are the conjoints of the coproduct injections for the disjoint union.

## Deriving terminators

Using companion and conjoint we can obtain the equivalent of “terminators” representing constant functionality/resources. Consider an element  $x$  of a poset  $\mathbf{P}$ . We can represent this constant element as a map  $f_x$  from the singleton to the poset:

$$\begin{aligned}
 f_x : \mathbf{1} &\rightarrow_{\text{Pos}} \mathbf{P}, \\
 \bullet &\mapsto x.
 \end{aligned} \tag{14}$$

By taking the companion of  $f_x$  we get

$$\begin{aligned}
 \text{comp}(f_x) : \mathbf{1} &\rightarrow \mathbf{P}, \\
 \langle \bullet, p \rangle &\mapsto (x \leq_{\mathbf{P}} p).
 \end{aligned} \tag{15}$$

By taking the conjoint, we get

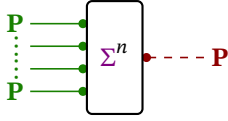
$$\begin{aligned}
 \text{conj}(f_x) : \mathbf{P} &\rightarrow \mathbf{1}, \\
 \langle p, \bullet \rangle &\mapsto (p \leq_{\mathbf{P}} x).
 \end{aligned} \tag{16}$$

These two cases represent design problems with either *constant* resources or constant, functionalities, respectively.



### 29.3. Monoidal DPs

If the underlying posets of functionality and resources are monoidal (Def. 27.1), then we can define some canonical DPs.

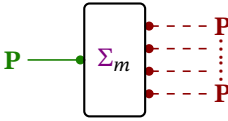


**Definition 29.9** (Sum of resources for monoidal posets)

If the poset  $\mathbf{P}$  is monoidal with monoidal product  $\otimes$ , then the “sum” of  $n$  copies of  $\mathbf{P}$  is a design problem given by

$$\begin{aligned} \Sigma^n : (\mathbf{P}^n)^{\text{op}} \times \mathbf{P} &\xrightarrow{\text{Pos}} \mathbf{Bool}, \\ \langle \langle p_1, \dots, p_n \rangle^*, q \rangle &\longmapsto (p_1 \otimes \dots \otimes p_n \leq_{\mathbf{P}} q). \end{aligned} \quad (17)$$

We can do the symmetric construction.



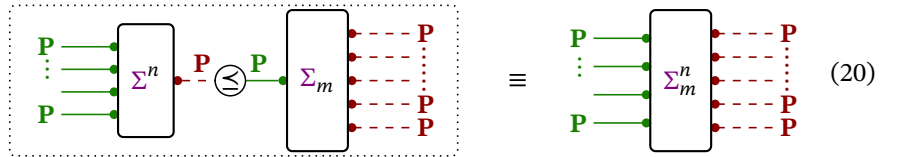
**Definition 29.10** (Sum of functionalities for monoidal posets)

If the poset  $\mathbf{P}$  is monoidal with monoidal product  $\otimes$ , then the “sum” of  $m$  copies of  $\mathbf{P}$  is a design problem given by

$$\begin{aligned} \Sigma_m : \mathbf{P}^{\text{op}} \times (\mathbf{P}^m) &\xrightarrow{\text{Pos}} \mathbf{Bool}, \\ \langle p^*, \langle q_1, \dots, q_m \rangle \rangle &\longmapsto p \leq_{\mathbf{P}} (q_1 \otimes \dots \otimes q_m). \end{aligned} \quad (18)$$

We can now put these in series to obtain the generic DP  $\Sigma_m^n$  with  $n$  functionalities and  $m$  resources:

$$\Sigma_m^n = \Sigma^n \circ \Sigma_m. \quad (19)$$



Note that this works with addition, but also with other associative operations, such as multiplication, max, etc.

## Solutions to selected exercises

**Solution of Exercise 40.** Assume that  $y$  and  $z$  are both least upper bounds of  $S \subseteq P$ . In other words, one knows  $x \leq_P y$  and  $x \leq_P z$  for all  $x \in S$ . However, one also has  $y \leq_P z$  and  $z \leq_P y$  (from  $y, z$  assumed to be both least upper bounds). Because of antisymmetry, this implies  $y = z$  and proves the uniqueness of least upper bounds in a poset.

**Solution of Exercise 41.** In Fig. 1 you find an example of a subset  $S$  of a poset  $P$  which has incomparable lower bounds. In Fig. 2 instead, there is a greatest lower bound.

**Solution of Exercise 42.** Consider the posets  $\langle \text{Pow } P, \subseteq \rangle$  and  $\langle \text{LSets } P, \subseteq \rangle$ , and let  $S_1, S_2 \in \text{Pow } P$ . It is clear that given  $S_1 \subseteq S_2$ , we have

$$\{y \in P \mid \exists x \in S_1 : y \leq_P x\} \subseteq \{y \in P \mid \exists x \in S_2 : y \leq_P x\}. \quad (21)$$

Therefore,  $\downarrow S_1 \subseteq \downarrow S_2$ , satisfying the monotonicity property for  $\downarrow$ .

**Solution of Exercise 43.** Clearly, given any lattice  $X$ , the identity map  $\text{id}_X$  is a lattice homomorphism, since

$$\begin{aligned} \text{id}_X(x_1 \wedge_X x_2) &= x_1 \wedge_X x_2 \\ \text{id}_X(x_1 \vee_X x_2) &= x_1 \vee_X x_2. \end{aligned} \quad (22)$$

This said, the identity map satisfies unitality. Now, given lattice homomorphisms

$$f : X \rightarrow Y \quad \text{and} \quad g : Y \rightarrow Z, \quad (23)$$

their composition is a lattice homomorphism, since

$$\begin{aligned} (f \circ g)(x_1 \wedge_X x_2) &= f(x_1 \wedge_X x_2) \circ g \\ &= g(f(x_1) \wedge_Y f(x_2)) \\ &= (f \circ g)(x_1) \wedge_Z (f \circ g)(x_2), \end{aligned} \quad (24)$$

and

$$\begin{aligned} (f \circ g)(x_1 \vee_X x_2) &= f(x_1 \vee_X x_2) \circ g \\ &= g(f(x_1) \vee_Y f(x_2)) \\ &= (f \circ g)(x_1) \vee_Z (f \circ g)(x_2), \end{aligned} \quad (25)$$

We have already checked in the past the map composition is associative (e.g., when checking that **Set** and **Pos** are categories).

**Solution of Exercise 44.** Consider the solution of Exercise 43 as a starting point. Clearly, given any lattice  $X$ , the identity map  $\text{id}_X$  is also a bounded lattice homomorphism, since

$$\begin{aligned} \text{id}_X(\perp_X) &= \perp_X \\ \text{id}_X(\top_X) &= \top_X. \end{aligned} \quad (26)$$

This said, the identity map satisfies unitality. Now, given lattice homomorphisms  $f : X \rightarrow Y$  and  $g : Y \rightarrow Z$ , their composition is also a bounded lattice homomorphism, since

$$\begin{aligned} (f \circ g)(\perp_X) &= g(\perp_Y) \\ &= \perp_Z, \end{aligned} \quad (27)$$

and

$$\begin{aligned} (f \circ g)(\top_X) &= g(\top_Y) \\ &= \top_Z. \end{aligned} \quad (28)$$

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

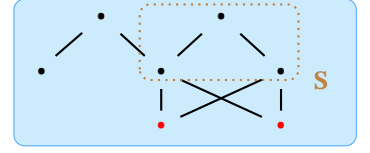


Figure 1.: Example of lower bounds of  $S$ .

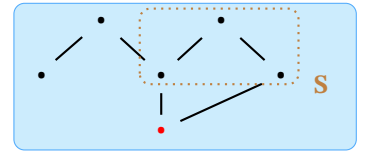


Figure 2.: Example of lower bounds and greatest lower bounds of  $S$ .

We have already checked in the past the map composition is associative (e.g., when checking that **Set** and **Pos** are categories).

# PART H.INTERCONNECTIONS



---

## 30. Parallel composition

425

## 31. Crossing wires

449

---

The *Sechseläuten* is a traditional spring holiday in the Zurich, Switzerland, usually happening on the 3rd monday of April. The old city guilds meet in the city center for a parade, climax of which is the burning of the “Böögg”, a snowman prepared with explosives, considered a weather oracle for the summer.





# 30. Parallel composition

30.1 Modeling parallelism . . . . .	426
30.2 Stacking semicategories . . . . .	429
30.3 Associative stacking (semi)categories . . . . .	431
30.4 Examples of associative stacking semicategories . . . . .	435
30.5 Functorial stacking semicategories . . . . .	439
30.6 Functorial stacking categories . . . . .	444
30.7 Monoidal stacking (semi)categories . . . . .	445
30.8 Strict monoidal (semi)categories . . . . .	448

*Raclette* is a Swiss dish, based on heating cheese and scraping off the melted part. In Switzerland, raclette is typically served with potatoes, cornichons, pickled onions, and Fendant wine.



## 30.1. Modeling parallelism

We have talked a lot about composition, and considered many examples. However, the types of compositions we studied were, so far, mostly of the “in series composition” kind. For instance, we considered the series composition of travel routes (Example 15.1) and trekking routes (Section 15.2), functions (Section 4.10) and relations (Section 5.2), engineering component dependencies (Section 15.4) and Moore machines (Section 16.1), *etc.*

In this chapter, we will consider composition both in series and *in parallel*. For example, given functions  $f : A \rightarrow B$  and  $g : B \rightarrow C$ , because the target set of the function  $f$  matches the source set of  $g$  they may be composed in series to obtain a function  $f \circ g : A \rightarrow C$ . On the other hand, any two functions  $f_1 : A \rightarrow B$  and  $f_2 : C \rightarrow D$  may be composed “in parallel” by taking their cartesian product: we obtain the function  $f_1 \times f_2 : A \times C \rightarrow B \times D$ . This parallel composition of  $f_1$  and  $f_2$  does not rely on any match-up of target and source sets, but it does rely on the “additional structure” provided by the cartesian product.

Such “additional structure” will be formalized in this chapter using the notion of a *monoidal structure*.

Composing components in parallel is of course a very familiar notion in engineering, and the mathematical concepts we develop here will, in particular, model parallel composition in this engineering sense. In the context of co-design of complex systems, for example, we have seen that series composition corresponds to relating the functionalities of one component to the required resources of a next component.

Parallel composition, on the other hand, will correspond to taking two components and thinking of them as a single component whose functionality and resource space are given by the cartesian products of the respective constituent functionality and resource spaces of the original two components.

In general, a monoidal structure will be a notion of “product” and “neutral element” that a category may be equipped with, in which case such is called a *monoidal category*. One thing that could potentially be confusing at this point is the following. At the beginning of this book, we studied monoids as a basic kind of algebraic gadget whose composition operation (also called multiplication) was generalized to the series composition encoded in the definition of a category. In this chapter, we will also use the basic pattern of a monoid as inspiration, but now for parallel composition! Thus, parallel composition is also “monoid-like”, and hence the name *monoidal structure*.

### Types of stacking operations

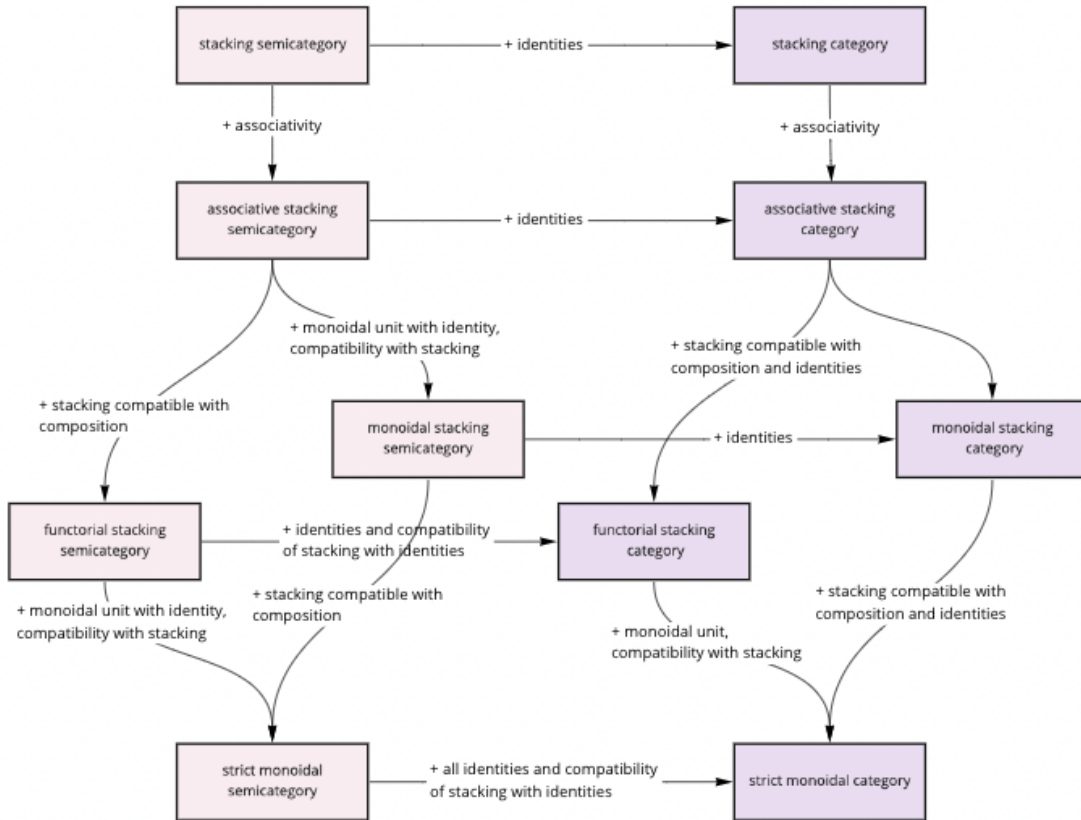
There are various properties that we can consider for semicategories equipped with a stacking operation. This leads to a number of definitions, each of which have a variant for the case when the semicategory in question is in fact a category. Here is a short overview of the most important definitions.

1. *Stacking semicategory* (Def. 30.2): a semicategory in which it is possible to stack two morphisms.
2. *Associative stacking semicategory* (Def. 30.3): a semicategory in which the stacking operation is associative and makes objects and morphisms into a semigroup.
3. *Monoidal stacking semicategory*: an associative stacking semicategory in which there is a special object which is neutral for the stacking operation.
4. *Functorial stacking semicategory* (Def. 30.18): a stacking semicategory in which the stacking operation is also a functor.

5. *Strict monoidal stacking semicategory* (Def. 30.33): an associative stacking semicategory which is both monoidal and functorial.
6. *Symmetric stacking semicategory*: a stacking semicategory in which there is a series of symmetry operations that can permute the objects in a stack.

In the next part we will talk about *monoidal categories* (Def. 34.1), which can be seen as a generalization of associative stacking semicategories where the stacking operation is not required to be associative “on the nose”, but rather only up to isomorphism.

#### KINDS OF STACKING



**Remark 30.1** (Choices in definitions ★). Of all these definitions, the only “classical” one is that of *monoidal category*, and its specializations, such as strict monoidal category, braided monoidal category, and symmetric monoidal category. We will look at all of those in Section 34.1.

We define the three non-classical notions mentioned above for the following reasons:

- ▷ Using the strictification construction of  $\langle \mathbf{Set} \rangle$  all the monoidal categories that we need through the book are strict. Therefore, we can study parallel and feedback composition without knowing natural transformations.
- ▷ There are important cases of semicategories without identities with a stacking operation that do not fit the notion of monoidal category. An example is



discrete-time linear systems. For these semicategories we also want to consider appropriate traces.

- ▷ There are important examples in which there is a way to stack morphisms, but the stacking is not functorial. For example, we will describe a category of effectful computation with side effects.

## 30.2. Stacking semicategories

So far we have seen how we can compose morphisms “horizontally”:

$$\frac{f : X \rightarrow Y \quad g : Y \rightarrow Z}{f \circ g : X \rightarrow Z} . \quad (1)$$

There are other notions of composition that allow us to compose morphisms by “stacking them vertically”. Given two morphisms

$$f : X \rightarrow Y, \quad (2)$$

$$g : Z \rightarrow U, \quad (3)$$

we will obtain by parallel stacking a morphism

$$(f \otimes g) : (X \otimes Z) \rightarrow (Y \otimes U), \quad (4)$$

where “ $\otimes$ ” and “ $\otimes$ ” are operations to be defined. Note that while in the case of sequential composition there was a compatibility condition to be defined, as the target of the first morphism must be the source of the second morphism, here instead we can stack arbitrary morphisms.

We also expect to be able to stack any number of morphisms. Having a collection of morphisms

$$f_i : X_i \rightarrow Y_i, \quad 1 \leq i \leq n, \quad (5)$$

we expect to be able to obtain the composed morphism

$$(\otimes_{i=1}^n f_i) : (\otimes_{i=1}^n X_i) \rightarrow (\otimes_{i=1}^n Y_i). \quad (6)$$

### Stacking semicategories

#### Definition 30.2 (Stacking semicategory)

A *stacking semicategory* is a semicategory  $\mathbf{C}$  with the following additional constituents and properties.

##### Constituents

- ▷ A stacking operation  $\otimes : \text{Ob}_{\mathbf{C}} \times \text{Ob}_{\mathbf{C}} \rightarrow \text{Ob}_{\mathbf{C}}$ .
- ▷ A stacking operation  $\otimes : \text{Mor}_{\mathbf{C}} \times \text{Mor}_{\mathbf{C}} \rightarrow \text{Mor}_{\mathbf{C}}$ .

##### Conditions

- ▷ The two operations  $\otimes$  and  $\otimes$  are compatible in the sense that

$$\frac{f_1 : X_1 \rightarrow Y_1 \quad f_2 : X_2 \rightarrow Y_2}{f_1 \otimes f_2 : X_1 \otimes X_2 \rightarrow Y_1 \otimes Y_2} . \quad (7)$$

In Fig. 1 we have depicted a string diagram of two stacked morphisms. Alternatively, in Fig. 2 we depict the stacking of the string diagrams for  $f_1$  and  $f_2$ , respectively, by stacking their diagrams vertically and drawing a box around them, merging their respective input and output terminals. The outer box denotes  $f_1 \otimes f_2$ ; we think of Fig. 1 as a “black-boxed” version of Fig. 2.

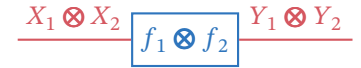


Figure 1.: Stacked morphisms

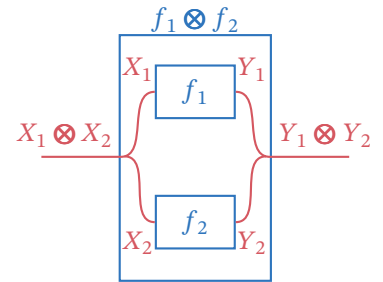
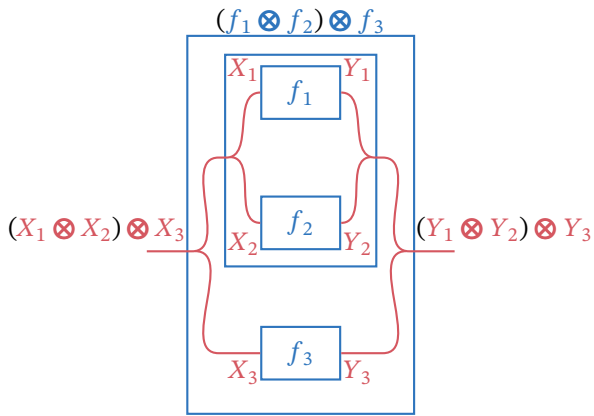
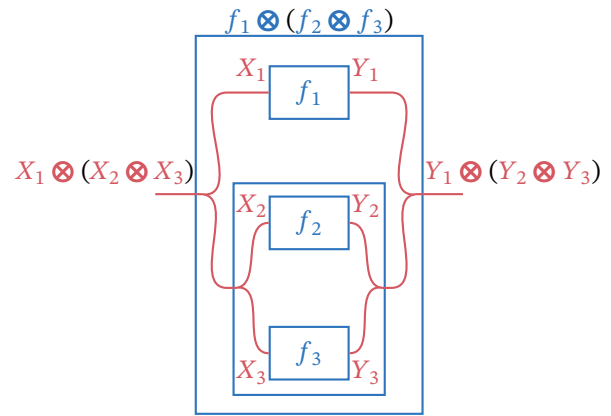


Figure 2.: Stacking string diagrams



(a) One way of stacking three morphisms



(b) Another way of stacking three morphisms

**Figure 3.:** Stacking three morphisms.

### 30.3. Associative stacking (semi)categories

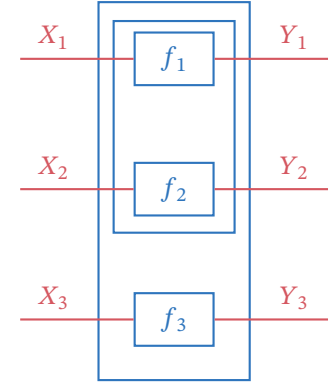
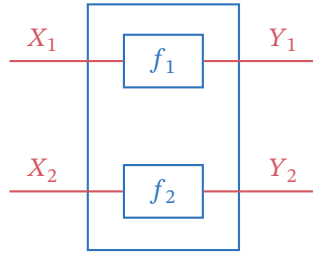
**Definition 30.3** (Associative stacking semicategory)

An *associative stacking semicategory* is a stacking semicategory  $\langle \mathbf{C}, \otimes, \boxtimes \rangle$  satisfying the following.

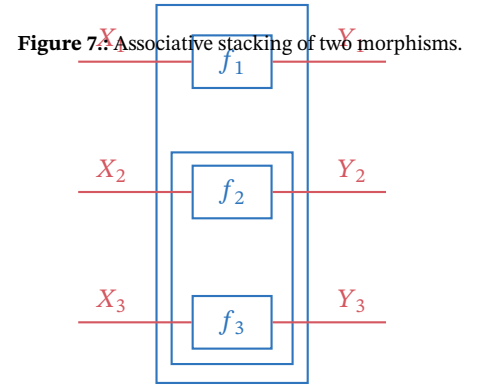
Conditions

- ▷ The stacking operation  $\otimes$  makes the objects into a semigroup.
- ▷ The stacking operation  $\boxtimes$  makes the morphisms into a semigroup.

If a semicategory is associative stacking, then the two ways of stacking three morphisms – as depicted in Fig. 3a and Fig. 3b, respectively – give the same result. For associative stacking semicategories we will use a simpler diagrammatic notation, where the diagrams in Fig. 3a and Fig. 3b instead look like the ones in Fig. 4 and Fig. 5. Since these two diagrams depict that same morphism, when it is convenient we will also simply depict them as in Fig. 6 (and similarly for any number of stacked morphisms). In particular, in an associative stacking semicategory, and stacking of two morphisms will be depicted as in Fig. 7.



**Figure 4.:** Associative stacking of three morphisms, in one order.



**Figure 5.:** Associative stacking of three morphisms, in another order.

**Remark 30.4.** In an associative stacking semicategory it follows that we can stack any number of morphisms without needing to bracket using parentheses:

$$\frac{f_i : X_i \rightarrow Y_i, \quad 1 \leq i \leq n,}{\boxtimes_{i=1}^n f_i : \otimes_{i=1}^n X_i \rightarrow \otimes_{i=1}^n Y_i.} \quad (8)$$

**Definition 30.5** (Associative stacking category)

An *associative stacking category* is a associative stacking semicategory  $\langle \mathbf{C}, \otimes, \boxtimes \rangle$  where  $\mathbf{C}$  is in fact a category.

**Example 30.6.** We let the integers  $\mathbb{Z}$  to be the set of objects of an associative stacking semicategory, and we say that there exists a unique morphism  $X \rightarrow Y$  if and only if  $X \leq Y$ . (We have already seen that this forms a semicategory; in fact, a category.) As our stacking operation for objects we set

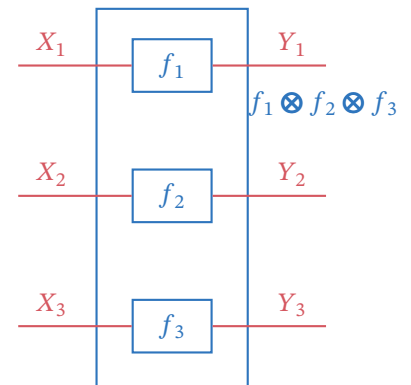
$$X \otimes Y := \max(X, Y) \quad (9)$$

and for stacking morphisms we say that if  $f : X \rightarrow Y$  and  $g : Z \rightarrow U$  exist, then there exists a unique morphism

$$f \boxtimes g : X \otimes Z \rightarrow Y \otimes U \quad (10)$$

which corresponds to (and is consistent with) the inequality

$$\max(X, Z) \leq \max(Y, U). \quad (11)$$



**Figure 6.:** Our string diagram notation for a triple stack.

**Example 30.7.** Let  $\mathbf{A}$  be a non-empty set and consider a semicategory where the collection of objects is  $\text{List } \mathbf{A}$ , the set of non-empty lists of elements of  $\mathbf{A}$ . We can define a pre-order on  $\text{List } \mathbf{A}$  by setting, for any lists  $X, Y$ ,

$$X \leq Y := \text{length}(X) \leq \text{length}(Y). \quad (12)$$

Then we view this pre-order as a semicategory, and define the following stacking operations. Given lists  $X, Y$ , let  $X \otimes Y$  be the concatenation of  $X$  and  $Y$ , and given morphisms  $f : X \rightarrow Y$  and  $g : Z \rightarrow U$  representing inequalities, we let  $f \otimes g : X \otimes Z \rightarrow Y \otimes U$  represent the inequality

$$\text{length}(X \otimes Z) \leq \text{length}(Y \otimes U). \quad (13)$$

**Lemma 30.8.**  $\langle \text{Set} \rangle$  is associative stacking using the structure that arises from tuple concatenation.

*Proof.* For the stacking operation on objects  $\otimes$  we use the operation  $\circ_q$  defined in Section 16.2 which was referred to as the “multiplication in  $\langle \text{Set} \rangle$ ”:

$$\langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle \otimes \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle := \langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle \circ_q \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle \quad (14)$$

$$= \langle \mathbf{A}_1, \dots, \mathbf{A}_m, \mathbf{B}_1, \dots, \mathbf{B}_n \rangle. \quad (15)$$

It was shown there that this operation is associative.

As for  $\otimes$ , we define it as follows:

$$\frac{f : \mathbf{A} \rightarrow_{\langle \text{Set} \rangle} \mathbf{B} \quad g : \mathbf{C} \rightarrow_{\langle \text{Set} \rangle} \mathbf{D}}{(f \otimes g) : \mathbf{A} \circ_q \mathbf{C} \rightarrow_{\langle \text{Set} \rangle} \mathbf{B} \circ_q \mathbf{D}}. \quad (16)$$

$$x \circ_q z \mapsto f(x) \circ_q g(z)$$

The two operations  $\otimes, \otimes$  so defined satisfy the compatibility conditions required by Def. 30.2.

To show associativity, consider three morphisms

$$f : \mathbf{A} \rightarrow_{\langle \text{Set} \rangle} \mathbf{B}, \quad g : \mathbf{C} \rightarrow_{\langle \text{Set} \rangle} \mathbf{D}, \quad h : \mathbf{E} \rightarrow_{\langle \text{Set} \rangle} \mathbf{F}. \quad (17)$$

We compute  $f \otimes (g \otimes h)$  and  $(f \otimes g) \otimes h$  following the recipe (16) to obtain

$$(f \otimes g) \otimes h : (\mathbf{A} \circ_q \mathbf{C}) \circ_q \mathbf{E} \rightarrow_{\langle \text{Set} \rangle} (\mathbf{B} \circ_q \mathbf{D}) \circ_q \mathbf{F}, \quad (18)$$

$$(x \circ_q z) \circ_q v \mapsto (f(x) \circ_q g(z)) \circ_q h(v),$$

$$f \otimes (g \otimes h) : \mathbf{A} \circ_q (\mathbf{C} \circ_q \mathbf{E}) \rightarrow_{\langle \text{Set} \rangle} \mathbf{B} \circ_q (\mathbf{D} \circ_q \mathbf{F}), \quad (19)$$

$$x \circ_q (z \circ_q v) \mapsto f(x) \circ_q (g(z) \circ_q h(v)).$$

Notice that the operations  $\circ_q$  and  $\circ_q$  are associative; therefore, we can remove all the light parentheses that appear in the formulas. This implies that both functions are equal to

$$f \otimes g \otimes h : \mathbf{A} \circ_q \mathbf{C} \circ_q \mathbf{E} \rightarrow_{\langle \text{Set} \rangle} \mathbf{B} \circ_q \mathbf{D} \circ_q \mathbf{F}, \quad (20)$$

$$x \circ_q z \circ_q v \mapsto f(x) \circ_q g(z) \circ_q h(v).$$

□

## One-object associative stacking semicategories

Consider a special kind of a stacking semicategory  $\langle \mathbf{C}, \otimes, \circ \rangle$  where the semicategory  $\mathbf{C}$  has only one object (call it  $X$ , say).

In this special case, we have  $\otimes_{i=1}^n X_i = X$  for any  $n \in \mathbb{N}$ . The only hom-set is  $\text{Hom}_{\mathbf{C}}(X, X)$  and this is equipped with the stacking operation

$$\otimes : \text{Hom}_{\mathbf{C}}(X, X) \times \text{Hom}_{\mathbf{C}}(X, X) \rightarrow \text{Hom}_{\mathbf{C}}(X, X) \quad (21)$$

which makes  $\text{Hom}_{\mathbf{C}}(X, X)$  into a semigroup. In other words, this means that an associative stacking semicategory with one object may equivalently be described as set together with two operations – serial composition and stacking – that equip said set with two semigroup structures.

**Example 30.9.** The integers  $\mathbb{Z}$ , equipped with addition and multiplication as serial composition and stacking respectively, specify an associative stacking category with one object. Alternatively, we may also choose multiplication as our serial composition, and addition as our stacking.

## The semicategory $\mathbf{Eff}$

We now construct a non-trivial semicategory that will serve as an example of a stacking semicategory that is associative but, as we will show later, is not functorial.

There are two types of computation: “functional”, or “pure”, or free of side effects, and *effectful*. In functional programming, functions are very much like mathematical functions: they need an input and produce an output. They don’t interfere with other functions.

Effectful procedures, instead, can “change the world”: for example, printing a page, sending an email, or placing an order of pizza. The order in which effectful procedures are evaluated might change the result. For example, the result of the sequence of operations

1. Order a pizza;
2. Cancel the last order;

is different from the result of the sequence of operations

1. Cancel the last order;
2. Order a pizza;

A very elegant way to treat side effects mathematically is using *linear types* [34]. We will mention those in a successive part on linear logic. For now, we stick to a simple treatment.

We are going to define a category  $\mathbf{Eff}$ . The idea is to add another variable that represents “the world” that can be affected. An effectful function

$$f : X \rightarrow_{\mathbf{Eff}} Y, \quad (22)$$

which could have some unknown side effects on the world, can be represented by a pure function

$$\text{rep}(f) : X \times \mathbb{W} \rightarrow_{\mathbf{Set}} Y \times \mathbb{W}, \quad (23)$$

or, in other words, as a morphism

$$\text{rep}(f) : \langle X, \mathbb{W} \rangle \rightarrow_{\langle \mathbf{Set} \rangle} \langle Y, \mathbb{W} \rangle, \quad (24)$$

where  $\mathbb{W}$  is the set of all possible worlds (Fig. 8).

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

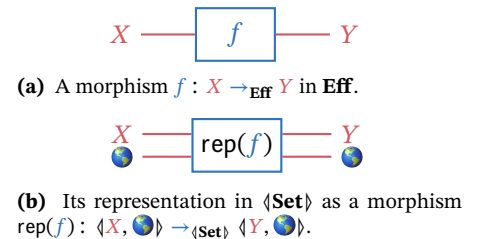


Figure 8.

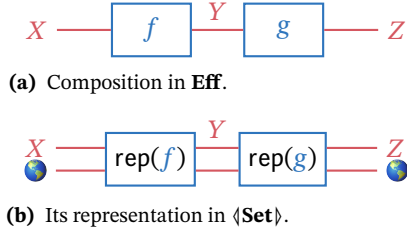


Figure 9.

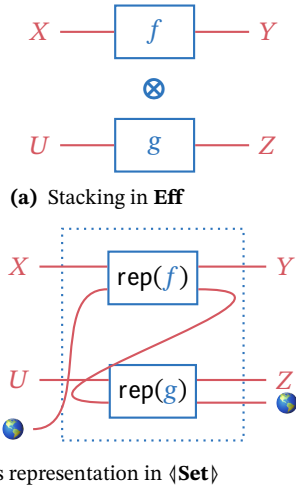


Figure 10.

The second input to  $\text{rep}(f)$  is the state of the world before the execution of the function. The second output of  $\text{rep}(f)$  is the state of the world after the execution of the function.

We can now interconnect different effectful functions, with some precautions. We cannot “split the world”, by creating a function of type  $\text{World} \rightarrow \text{World} \times \text{World}$ . We will re-state this formally when we get to linear logic.

We can extend usual function composition to composition of effectful functions as in Fig. 9. The second effectful function operates on the world after it was possibly modified by the first effectful function.

We have all the ingredients to define the category **Eff** of effectful computation.

**Definition 30.10** (Category of effectful procedures **Eff**)

Fix a set  $\text{World}$  of all possible worlds. The category **Eff** is defined by the following:

- ▷ *Objects*: same as the objects of  $\langle \mathbf{Set} \rangle$ ;
- ▷ *Morphisms*: a morphism  $f : X \rightarrow_{\mathbf{Eff}} Y$  is a morphism

$$\text{rep}(f) : \langle X, \text{World} \rangle \rightarrow_{\langle \mathbf{Set} \rangle} \langle Y, \text{World} \rangle. \quad (25)$$

- ▷ *Composition*: The composition of  $f : X \rightarrow_{\mathbf{Eff}} Y$  and  $g : Y \rightarrow_{\mathbf{Eff}} Z$  is the morphism  $f \circ g : X \rightarrow_{\mathbf{Eff}} Z$  with  $\text{rep}(f \circ g)$  given by

$$\text{rep}(f \circ_{\mathbf{Eff}} g) = \text{rep}(f) \circ_{\langle \mathbf{Set} \rangle} \text{rep}(g) \quad (26)$$

as illustrated in Fig. 9.

We can now make **Eff** into a stacking semicategory by deciding how to evaluate a stack of functions. We cannot evaluate them in parallel. We need to decide on a sequence. For example, top to bottom. This is shown in Fig. 10.

**Lemma 30.11.** **Eff** is an associative stacking category using the evaluation rule as in Fig. 10.

## 30.4. Examples of associative stacking semicategories

### Moo is associative stacking

When considering Moore machines, we can define stacking operations and show that **Moo** forms a stacking semicategory (Def. 30.2). The objects of **Moo** are objects of  $\langle \mathbf{Set} \rangle$ , and therefore the stacking operation for objects corresponds to the “multiplication in  $\langle \mathbf{Set} \rangle$ ”, denoted by  $\circledast$ .

The operation on morphisms “stacks” Moore machines onto each other. Formally:

$$\frac{f : \mathbf{U}_f \rightarrow_{\mathbf{Moo}} \mathbf{Y}_f \quad g : \mathbf{U}_g \rightarrow_{\mathbf{Moo}} \mathbf{Y}_g}{f \otimes g = \langle \mathbf{U}_f \circledast \mathbf{U}_g, \mathbf{X}_f \circledast \mathbf{X}_g, \mathbf{Y}_f \circledast \mathbf{Y}_g, \text{dyn}_{f \otimes g}, \text{rof}_{\otimes g}, \text{st}_f \circledast \text{st}_g \rangle}, \quad (27)$$

with

$$\begin{aligned} \text{dyn}_{f \otimes g} : \mathbf{U}_f \circledast \mathbf{U}_g \circledast \mathbf{X}_f \circledast \mathbf{X}_g &\rightarrow \mathbf{X}_f \circledast \mathbf{X}_g, \\ u_f \circledast u_g \circledast x_f \circledast x_g &\mapsto \text{dyn}_f(u_f \circledast x_f) \circledast \text{dyn}_g(u_g \circledast x_g), \end{aligned}$$

and

$$\begin{aligned} \text{rof}_{\otimes g} : \mathbf{X}_f \circledast \mathbf{X}_g &\rightarrow \mathbf{Y}_f \circledast \mathbf{Y}_g, \\ x_f \circledast x_g &\mapsto \text{rof}(x_f) \circledast \text{rof}(x_g). \end{aligned}$$

While we have already proved that the operation  $\circledast$  is associative, it is also easy to see that the stacking of Moore machines is associative. Therefore, **Moo** equipped with the described stacking operations forms an associative stacking semicategory.

### LTI is associative stacking

When considering LTI systems, we can define stacking operations and show that **LTI** forms a stacking semicategory (Def. 30.2). The stacking operation on objects is just addition on the natural numbers:

$$\begin{aligned} \otimes : \text{Ob}_{\mathbf{LTI}} \times \text{Ob}_{\mathbf{LTI}} &\rightarrow \text{Ob}_{\mathbf{LTI}}, \\ \langle l, m \rangle &\mapsto l + m. \end{aligned} \quad (28)$$

On the other hand, the operation on morphisms “stacks” the LTI systems onto each other\*. Formally:

$$\frac{f : l \rightarrow_{\mathbf{LTI}} m \quad g : n \rightarrow_{\mathbf{LTI}} o}{f \otimes g = \langle \text{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle : l + n \rightarrow_{\mathbf{LTI}} m + o}. \quad (29)$$

with

$$\text{st} = \begin{bmatrix} \text{st}_f \\ \text{st}_g \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_g \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_f & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_g \end{bmatrix}, \quad (30)$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_f & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_g \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} \mathbf{D}_f & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_g \end{bmatrix} \quad (31)$$

\* For the control engineers out there: the resulting LTI system will have a Relative Gain Array (RGA) matrix corresponding to the identity matrix.



Given that the resulting LTI system is described by diagonal block matrices, it is evident that the defined operations are associative, and therefore **LTI** is an associative stacking semicategory.

**Graded exercise H.1** (StackingLTI)

Consider the category of finite-dimensional linear time-invariant systems defined in Def. 16.26 with the stacking defined above.

Your task: supposing that morphisms  $f: l \rightarrow m$ ,  $h: m \rightarrow n$ ,  $g: o \rightarrow p$ ,  $i: p \rightarrow q$  are given, compute the matrices

$$\mathbf{A}_{(f \circ h) \otimes (g \circ i)} \quad (32)$$

and

$$\mathbf{A}_{(f \otimes g) \circ (h \otimes i)} \quad (33)$$

associated with the morphisms  $(f \circ h) \otimes (g \circ i)$  and  $(f \otimes g) \circ (h \otimes i)$ , respectively.

### The category $\langle \mathbf{Pos} \rangle$

We define a category analogous to  $\langle \mathbf{Set} \rangle$ , but its objects are “tuple posets”.

Given posets  $P_1, \dots, P_n$  we define the poset

$$\langle P_1, \dots, P_n \rangle := \langle \langle P_1, \dots, P_n \rangle, \leq_{\langle P_1, \dots, P_n \rangle} \rangle, \quad (34)$$

where  $\langle P_1, \dots, P_n \rangle$  is a set of tuples, and we use the product order:

$$\frac{\langle x_1, \dots, x_n \rangle \leq_{\langle P_1, \dots, P_n \rangle} \langle y_1, \dots, y_n \rangle}{x_i \leq_{P_i} y_i \text{ for all } i \in \{1, \dots, n\}}. \quad (35)$$

**Definition 30.12** (The category  $\langle \mathbf{Pos} \rangle$ )

The category  $\langle \mathbf{Pos} \rangle$  is the subcategory of **Pos** where the objects are tuple posets, posets of the form

$$\langle P_1, \dots, P_n \rangle = \langle \langle P_1, \dots, P_n \rangle, \leq_{\langle P_1, \dots, P_n \rangle} \rangle. \quad (36)$$

**Lemma 30.13.**  $\langle \mathbf{Pos} \rangle$  is associative stacking using the structure induced by tuple concatenation.

*Proof.* Analogously to what we did for  $\langle \mathbf{Set} \rangle$ , we can define a multiplication operation  $\otimes$  in  $\langle \mathbf{Pos} \rangle$ .

Given two objects  $P = \langle P_1, \dots, P_n \rangle$  and  $Q = \langle Q_1, \dots, Q_n \rangle$ , we define

$$\langle P_1, \dots, P_n \rangle \otimes \langle Q_1, \dots, Q_n \rangle := \langle P_1, \dots, P_n, Q_1, \dots, Q_n \rangle. \quad (37)$$

For the multiplication on morphisms, we define

$$\frac{f: P \rightarrow Q \quad g: R \rightarrow S}{(f \otimes g): P \otimes R \rightarrow Q \otimes S}.$$

$$p \circledcirc r \mapsto f(p) \circledcirc g(r) \quad (38)$$

We need to check that the expression  $f(p) \circledcirc g(r)$  is monotone. This can be easily seen because the order on  $Q \otimes S$  is akin to a product order. The proof

for associativity of  $\otimes$  is the same as in the proof of  $\langle \mathbf{Set} \rangle$  (Lemma 30.8).  $\square$

## The category $\langle \mathbf{Rel} \rangle$

We define a category  $\langle \mathbf{Rel} \rangle$  where the objects are sets of tuples (as in  $\langle \mathbf{Set} \rangle$ ).

**Definition 30.14** (The category  $\langle \mathbf{Rel} \rangle$ )

The category  $\langle \mathbf{Rel} \rangle$  is the subcategory of  $\mathbf{Rel}$  where the objects are tuples sets (objects of  $\langle \mathbf{Set} \rangle$ ).

**Lemma 30.15.**  $\langle \mathbf{Rel} \rangle$  is associative stacking with the structure induced by tuple concatenation.

*Proof.* The multiplication  $\otimes$  is  $\circ_{\langle \rangle}$ , the same as the one defined for  $\langle \mathbf{Set} \rangle$ .

The multiplication  $\otimes$  is defined as follows:

$$\frac{R \subseteq A \times B \quad S \subseteq C \times D}{(R \otimes S) \subseteq (A \circ_{\langle \rangle} C) \times (B \circ_{\langle \rangle} D)} \quad (39)$$

where

$$(R \otimes S) = \{ \langle a \circ_{\langle \rangle} c, b \circ_{\langle \rangle} d \rangle \mid (a R b) \wedge (c S d) \}. \quad (40)$$

The rest of the proof is left as an exercise.  $\square$

**Exercise 45.** Consider the stacking operations on objects and on morphisms introduced in this section. Prove that  $\langle \mathbf{Rel} \rangle$  is associative stacking.

*See solution on page 459.*

## The category $\langle \mathbf{DP} \rangle$

Analogously, we define the category  $\langle \mathbf{DP} \rangle$ .

**Definition 30.16** (The category  $\langle \mathbf{DP} \rangle$ )

The category  $\langle \mathbf{DP} \rangle$  is the subcategory of  $\mathbf{DP}$  where the objects are posets of tuple posets (objects of  $\langle \mathbf{Pos} \rangle$ ).

**Lemma 30.17.**  $\langle \mathbf{DP} \rangle$  is associative stacking using the structure induced by tuple concatenation.

*Proof.* For the stacking operation  $\otimes$  on objects, we use  $\otimes_{\langle \mathbf{DP} \rangle} := \otimes_{\langle \mathbf{Pos} \rangle}$ . For stacking morphisms, we define  $\otimes$  by

$$\frac{\mathbf{d} : P^{\text{op}} \times R \rightarrow \mathbf{Bool} \quad \mathbf{e} : Q^{\text{op}} \times S \rightarrow \mathbf{Bool}}{\mathbf{d} \otimes \mathbf{e} : (P \otimes Q)^{\text{op}} \times (R \otimes S) \rightarrow \mathbf{Bool}} \quad (41)$$

$$\langle a^* \circ_{\langle \rangle} c^*, b \circ_{\langle \rangle} d \rangle \mapsto \mathbf{d}(a^*, c) \wedge \mathbf{e}(c^*, d)$$

Note that this is a valid definition of a design problem because the expression  $\mathbf{d}(a^*, c) \wedge \mathbf{e}(c^*, d)$  is monotone, as may readily be checked.

To show associativity, consider three DPs

$$\mathbf{d} : P^{\text{op}} \times R \rightarrow_{\mathbf{Pos}} \mathbf{Bool}, \quad \mathbf{e} : Q^{\text{op}} \times S \rightarrow_{\mathbf{Pos}} \mathbf{Bool}, \quad \mathbf{g} : T^{\text{op}} \times U \rightarrow_{\mathbf{Pos}} \mathbf{Bool}. \quad (42)$$

We compute  $(\mathbf{d} \otimes \mathbf{e}) \otimes \mathbf{g}$  and  $\mathbf{d} \otimes (\mathbf{e} \otimes \mathbf{g})$  according to the recipe in (41):

$$(\mathbf{d} \otimes \mathbf{e}) \otimes \mathbf{g} : ((\mathbf{P} \otimes \mathbf{Q}) \otimes \mathbf{T})^{\text{op}} \times (\mathbf{R} \otimes \mathbf{S}) \otimes \mathbf{U} \rightarrow_{\text{Pos}} \mathbf{Bool},$$

$$\left\langle \left( a^* \circ_{\langle} c^* \right) \circ_{\langle} e^*, \left( b \circ_{\langle} d \right) \circ_{\langle} f \right\rangle \mapsto (\mathbf{d}(a^*, c) \wedge \mathbf{e}(c^*, d)) \wedge \mathbf{g}(b, f)$$

$$\mathbf{d} \otimes (\mathbf{e} \otimes \mathbf{g}) : (\mathbf{P} \otimes (\mathbf{Q} \otimes \mathbf{T}))^{\text{op}} \times (\mathbf{R} \otimes (\mathbf{S} \otimes \mathbf{U})) \rightarrow_{\text{Pos}} \mathbf{Bool},$$

$$\left\langle a^* \circ_{\langle} (c^* \circ_{\langle} e^*), b \circ_{\langle} (d \circ_{\langle} f) \right\rangle \mapsto \mathbf{d}(a^*, c) \wedge (\mathbf{e}(c^*, d) \wedge \mathbf{g}(b, f))$$

Because the operations  $\otimes$  and  $\wedge$  are associative, we can erase all the light parentheses in  $(\mathbf{d} \otimes \mathbf{e}) \otimes \mathbf{g}$  and  $\mathbf{d} \otimes (\mathbf{e} \otimes \mathbf{g})$  are equal to the design problem

$$\mathbf{d} \otimes \mathbf{e} \otimes \mathbf{g} : (\mathbf{P} \otimes \mathbf{Q} \otimes \mathbf{T})^{\text{op}} \times (\mathbf{R} \otimes \mathbf{S} \otimes \mathbf{U}) \rightarrow_{\text{Pos}} \mathbf{Bool},$$

$$\left\langle a^* \circ_{\langle} c^* \circ_{\langle} e^*, b \circ_{\langle} d \circ_{\langle} f \right\rangle \mapsto \mathbf{d}(a^*, c) \wedge \mathbf{e}(c^*, d) \wedge \mathbf{g}(b, f)$$

## 30.5. Functorial stacking semicategories

**Definition 30.18** (Functorial stacking semicategory)

A *functorial stacking semicategory* is a stacking semicategory where the two stacking operations  $\otimes$  and  $\circ$  are the two components of a semifunctor

$$\otimes : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}. \quad (46)$$

Concretely, and in infix notation, this means that

$$(f \circ h) \otimes (g \circ i) = (f \otimes g) \circ (h \otimes i) \quad (47)$$

for all morphisms  $f, g, h$ , and  $i$  where respectively  $f$  and  $h$ , and  $g$  and  $i$  are composable.

This describes a sort of commutativity property: we can either first compose horizontally and then vertically, or vice versa, and we obtain the same morphism (Fig. 11).

This property is somewhat strong. We give several examples ( $\langle \mathbf{Set} \rangle$ ,  $\langle \mathbf{Pos} \rangle$ ,  $\langle \mathbf{Rel} \rangle$ ,  $\langle \mathbf{DP} \rangle$ ), and then several counterexamples.

### Examples

**Example 30.19** ( $\langle \mathbf{Set} \rangle$  is a functorial stacking semicategory). We want to show that  $\langle \mathbf{Set} \rangle$  is a functorial stacking semicategory.

Consider four morphisms

$$\begin{aligned} f &: \mathbf{A} \rightarrow \mathbf{B}, & h &: \mathbf{B} \rightarrow \mathbf{C}, \\ g &: \mathbf{D} \rightarrow \mathbf{E}, & i &: \mathbf{E} \rightarrow \mathbf{F}. \end{aligned} \quad (48)$$

We want to show that

$$(f \otimes g) \circ (h \otimes i) = (f \circ h) \otimes (g \circ i), \quad (49)$$

We show this by showing that, for any  $a \circlearrowleft d \in \mathbf{A} \circlearrowleft \mathbf{D}$ :

$$\begin{aligned} ((f \otimes g) \circ (h \otimes i))(a \circlearrowleft d) &= (h \otimes i)(f(a) \circlearrowleft g(d)) \\ &= h(f(a)) \circlearrowleft i(g(d)) \\ &= (f \circ h)(a) \circlearrowleft (g \circ i)(d) \\ &= ((f \circ h) \otimes (g \circ i))(a \circlearrowleft d). \end{aligned} \quad (50)$$

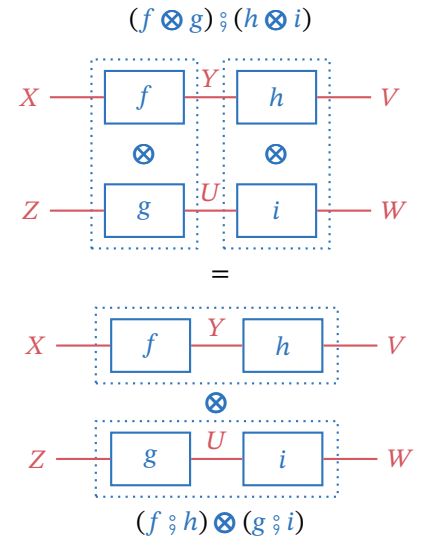
**Lemma 30.20.**  $\langle \mathbf{Pos} \rangle$ , equipped with the aforementioned stacking operations on objects and morphisms, is functorial stacking semicategory.

*Proof.* The proof is analogous to the one for  $\langle \mathbf{Set} \rangle$  in Example 30.19.  $\square$

### Graded exercise H.2 (RelFunStack)

Prove that the structure defined in Exercise 45 makes  $\langle \mathbf{Rel} \rangle$  a functorial stacking semicategory.

**Lemma 30.21** ( $\langle \mathbf{DP} \rangle$  is a functorial stacking semicategory).  $\langle \mathbf{DP} \rangle$ , equipped with the aforementioned stacking operations on objects and morphisms, is functorial stacking semicategory.



**Figure 11.** Commutation of stacking and composition in a functorial stacking semicategory.

*Proof.* Consider

$$\begin{aligned}
 \mathbf{d} &: \mathbf{P}^{\text{op}} \times \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{Bool} \\
 \mathbf{e} &: \mathbf{Q}^{\text{op}} \times \mathbf{S} \rightarrow_{\text{Pos}} \mathbf{Bool} \\
 \mathbf{g} &: \mathbf{R}^{\text{op}} \times \mathbf{T} \rightarrow_{\text{Pos}} \mathbf{Bool} \\
 \mathbf{h} &: \mathbf{S}^{\text{op}} \times \mathbf{U} \rightarrow_{\text{Pos}} \mathbf{Bool}
 \end{aligned} \tag{51}$$

We want to prove that

$$(\mathbf{d} \circledast \mathbf{g}) \otimes (\mathbf{e} \circledast \mathbf{h}) = (\mathbf{d} \otimes \mathbf{e}) \circledast (\mathbf{g} \otimes \mathbf{h}). \tag{52}$$

We start from the left-hand side. We have

$$(\mathbf{d} \circledast \mathbf{g})(p^*, t) = \bigvee_{r \in \mathbf{R}} \mathbf{d}(p^*, r) \wedge \mathbf{g}(r^*, t) \tag{53}$$

and

$$(\mathbf{e} \circledast \mathbf{h})(q^*, u) = \bigvee_{s \in \mathbf{S}} \mathbf{e}(q^*, s) \wedge \mathbf{h}(s^*, u) \tag{54}$$

Therefore, we know

$$\begin{aligned}
 &((\mathbf{d} \circledast \mathbf{g}) \otimes (\mathbf{e} \circledast \mathbf{h}))((p \circledast_{\mathbf{Q}} q)^*, t \circledast_{\mathbf{T}} u) \\
 &= \bigvee_{r \in \mathbf{R}} \mathbf{d}(p^*, r) \wedge \mathbf{g}(r^*, t) \wedge \bigvee_{s \in \mathbf{S}} \mathbf{e}(q^*, s) \wedge \mathbf{h}(s^*, u).
 \end{aligned} \tag{55}$$

On the other hand, we have

$$(\mathbf{d} \otimes \mathbf{e})((p \circledast_{\mathbf{Q}} q)^*, r \circledast_{\mathbf{S}} s) = \mathbf{d}(p^*, r) \wedge \mathbf{e}(q^*, s) \tag{56}$$

and

$$(\mathbf{g} \otimes \mathbf{h})((r \circledast_{\mathbf{T}} s)^*, t \circledast_{\mathbf{U}} u) = \mathbf{g}(r^*, t) \wedge \mathbf{h}(s^*, u) \tag{57}$$

Therefore, we know

$$\begin{aligned}
 &((\mathbf{d} \otimes \mathbf{e}) \circledast (\mathbf{g} \otimes \mathbf{h}))((p \circledast_{\mathbf{Q}} q)^*, t \circledast_{\mathbf{T}} u) \\
 &= \bigvee_{r \circledast_{\mathbf{S}} s \in \langle \mathbf{R}, \mathbf{S} \rangle} (\mathbf{d} \otimes \mathbf{e})((p \circledast_{\mathbf{Q}} q)^*, r \circledast_{\mathbf{S}} s) \wedge (\mathbf{g} \otimes \mathbf{h})((r \circledast_{\mathbf{T}} s)^*, t \circledast_{\mathbf{U}} u) \\
 &= \bigvee_{r \circledast_{\mathbf{S}} s \in \langle \mathbf{R}, \mathbf{S} \rangle} \mathbf{d}(p^*, r) \wedge \mathbf{e}(q^*, s) \wedge \mathbf{g}(r^*, t) \wedge \mathbf{h}(s^*, u) \\
 &= \bigvee_{r \in \mathbf{R}} \mathbf{d}(p^*, r) \wedge \mathbf{g}(r^*, t) \wedge \bigvee_{s \in \mathbf{S}} \mathbf{e}(q^*, s) \wedge \mathbf{h}(s^*, u),
 \end{aligned} \tag{58}$$

proving the statement for any posets  $\mathbf{P}, \mathbf{Q}, \mathbf{R}, \mathbf{S}, \mathbf{T}, \mathbf{U}$  (and hence, also for posets of tuples). □

## Counter-examples

**Eff** is an example of a stacking category that is not functorial.

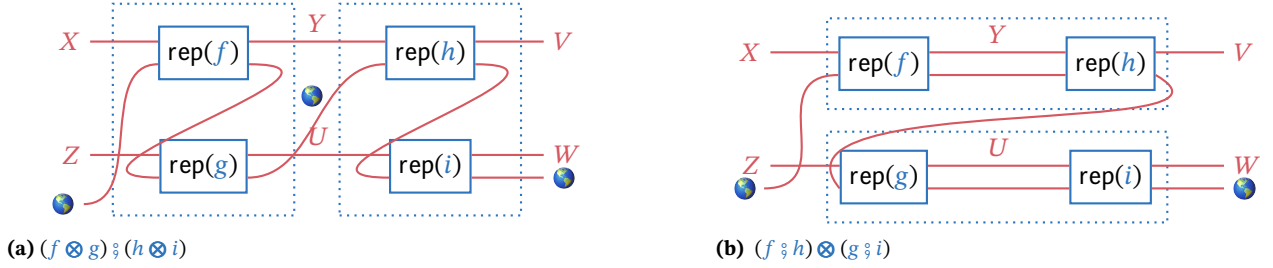
**Lemma 30.22.** **Eff** is not a functorial stacking semicategory.

*Proof.* In general, we have

$$(f \otimes g) \circledast (h \otimes i) \neq (f \circledast h) \otimes (g \circledast i). \tag{59}$$

This is shown graphically in Fig. 12.

□



**Figure 12.:** Proof that **Eff** is not a functorial stacking category by showing that the two morphisms above have different representations in  $\langle \mathbf{Set} \rangle$ .

**Example 30.23.** We want to show that **Moo**, equipped with the defined stacking operations, is almost a functorial stacking semicategory, but not quite. Consider four Moore machines:

$$f : \mathbf{U}_f \rightarrow \mathbf{Y}_f, \quad g : \mathbf{U}_g \rightarrow \mathbf{Y}_g, \quad h : \mathbf{Y}_f \rightarrow \mathbf{Y}_h, \quad i : \mathbf{Y}_g \rightarrow \mathbf{Y}_i.$$

We want to check if the equation

$$(f \circ h) \otimes (g \circ i) = (f \otimes g) \circ (h \otimes i)$$

holds. Let's start from the left-hand side. First, we have:

$$\begin{aligned} f \circ h &= \langle \mathbf{U}_f, \mathbf{X}_f \circ \mathbf{X}_h, \mathbf{Y}_h, \text{dyn}_{f \circ h}, \text{ro}_{f \circ h}, \text{st}_f \circ \text{st}_h \rangle, \\ g \circ i &= \langle \mathbf{U}_g, \mathbf{X}_g \circ \mathbf{X}_i, \mathbf{Y}_i, \text{dyn}_{g \circ i}, \text{ro}_{g \circ i}, \text{st}_g \circ \text{st}_i \rangle, \end{aligned}$$

with

$$\begin{aligned} \text{dyn}_{f \circ h} : \mathbf{U}_f \circ \mathbf{X}_f \circ \mathbf{X}_h &\rightarrow \mathbf{X}_f \circ \mathbf{X}_h, \\ u_f \circ x_f \circ x_h &\mapsto \text{dyn}_f(u_f \circ x_f) \circ \text{dyn}_h(\text{ro}_f(x_f) \circ x_h), \\ \text{dyn}_{g \circ i} : \mathbf{U}_g \circ \mathbf{X}_g \circ \mathbf{X}_i &\rightarrow \mathbf{X}_g \circ \mathbf{X}_i, \\ u_g \circ x_g \circ x_i &\mapsto \text{dyn}_g(u_g \circ x_g) \circ \text{dyn}_i(\text{ro}_g(x_g) \circ x_i), \\ \text{ro}_{f \circ h} : \mathbf{X}_f \circ \mathbf{X}_h &\rightarrow \mathbf{Y}_h, \\ x_f \circ x_h &\mapsto \text{ro}_h(x_h), \end{aligned}$$

and

$$\begin{aligned} \text{ro}_{g \circ i} : \mathbf{X}_g \circ \mathbf{X}_i &\rightarrow \mathbf{Y}_i, \\ x_g \circ x_i &\mapsto \text{ro}_i(x_i). \end{aligned}$$

Furthermore:

$$(f \circ h) \otimes (g \circ i) = \langle \mathbf{U}_f \circ \mathbf{U}_g, \mathbf{X}_f \circ \mathbf{X}_h \circ \mathbf{X}_g \circ \mathbf{X}_i, \mathbf{Y}_h \circ \mathbf{Y}_i, \text{dyn}_{(f \circ h) \otimes (g \circ i)}, \text{ro}_{(f \circ h) \otimes (g \circ i)}, \text{st}_f \circ \text{st}_h \circ \text{st}_g \circ \text{st}_i \rangle,$$

with

$$\begin{aligned} \text{dyn}_{(f \circ h) \otimes (g \circ i)} : \mathbf{U}_f \circ \mathbf{U}_g \circ \mathbf{X}_f \circ \mathbf{X}_h \circ \mathbf{X}_g \circ \mathbf{X}_i &\rightarrow \mathbf{X}_h \circ \mathbf{X}_g \circ \mathbf{X}_i, \\ u_f \circ u_g \circ x_f \circ x_h \circ x_g \circ x_i &\mapsto \underbrace{\text{dyn}_{f \circ h}(u_f \circ x_f \circ x_h) \circ \text{dyn}_{g \circ i}(u_g \circ x_g \circ x_i)}_{(1)} \end{aligned}$$

where

$$(1) = \text{dyn}_f(u_f \circ x_f) \circ \text{dyn}_h(\text{ro}_f(x_f) \circ x_h) \circ \text{dyn}_g(u_g \circ x_g) \circ \text{dyn}_i(\text{ro}_g(x_g) \circ x_i),$$

and

$$\begin{aligned} \text{ro}_{(f \circ h) \otimes (g \circ i)} : \mathbf{X}_f \circ \mathbf{X}_h \circ \mathbf{X}_g \circ \mathbf{X}_i &\rightarrow \mathbf{Y}_h \circ \mathbf{Y}_i, \\ x_f \circ x_h \circ x_g \circ x_i &\mapsto \underbrace{\text{ro}_{f \circ h}(x_f \circ x_h) \circ \text{ro}_{g \circ i}(x_g \circ x_i)}_{(2)}, \end{aligned}$$

where

$$(2) = \text{ro}_h(x_h) \circ \text{ro}_i(x_i).$$

On the other hand, we have:

$$\begin{aligned} f \otimes g &= \langle \mathbf{U}_f \circ \mathbf{U}_g, \mathbf{X}_f \circ \mathbf{X}_g, \mathbf{Y}_f \circ \mathbf{Y}_g, \text{dyn}_{f \otimes g}, \text{ro}_{f \otimes g}, \text{st}_f \circ \text{st}_g \rangle, \\ h \otimes i &= \langle \mathbf{U}_h \circ \mathbf{U}_i, \mathbf{X}_h \circ \mathbf{X}_i, \mathbf{Y}_h \circ \mathbf{Y}_i, \text{dyn}_{h \otimes i}, \text{ro}_{h \otimes i}, \text{st}_h \circ \text{st}_i \rangle, \end{aligned}$$

with

$$\begin{aligned} \text{dyn}_{f \otimes g} : \mathbf{U}_f \circ \mathbf{U}_g \circ \mathbf{X}_f \circ \mathbf{X}_g &\rightarrow \mathbf{X}_f \circ \mathbf{X}_g, \\ u_f \circ u_g \circ x_f \circ x_g &\mapsto \text{dyn}_f(u_f \circ x_f) \circ \text{dyn}_g(u_g \circ x_g), \\ \text{dyn}_{h \otimes i} : \mathbf{U}_h \circ \mathbf{U}_i \circ \mathbf{X}_h \circ \mathbf{X}_i &\rightarrow \mathbf{X}_h \circ \mathbf{X}_i, \\ u_h \circ u_i \circ x_h \circ x_i &\mapsto \text{dyn}_h(u_h \circ x_h) \circ \text{dyn}_i(u_i \circ x_i), \\ \text{ro}_{f \otimes g} : \mathbf{X}_f \circ \mathbf{X}_g &\rightarrow \mathbf{Y}_f \circ \mathbf{Y}_g, \\ x_f \circ x_g &\mapsto \text{ro}_f(x_f) \circ \text{ro}_g(x_g), \end{aligned}$$

and

$$\begin{aligned} \text{ro}_{h \otimes i} : \mathbf{X}_h \circ \mathbf{X}_i &\rightarrow \mathbf{Y}_h \circ \mathbf{Y}_i, \\ x_h \circ x_i &\mapsto \text{ro}_h(x_h) \circ \text{ro}_i(x_i). \end{aligned}$$

Furthermore:

$$(f \otimes g) \circ (h \otimes i) = \langle \mathbf{U}_f \circ \mathbf{U}_g, \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h \circ \mathbf{X}_i, \mathbf{Y}_h \circ \mathbf{Y}_i, \text{dyn}_{(f \otimes g) \otimes (h \otimes i)}, \text{ro}_{(f \otimes g) \circ (h \otimes i)}, \text{st}_f \circ \text{st}_g \circ \text{st}_h \circ \text{st}_i \rangle,$$

with

$$\begin{aligned} \text{dyn}_{(f \otimes g) \circ (h \otimes i)} : \mathbf{U}_f \circ \mathbf{U}_g \circ \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h \circ \mathbf{X}_i &\rightarrow \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h \circ \mathbf{X}_i, \\ u_f \circ u_g \circ x_f \circ x_g \circ x_h \circ x_i &\mapsto \underbrace{\text{dyn}_{f \otimes g}(u_f \circ u_g \circ x_f \circ x_g) \circ \text{dyn}_{h \otimes i}(\text{ro}_{f \otimes g}(x_f \circ x_g) \circ x_h \circ x_i)}_{(3)}, \end{aligned}$$

with

$$\begin{aligned} (3) &= \text{dyn}_f(u_f \circ x_f) \circ \text{dyn}_g(u_g \circ x_g) \circ \text{dyn}_{h \otimes i}(\text{ro}_f(x_f) \circ \text{ro}_g(x_g) \circ x_h \circ x_i) \\ &= \text{dyn}_f(u_f \circ x_f) \circ \underbrace{\text{dyn}_g(u_g \circ x_g)}_{(*)} \circ \underbrace{\text{dyn}_h(\text{ro}_f(x_f) \circ x_h) \circ \text{dyn}_i(\text{ro}_g(x_g) \circ x_i)}_{(**)} \end{aligned}$$

and

$$\begin{aligned} \text{ro}_{(f \otimes g) \circ (h \otimes i)} : \mathbf{X}_f \circ \mathbf{X}_g \circ \mathbf{X}_h \circ \mathbf{X}_i &\rightarrow \mathbf{Y}_h \circ \mathbf{Y}_i, \\ x_f \circ x_g \circ x_h \circ x_i &\mapsto \text{ro}_{h \otimes i}(x_h \circ x_i) = \text{ro}_h(x_h) \circ \text{ro}_i(x_i), \end{aligned}$$

As one can see from the expression for (3), the two terms  $(*)$  and  $(**)$  are switched compared to (1). Apart from this switch (and the corresponding switch in the signatures of the dynamics maps), we can see that there is a moral correspondence between the Moore machines in the functorial stacking axiom.

**Example 30.24.** We want to show that **LTI**, equipped with the defined stacking operations, is almost a functorial stacking semicategory, but not quite. Given morphisms  $f : l \rightarrow m, h : m \rightarrow n, g : o \rightarrow p, i : p \rightarrow q$ , we would need to have

$$(f \circledast h) \otimes (g \circledast i) = (f \otimes g) \circledast (h \otimes i).$$

This, however, is not true. Let's see this by looking at the first matrix component of the LTI system. On one hand we have:

$$\mathbf{A}_{(f \circledast h) \otimes (g \circledast i)} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{B}_h \mathbf{C}_f & \mathbf{A}_h & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_g & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{B}_i \mathbf{C}_g & \mathbf{A}_i \end{bmatrix}. \quad (60)$$

On the other hand we have:

$$\mathbf{A}_{(f \otimes g) \circledast (h \otimes i)} = \begin{bmatrix} \mathbf{A}_f & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_g & \mathbf{0} & \mathbf{0} \\ \mathbf{B}_h \mathbf{C}_f & \mathbf{0} & \mathbf{A}_h & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_i \mathbf{C}_g & \mathbf{0} & \mathbf{A}_i \end{bmatrix}. \quad (61)$$

These two are different, and will therefore describe different systems. However, the two matrices just differ by two permutations, which can be expressed via an invertible linear transformation  $\mathbf{T}$  as follows:

$$\mathbf{A}_{(f \otimes g) \circledast (h \otimes i)} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \cdot \mathbf{A}_{(f \circledast h) \otimes (g \circledast i)} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (62)$$

It turns out that  $(f \circledast h) \otimes (g \circledast i)$  and  $(f \otimes g) \circledast (h \otimes i)$  are equivalent systems (Def. 16.25), even though they are not equal. In particular, although **LTI** is not a functorial stacking semicategory, by Lemma 16.32  $(f \circledast h) \otimes (g \circledast i)$  and  $(f \otimes g) \circledast (h \otimes i)$  have the same action, and hence **LTI** is, in one sense, “morally” functorial.



## 30.6. Functorial stacking categories

**Definition 30.25** (Functorial stacking category)

A *functorial stacking category* is a stacking category where the two stacking operations  $\otimes$  and  $\circ$  are the two components of a functor

$$\otimes : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}. \quad (63)$$

In infix notation, this means that, in addition to the equation

$$(f \circ h) \otimes (g \circ i) = (f \otimes g) \circ (h \otimes i) \quad (64)$$

it is now also the case that  $\otimes$  is compatible with identity morphisms:

$$\text{id}_X \otimes \text{id}_Y = \text{id}_{X \otimes Y} \quad (65)$$

for all objects  $X, Y$  of  $\mathbf{C}$ .

**Remark 30.26.** Note that a functorial stacking category is *not* the same thing as a functorial stacking semicategory where the semicategory in question happens to be a category.

### Examples

This remark implies that having proved that  $\langle \mathbf{Set} \rangle$ ,  $\langle \mathbf{Pos} \rangle$ ,  $\langle \mathbf{Rel} \rangle$ ,  $\langle \mathbf{DP} \rangle$  are functorial stacking *semicategories*, we need to make an additional step to prove that they are also functorial stacking *categories*.

**Exercise 46.** [The usual suspects are functorial stacking categories] Prove that  $\langle \mathbf{Set} \rangle$ ,  $\langle \mathbf{Pos} \rangle$ ,  $\langle \mathbf{Rel} \rangle$ ,  $\langle \mathbf{DP} \rangle$  are functorial stacking categories. Hint: You have to define the functor  $\otimes$  and check that it satisfies the two equations in Def. 30.25.

*See solution on page 459.*

## 30.7. Monoidal stacking (semi)categories

**Definition 30.27** (Strict monoidal stacking semicategory)

A *strict monoidal stacking semicategory* is a stacking semicategory  $\langle \mathbf{C}, \otimes, \boxtimes \rangle$  with

Constituents

1. an object  $\mathbf{1} \in \text{Ob}_{\mathbf{C}}$ , called the *monoidal unit*

Conditions

1. For any object  $X$  of  $\mathbf{C}$ ,

$$X \otimes \mathbf{1} = X \quad \text{and} \quad \mathbf{1} \otimes X = X. \quad (66)$$

2. The monoidal unit  $\mathbf{1}$  has an identity morphism  $\text{id}_{\mathbf{1}}$ , and for any morphism  $f : X \rightarrow Y$ ,

$$f \otimes \text{id}_{\mathbf{1}} = f \quad \text{and} \quad \text{id}_{\mathbf{1}} \otimes f = f. \quad (67)$$

**Example 30.28.** Consider the associative stacking category from Example 30.6, where objects are integers and stacking of objects is taking their maximum. There is no possible monoidal unit here: it would have to be a neutral element for the operation “max”, but such does not exist for  $\mathbb{Z}$ . However, we could modify this example, and replace  $\mathbb{Z}$  with a bounded set of numbers, such as  $\mathbb{N}$ . Then the smallest number in  $\mathbb{N}$ , namely 0, serves a neutral element for “max” and provides a monoidal unit  $\mathbf{1}$ .

**Example 30.29.** Consider the associative stacking category from Example 30.7. A monoidal unit would need to be a neutral element for list concatenation. This would be the empty list. In Example 30.7 we specified that objects are only non-empty lists, hence we do not have a strict monoidal semicategory. However, this example can be easily adjusted to include also the empty list, in which case we *do* obtain a strict monoidal semicategory.

**Example 30.30.** We can look at **Moo** and ask whether it is a strict monoidal semicategory. The monoidal unit is given by the object

$$\mathbf{1} = \langle \rangle.$$

Its identity morphism is the Moore machine

$$\text{id}_{\mathbf{1}} = \langle \langle \rangle, \langle \rangle, \langle \rangle, \text{dyn}_{\mathbf{1}}, \text{ro}_{\mathbf{1}}, \langle \rangle \rangle,$$

where

$$\begin{aligned} \text{dyn}_{\mathbf{1}} : \langle \rangle \circ_{\mathbf{q}} \langle \rangle &\rightarrow \langle \rangle, \\ \langle \rangle \circ_{\mathbf{q}} \langle \rangle &\mapsto \langle \rangle, \end{aligned}$$

and

$$\begin{aligned} \text{ro}_{\mathbf{1}} : \langle \rangle &\rightarrow \langle \rangle, \\ \langle \rangle &\mapsto \langle \rangle. \end{aligned}$$

Clearly,  $A \circ_{\mathbf{q}} \langle \rangle = \langle \rangle \circ_{\mathbf{q}} A = A$  for every  $A \in \text{Ob}_{\mathbf{Moo}}$ . Furthermore, consider a Moore machine  $f : \mathbf{U} \rightarrow \mathbf{Y}$  with

$$f = \langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle.$$

One has:

$$\begin{aligned} f \otimes \text{id}_{\mathbf{1}} &= \langle \mathbf{U} \circ_{\mathbf{q}} \langle \rangle, \mathbf{X} \circ_{\mathbf{q}} \langle \rangle, \mathbf{Y} \circ_{\mathbf{q}} \langle \rangle, \text{dyn}_{f \otimes \text{id}_{\mathbf{1}}}, \text{ro}_{f \otimes \text{id}_{\mathbf{1}}}, \text{st} \circ_{\mathbf{q}} \langle \rangle \rangle \\ &= \langle \mathbf{U}, \mathbf{X}, \mathbf{Y}, \text{dyn}, \text{ro}, \text{st} \rangle = f, \end{aligned}$$

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

where we used

$$\begin{aligned} \text{dyn}_{f \otimes \text{id}_1} : \mathbf{U} \circledast \mathbb{D} \mathbb{D} \mathbb{D} \mathbf{X} \circledast \mathbb{D} \mathbb{D} &\rightarrow \mathbf{X} \circledast \mathbb{D} \mathbb{D} \\ u \circledast \langle \rangle \circledast x \circledast \langle \rangle &\mapsto \text{dyn}(u, x) \circledast \text{dyn}_1(\langle \rangle, \langle \rangle) = \text{dyn}(u, x) \end{aligned}$$

and

$$\begin{aligned} \text{ro}_{f \otimes \text{id}_1} : \mathbf{X} \circledast \mathbb{D} \mathbb{D} &\rightarrow \mathbf{Y} \circledast \mathbb{D} \mathbb{D} \\ x \circledast \langle \rangle &\mapsto \text{ro}(x) \circledast \text{ro}_1(\langle \rangle) = \text{ro}(x) \end{aligned}$$

to show the equivalences  $\text{dyn} = \text{dyn}_{f \otimes \text{id}_1}$  and  $\text{ro} = \text{ro}_{f \otimes \text{id}_1}$ . The argument for  $\text{id}_1 \otimes f$  follows analogously.

**Definition 30.31** (Strict monoidal stacking category)

A *strict monoidal stacking category* is a strict monoidal stacking semicategory for which all identity morphisms exist.

**Example 30.32.** **LTI**, equipped with previously described stacking operations and an appropriate unit, is a strict monoidal stacking category. The unit is given by the object 0, and its identity morphism is given by the LTI system

$$\text{id}_1 = \langle \mathbf{0}^{0 \times 1}, \mathbf{0}^{0 \times 0}, \mathbf{0}^{0 \times 0}, \mathbf{0}^{0 \times 0}, \mathbf{0}^{0 \times 0} \rangle.$$

On the side of objects, clearly  $l + 0 = 0 + l = l$  for any object  $l \in \text{Ob}_{\text{LTI}}$ . Consider  $f : l \rightarrow m$ . On the side of morphisms we have:

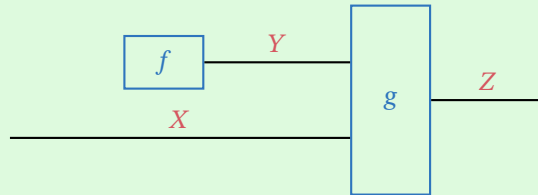
$$\begin{aligned} f \otimes \text{id}_1 &= \left\langle \begin{bmatrix} \text{st} \\ \mathbf{0}^{0 \times 1} \end{bmatrix}, \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0}^{0 \times 0} \end{bmatrix}, \begin{bmatrix} \mathbf{B} & \mathbf{0} \\ \mathbf{0} & \mathbf{0}^{0 \times 0} \end{bmatrix}, \begin{bmatrix} \mathbf{C} & \mathbf{0} \\ \mathbf{0} & \mathbf{0}^{0 \times 0} \end{bmatrix}, \begin{bmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{0}^{0 \times 0} \end{bmatrix} \right\rangle \\ &= \langle \text{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle \end{aligned}$$

Similarly:

$$\begin{aligned} \text{id}_1 \otimes f &= \left\langle \begin{bmatrix} \mathbf{0}^{0 \times 1} \\ \text{st} \end{bmatrix}, \begin{bmatrix} \mathbf{0}^{0 \times 0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A} \end{bmatrix}, \begin{bmatrix} \mathbf{0}^{0 \times 0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B} \end{bmatrix}, \begin{bmatrix} \mathbf{0}^{0 \times 0} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} \end{bmatrix}, \begin{bmatrix} \mathbf{0}^{0 \times 0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} \end{bmatrix} \right\rangle \\ &= \langle \text{st}, \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D} \rangle \end{aligned}$$

**Graded exercise H.3** (StringDiagrams)

The following is a string diagram which can represent a composition of morphisms (having certain types) in any given monoidal category  $\langle \mathbf{C}, \otimes, \mathbf{1} \rangle$  (We read the diagram left-to-right for series composition, and top-to-bottom for parallel composition). The resulting morphism described by the total diagram – call it  $h$  – is one of the type  $h : X \rightarrow Z$ .



In each part of this exercise, we will specify a monoidal category and specific objects and morphisms to plug into the variables  $X, Y, Z$  and  $f, g$  in this diagram. Your task is to compute the respective resulting morphism  $h$  as dictated by the diagram.

1. In this part, let  $\langle \mathbf{C}, \otimes, \mathbf{1} \rangle$  be the monoidal category where  $\mathbf{C}$  is the category **Set** of sets and functions,  $\otimes$  is the cartesian product of sets and functions, and  $\mathbf{1}$  is a chosen 1-element set that we denote by  $\mathbf{1}$ . In the string diagram above, let  $X = \mathbb{Z}$ ,  $Y = \mathbb{N}$ , and  $Z = \mathbb{Z}$ . Furthermore, let

$$f : \mathbf{1} \rightarrow Y \quad (68)$$

be the function with  $f(\bullet) = 5$ , and let

$$\begin{aligned} g : \mathbb{N} \times \mathbb{Z} &\rightarrow \mathbb{Z}, \\ \langle y, x \rangle &\mapsto y + x. \end{aligned} \quad (69)$$

Compute the composite morphism  $h$  described by the string diagram in this case.

2. In this part, let  $\langle \mathbf{C}, \otimes, \mathbf{1} \rangle$  be the monoidal category where  $\mathbf{C}$  is the category **Rel** of sets and relations,  $\otimes$  is the cartesian product of sets and relations, and  $\mathbf{1}$  is a chosen 1-element set that we again denote by  $\mathbf{1}$ . In the string diagram, let  $X = \mathbb{Z}$ ,  $Y = \mathbb{Z}$ , and  $Z = \mathbb{Z}$ . Furthermore, let  $f : \mathbf{1} \rightarrow \mathbb{Z}$  be the relation

$$f = \{ \langle \bullet, y \rangle \in \mathbf{1} \times \mathbb{Z} \mid y \text{ is an even number} \} \quad (70)$$

and let  $g : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$  be the relation

$$g = \{ \langle \langle y, x \rangle, z \rangle \in (\mathbb{Z} \times \mathbb{Z}) \times \mathbb{Z} \mid y = x = z \}. \quad (71)$$

Compute the composite morphism  $h$  described by the string diagram in this case.

3. In this part, let  $\langle \mathbf{C}, \otimes, \mathbf{1} \rangle$  be the monoidal category where  $\mathbf{C}$  is the category of real vector spaces and real linear maps,  $\otimes$  is the direct sum, and  $\mathbf{1}$  is the 0-dimensional real vector space  $\{0\}$ . In the string diagram, let  $X = Y = Z = \mathbb{R}^3$ . Furthermore, let

$$f : \{0\} \rightarrow \mathbb{R}^3 \quad (72)$$

be the linear function with  $f(0) = 0$ , and let

$$\begin{aligned} g : \mathbb{R}^3 \oplus \mathbb{R}^3 &\rightarrow \mathbb{R}^3, \\ \langle y, x \rangle &\mapsto y + x. \end{aligned} \quad (73)$$

Compute the composite morphism  $h$  described by the string diagram in this case.

### 30.8. Strict monoidal (semi)categories

**Definition 30.33** (Strict monoidal semicategory)

A *strict monoidal semicategory* is a strict monoidal stacking semicategory that is functorial stacking.

The following definition is analogous to the definition of a functorial stacking semicategory; the one difference is that we start with a functorial stacking category.

**Definition 30.34**

A *strict monoidal category* is a strict monoidal stacking category which is a functorial stacking category  $\langle \mathbf{C}, \otimes \rangle$ .

Unpacked somewhat, this means there is a stacking functor

$$\otimes : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C} \quad (74)$$

and an object  $\mathbf{1} \in \text{Ob}_{\mathbf{C}}$ , the *monoidal unit*, satisfying

$$X \otimes \mathbf{1} = X \quad \text{and} \quad \mathbf{1} \otimes X = X \quad (75)$$

and

$$f \otimes \text{id}_{\mathbf{1}} = f \quad \text{and} \quad \text{id}_{\mathbf{1}} \otimes f = f. \quad (76)$$

for any object  $X$  and any morphism  $f$  in  $\mathbf{C}$ .



## 31. Crossing wires

31.1 Morphism-representable wire crossings . . . . .	450
31.2 A strict monoidal category of permutations . . . . .	452
31.3 Symmetric stacking semicategories . . . . .	454

### 31.1. Morphism-representable wire crossings

#### Definition 31.1

A *symmetric stacking category* is an associative stacking category  $\langle \mathbf{C}, \otimes, \otimes \rangle$  with

#### Constituents

1. For any two objects  $X, Y \in \text{Ob}_{\mathbf{C}}$  there exists an isomorphism

$$\sigma_{X,Y} : X \otimes Y \rightarrow Y \otimes X, \quad (1)$$

called the *braiding*.

#### Conditions

1. *Naturality*: For any morphisms  $f : X \rightarrow Z, g : Y \rightarrow U$ , the diagram

$$\begin{array}{ccc} X \otimes Y & \xrightarrow{\sigma_{X,Y}} & Y \otimes X \\ f \otimes g \downarrow & & \downarrow g \otimes f \\ Z \otimes U & \xrightarrow{\sigma_{Z,U}} & U \otimes Z \end{array} \quad (2)$$

commutes.

2. *Compatibility with nesting*:

$$\begin{array}{ccc} & (X \otimes Y) \otimes Z & \\ & \swarrow \quad \searrow & \\ X \otimes (Y \otimes Z) & & (Y \otimes X) \otimes Z \\ \downarrow \sigma_{X,Y \otimes Z} & & \parallel \\ (Y \otimes Z) \otimes X & & Y \otimes (X \otimes Z) \\ & \swarrow \quad \searrow & \\ & Y \otimes (Z \otimes X) & \end{array} \quad \begin{array}{l} \sigma_{X,Y} \otimes \text{id}_Z \\ \text{id}_Y \otimes \sigma_{X,Z} \end{array} \quad (3)$$

3. *Symmetry*: For all  $X, Y \in \text{Ob}_{\mathbf{C}}$ ,

$$\sigma_{X,Y} \circ \sigma_{Y,X} = \text{id}_{X \otimes Y}. \quad (4)$$

We note that the braidings  $\sigma_{X,Y}$  of a symmetric stacking category induce source and target permutation operations as in the Def. 31.12 of symmetric stacking semicategory. To see how this works, recall that every permutation can be written in a (non-unique) way as a composition of transpositions.

**Lemma 31.2.** Every symmetric stacking category  $\langle \mathbf{C}, \otimes, \otimes, \sigma \rangle$ , when equipped with the source and target permutation operations induced by the braidings  $\sigma_{X,Y}$ , is a symmetric stacking semicategory.

### Symmetric strict monoidal categories

#### Definition 31.3 (Symmetric strict monoidal category)

A *symmetric strict monoidal category* is a strict monoidal category  $\langle \mathbf{C}, \otimes, \mathbf{1} \rangle$  which is also a symmetric stacking category. This means that  $\mathbf{C}$  is equipped

with braidings

$$\sigma_{X,Y} : X \otimes Y \rightarrow Y \otimes X \quad (5)$$

which satisfy the conditions of naturality, compatibility with nesting, and symmetry.

## Props

### Definition 31.4

A *prop* is a symmetric strict monoidal category  $\mathbf{C}$  where the collection of objects is the natural numbers  $\mathbb{N}$ , the monoidal product on objects is addition of natural numbers, and the monoidal unit is  $0 \in \mathbb{N}$ .

**Example 31.5.** There is a prop **FinSet** where

- ▷ the set of morphisms from  $m$  to  $n$  (for  $m, n \in \mathbb{N}$ ) is defined to be the set of functions from  $\{1, \dots, m\}$  to  $\{1, \dots, n\}$ ;
- ▷ composition is the usual composition of functions;
- ▷ identity morphisms are identity functions;
- ▷ the monoidal product of functions  $f : \{1, \dots, m\} \rightarrow \{1, \dots, n\}$  and  $f' : \{1, \dots, m'\} \rightarrow \{1, \dots, n'\}$  is the “disjoint union”

$$f + f' : \{1, \dots, m + m'\} \rightarrow \{1, \dots, n + n'\}. \quad (6)$$

**Example 31.6.** We define a prop  $\mathbf{Mat}_{\mathbb{R}}$  where:

- ▷ morphisms from  $m$  to  $n$  are  $n \times m$  matrices with entries in  $\mathbb{R}$  (we also allow zero-dimensional matrices);
- ▷ composition is matrix multiplication ;
- ▷ identity morphisms are identity matrices;
- ▷ the monoidal product of matrices  $\mathbf{A} : m \rightarrow n$  and  $\mathbf{B} : m' \rightarrow n'$  is

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{B} \end{bmatrix} : m + m' \rightarrow n + n'. \quad (7)$$

**Example 31.7.** We define a prop  $\mathbf{LinRel}_{\mathbb{R}}$  where:

- ▷ morphisms from  $m$  to  $n$  are linear relations  $\mathbb{R}^m \rightarrow \mathbb{R}^n$  (in other words, linear subspaces of  $\mathbb{R}^m \oplus \mathbb{R}^n$ );
- ▷ composition is composition of relations;
- ▷ identity morphisms are identity relations;
- ▷ the monoidal product of linear relations  $R : m \rightarrow n$  and  $S : m' \rightarrow n'$  is  $R \oplus S : \mathbb{R}^m \oplus \mathbb{R}^{m'} \rightarrow \mathbb{R}^n \oplus \mathbb{R}^{n'}$ , where

$$R \oplus S = \{ \langle \langle v, v' \rangle, \langle w, w' \rangle \rangle \mid \langle v, w \rangle \in R \text{ and } \langle v', w' \rangle \in S \}. \quad (8)$$



## 31.2. A strict monoidal category of permutations

Recall that a *permutation* of order  $n \in \mathbb{N}$  is a bijective function from the set  $\{1, 2, \dots, n\}$  to itself. For fixed  $n \in \mathbb{N}$ , the set of permutations of order  $n$  has a group structure, where the composition operation is composition of functions. We call this group  $\mathbf{Perms}(n)$ . For the special case of  $n = 0$ , we interpret  $\mathbf{Perms}(0)$  as containing the unique function from the empty set  $\emptyset$  to itself; this function is bijective.

There are several typical notations for indicating permutations. We will use the following one, and illustrate it by example. If, for instance,  $\sigma \in \mathbf{Perms}(3)$  is the permutation

$$\begin{aligned} \{1, 2, 3\} &\rightarrow \{1, 2, 3\} \\ 1 &\mapsto 2 \\ 2 &\mapsto 3 \\ 3 &\mapsto 1 \end{aligned}$$

then we indicate this by writing  $\sigma = (3, 1, 2)$ . The way we think of this notation is that we start with  $(1, 2, 3)$  and then apply  $\sigma$  to each entry. In other words, in general, our notation for some  $\sigma \in \mathbf{Perms}(n)$  is  $(\sigma(1), \dots, \sigma(n))$ .

To help us talk about permuting inputs and output of morphisms in stacking semicategories, we make the following definitions.

### Definition 31.8 (Permutation category)

The *permutation category*  $\mathbf{Pm}$  is

1. *Objects*: the natural numbers  $\mathbb{N}$ ;
2. *Morphisms*: permutations in  $\mathbf{Perms}(n)$ , for all  $n \in \mathbb{N}$ .
3. *Composition*: composition of permutations;
4. *Identities*: identity functions (which are permutations). We will denote the identity permutation in  $\mathbf{Perms}(n)$  by  $1_n$ .

### Definition 31.9

For any two objects  $n, m \in \mathbb{N}$  of  $\mathbf{Pm}$ , let

$$n \otimes m := n + m, \tag{9}$$

and for any two morphisms  $\sigma = (\sigma(1), \dots, \sigma(n))$ ,  $\pi = (\pi(1), \dots, \pi(m))$  of  $\mathbf{Pm}$ , let

$$\sigma \otimes \pi := (\sigma(1), \dots, \sigma(n), \pi(1) + n, \dots, \pi(m) + n). \tag{10}$$

**Lemma 31.10.** The operations  $\otimes$  and  $\otimes$  make  $\mathbf{Pm}$  into a functorial stacking category  $\langle \mathbf{Pm}, \otimes \rangle$ .

**Lemma 31.11.**  $\langle \mathbf{Pm}, \otimes \rangle$  is a strict monoidal category, with unit  $\mathbf{1}$  given by  $0 \in \mathbb{N}$ .

## Block permutations

We will also need to describe permutations that permute “blocks” of elements of  $\{1, \dots, n\}$ . Suppose we are given natural numbers  $k_1, \dots, k_r$  such that  $k_1 + \dots + k_r = n$ . We call this a partition of  $n$  and we think of it as partitioning the set  $\{1, \dots, n\}$  into blocks of size  $k_j$ ,  $1 \leq j \leq r$ .

Given a permutation  $\sigma \in \mathbf{Perms}(r)$ , there is a way to turn it into a *block permutation*  $\sigma_*(k_1, \dots, k_r) \in \mathbf{Perms}(r)$ .

We'll start with an example, then give a general formula. Suppose  $n = 5$ ,  $r = 2$ , and  $k_1 = 3$  and  $k_2 = 2$ . Let  $\sigma = (2, 1) \in \mathbf{Perms}(2)$ . Then

$$\sigma_*(k_1, k_2) = \sigma_*(3, 2) = (4, 5, 1, 2, 3) \in \mathbf{Perms}(5).$$

What is happening is that we start with  $(1, 2, 3, 4, 5)$ , we split it into the blocks  $((1, 2, 3), (4, 5))$  of length 3 and 2 respectively, we apply  $\sigma$  to the blocks to get  $((4, 5), (1, 2, 3))$ , and then we forget the subdivision into blocks, obtaining  $(4, 5, 1, 2, 3)$ .

The general formula to compute  $\sigma_*(k_1, \dots, k_r)$  is that, for each  $j = 1, \dots, r$ , in the sequence  $(\sigma(1), \dots, \sigma(r))$  we replace  $\sigma(j)$  with the sequence

$$k_1 + \dots + k_{j-1} + 1, \quad k_1 + \dots + k_{j-1} + 2, \quad \dots, \quad k_1 + \dots + k_{j-1} + k_j.$$

### 31.3. Symmetric stacking semicategories

Let  $\mathbf{C}$  be an associative stacking semicategory, and consider the morphisms

$$f \otimes g : X \otimes Y \rightarrow Z \otimes U \quad (11)$$

and

$$h \otimes i : U \otimes Z \rightarrow W \otimes V. \quad (12)$$

If  $Z \otimes U \neq U \otimes Z$ , then the morphisms  $f \otimes g$  and  $h \otimes i$  are, in our model, technically not composable. However, in some examples, we will in fact want to be able to compose such morphisms, because in those cases  $Z \otimes U$  and  $U \otimes Z$  are essentially equal, even if they aren't equal on the nose.

For example,  $f, g, h, i$  might be systems, represented graphically by boxes with wires for input and output. The stacked system  $f \otimes g$  will have output wires for  $Z$  and  $U$ , and the stacked system  $h \otimes i$  will have input wires for  $U$  and  $Z$ . In some applications we are often indeed able to connect the one output wire labeled with  $Z$  to the input wire also labeled with  $Z$ , and similarly for  $U$ .

To model this for stacking semicategories, we introduce operations that correspond to crossing (or permuting) wires, both on the input and output sides of a morphism. For instance, if  $f$  is a morphism that can be written as having the type

$$f : X_1 \otimes X_2 \otimes X_3 \rightarrow Y_1 \otimes Y_2 \quad (13)$$

then we want to be able to use any permutation  $\sigma \in \mathbf{Perms}(3)$  as a recipe to permute the input wires of  $f$ . Specifically, given for example  $\sigma = (3, 1, 2) \in \mathbf{Perms}(3)$ , we want a corresponding function

$$\sigma_s : \mathbf{Hom}_{\mathbf{C}}(X_1 \otimes X_2 \otimes X_3; Y_1 \otimes Y_2) \rightarrow \mathbf{Hom}_{\mathbf{C}}(X_3 \otimes X_1 \otimes X_2; Y_1 \otimes Y_2) \quad (14)$$

which maps any morphism  $f$  of the type (13) to a morphism that corresponds to  $f$  with the its input wires permuted according to  $\sigma$ .

One subtlety here is that, for instance, the representation of the source object of the morphism (13) as a three-fold stacking is not unique. For example, it could be the case that there exist objects  $Z_1$  and  $Z_2$  such that

$$X_1 \otimes X_2 \otimes X_3 = Z_1 \otimes Z_2. \quad (15)$$

In this case,  $f$  can also be viewed as having the type

$$f : Z_1 \otimes Z_2 \rightarrow Y_1 \otimes Y_2, \quad (16)$$

and in this representation, it is no longer appropriate to permute the inputs of  $f$  using a permutation in  $\mathbf{Perms}(3)$  but rather one in  $\mathbf{Perms}(2)$ .

Thus the representation of a morphism as a box with some number of input wires and some number of output wires is just one of potentially many different representations of the very same morphism and the application of a permutation to a morphism is always relative to a specific representation.

One special case of situations like the one indicated in (15) is when we use parentheses to group together an  $n$ -fold stacking of objects into “blocks” of size  $k_1, \dots, k_r$ . For example, given a stacking of objects

$$X_1 \otimes X_2 \otimes X_3 \otimes X_4 \otimes X_5 \quad (17)$$

we can use parentheses, as below, to think of (15) as a stacking of just *two* objects:

$$(X_1 \otimes X_2 \otimes X_3) \otimes (X_4 \otimes X_5). \quad (18)$$

In other words, there are objects  $Z_1 = X_1 \otimes X_2 \otimes X_3$  and  $Z_2 = X_4 \otimes X_5$  such that

$$X_1 \otimes X_2 \otimes X_3 \otimes X_4 \otimes X_5 = Z_1 \otimes Z_2. \quad (19)$$

**Definition 31.12** (Symmetric stacking semicategory)

A symmetric stacking semicategory is an associative stacking semicategory  $\mathbf{C}$  with:

Constituents

▷ Functions

$$\sigma_s : \text{Hom}_{\mathbf{C}}(\bigotimes_{i=1}^n X_i; Y) \rightarrow \text{Hom}_{\mathbf{C}}(\bigotimes_{i=1}^n X_{\sigma^{-1}(i)}; Y) \quad (20)$$

and

$$\sigma_t : \text{Hom}_{\mathbf{C}}(X; \bigotimes_{i=1}^n Y_i) \rightarrow \text{Hom}_{\mathbf{C}}(X; \bigotimes_{i=1}^n Y_{\sigma(i)}) \quad (21)$$

for every  $\sigma \in \mathbf{Perms}(n)$ ,  $n \geq 1$ . We call these functions *source permutations* and *target permutations*, respectively.

Conditions

▷ Compatibility with permutation composition:

The equations

$$\sigma_s \circ \pi_s = (\pi \circ \sigma)_s \quad \text{and} \quad \sigma_t \circ \pi_t = (\sigma \circ \pi)_t \quad (22)$$

hold for all  $\sigma, \pi \in \mathbf{Perms}(n)$ ,  $n \geq 2$ .

▷ Compatibility with permutation identities:

For any  $n \in \mathbb{N}$ , if  $\sigma = 1_n$  is the identity permutation in  $\mathbf{Perms}(n)$ , then

$$(1_n)_s : \text{Hom}_{\mathbf{C}}(\bigotimes_{i=1}^n X_i; Y) \rightarrow \text{Hom}_{\mathbf{C}}(\bigotimes_{i=1}^n X_i; Y) \quad (23)$$

and

$$(1_n)_t : \text{Hom}_{\mathbf{C}}(X; \bigotimes_{i=1}^n Y_i) \rightarrow \text{Hom}_{\mathbf{C}}(X; \bigotimes_{i=1}^n Y_i) \quad (24)$$

are the identity functions.

▷ Compatibility with block permutations:

For any  $n \in \mathbb{N}$  and any partition  $n = k_1 + \dots + k_r$ , if  $Z_1, \dots, Z_r$  are equal to the  $r$  factors in a “partitioned” stacking of the form

$$X = (X_1 \otimes \dots \otimes X_{k_1}) \otimes (X_{k_1+1} \otimes \dots \otimes X_{k_1+k_2}) \otimes \dots \otimes (X_{k_1+\dots+k_{r-1}+1} \otimes \dots \otimes X_n)$$

then for any  $\sigma \in \mathbf{Perms}(r)$ , the source permutation map  $\sigma_s$  with respect to this  $r$ -fold stacking is equal to the source permutation map  $(\sigma_*(k_1, \dots, k_r))_s$  with respect to the  $n$ -fold stacking

$$X = X_1 \otimes \dots \otimes X_n.$$

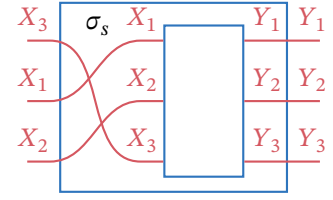
(Here,  $\sigma_*(k_1, \dots, k_r)$  is the block permutation induced by  $\sigma$  and the partition  $n = k_1 + \dots + k_r$ .)

The analogous statement for target permutation maps is also required to hold.

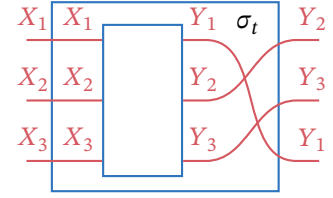
▷ Compatibility with semicategory composition:

For all  $f \in \text{Hom}_{\mathbf{C}}(X; \bigotimes_{i=1}^n Y_i)$  and  $g \in \text{Hom}_{\mathbf{C}}(\bigotimes_{i=1}^n Y_{\sigma(i)}; Z)$

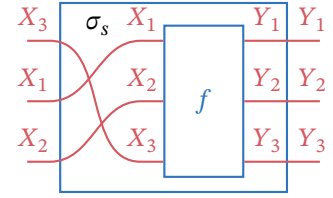
$$\sigma_t(f) \circ g = f \circ \sigma_s(g). \quad (25)$$



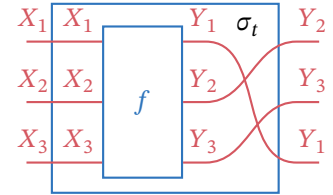
**Figure 1.** A diagram for a source permutation map.



**Figure 2.** A diagram for a target permutation map.



**Figure 3.** A source permutation applied to a morphism  $f : X \rightarrow Y$  whose source has a factorization  $X = X_1 \otimes X_2 \otimes X_3$  and whose target has a factorization  $Y = Y_1 \otimes Y_2 \otimes Y_3$ .



**Figure 4.** The target permutation using the same permutation as in Fig. 1 and applied to the same morphism.

▷ *Compatibility with stacking:*

For all  $f \in \text{Hom}_{\mathbf{C}}(\bigotimes_{i=1}^n X_i; Y_1)$ ,  $g \in \text{Hom}_{\mathbf{C}}(\bigotimes_{i=n+1}^{n+m} X_i; Y_2)$ ,

$$\sigma_s(f) \otimes \pi_s(g) = (\sigma \otimes \pi)_s(f \otimes g) \quad (26)$$

holds for all  $\sigma \in \mathbf{Perms}(n)$ ,  $\pi \in \mathbf{Perms}(m)$ , where  $\otimes$  is the stacking operation in  $\mathbf{Pm}$ .

Similarly, for all  $f \in \text{Hom}_{\mathbf{C}}(X_1; \bigotimes_{i=1}^n Y_i)$  and  $g \in \text{Hom}_{\mathbf{C}}(X_2; \bigotimes_{i=n+1}^{n+m} Y_i)$ ,

$$\sigma_t(f) \otimes \pi_t(g) = (\sigma \otimes \pi)_t(f \otimes g) \quad (27)$$

holds for all  $\sigma \in \mathbf{Perms}(n)$ ,  $\pi \in \mathbf{Perms}(m)$ .

▷ *Sliding:*

Given  $f_i : X_i \rightarrow Y_i$ ,  $1 \leq i \leq n$ , it holds that

$$\sigma_t(\bigotimes_{i=1}^n f_i) = \sigma_s(\bigotimes_{i=1}^n f_{\sigma(i)}) \quad (28)$$

and

$$\sigma_s(\bigotimes_{i=1}^n f_i) = \sigma_t(\bigotimes_{i=1}^n f_{\sigma^{-1}(i)}). \quad (29)$$

In terms of diagrams, the condition of compatibility with composition (25) is illustrated in terms of diagrams in Fig. 5.

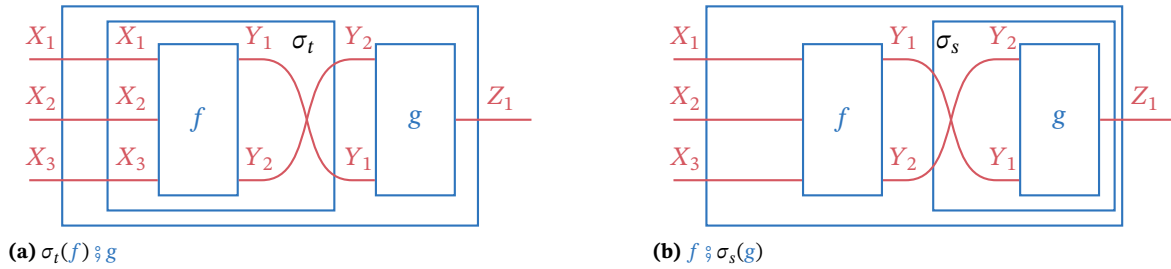


Figure 5.: Compatibility with composition.

In Fig. 6 the first equation (28) for compatibility with stacking is illustrated diagrammatically.

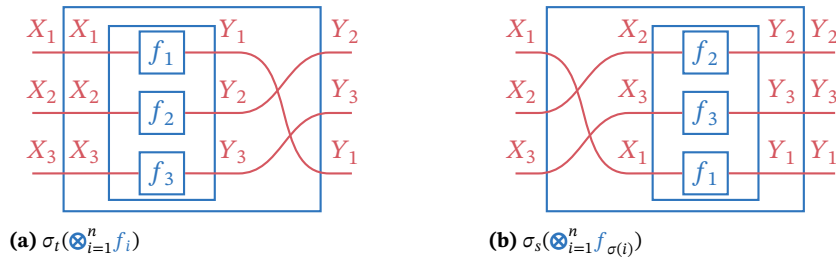


Figure 6.: Sliding.

Fig. 7 illustrates the first equation (22) for compatibility with permutation composition.

**Example 31.13.**  $\langle \mathbf{Set} \rangle$  is a symmetric stacking semicategory in a straightforward manner.

**Example 31.14.**  $\mathbf{LTI}$  with  $D = 0$  is a symmetric stacking semicategory.

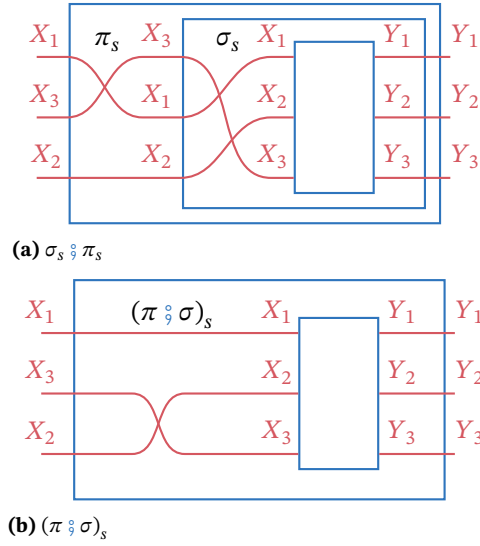


Figure 7.: Compatibility with composition.

**Lemma 31.15.** **Eff** is not a symmetric stacking semicategory.

*Proof.* Since **Eff** is not functorial stacking (see Lemma Lemma 30.22), we may safely conclude that **Eff** is not symmetric. Consider two morphisms  $F_1 : X_1 \rightarrow Y_1, F_2 : X_2 \rightarrow Y_2$  in **Eff** and  $\sigma$  a permutation. Then, in general permuting after the stacking of  $F_1 \otimes F_2$  is not equal to applying the permutation before the stacking of  $F_2 \otimes F_1$ . This is due to the difference in order of effects on  $\otimes$ .  $\square$

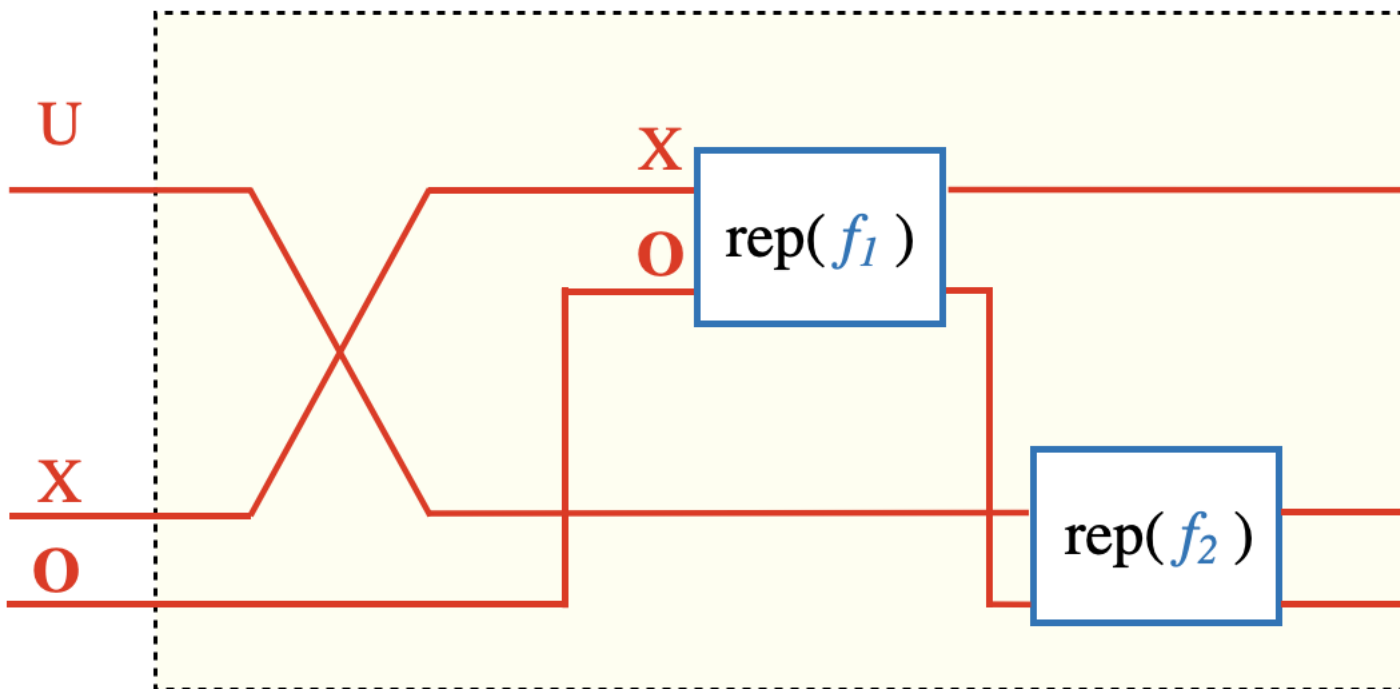
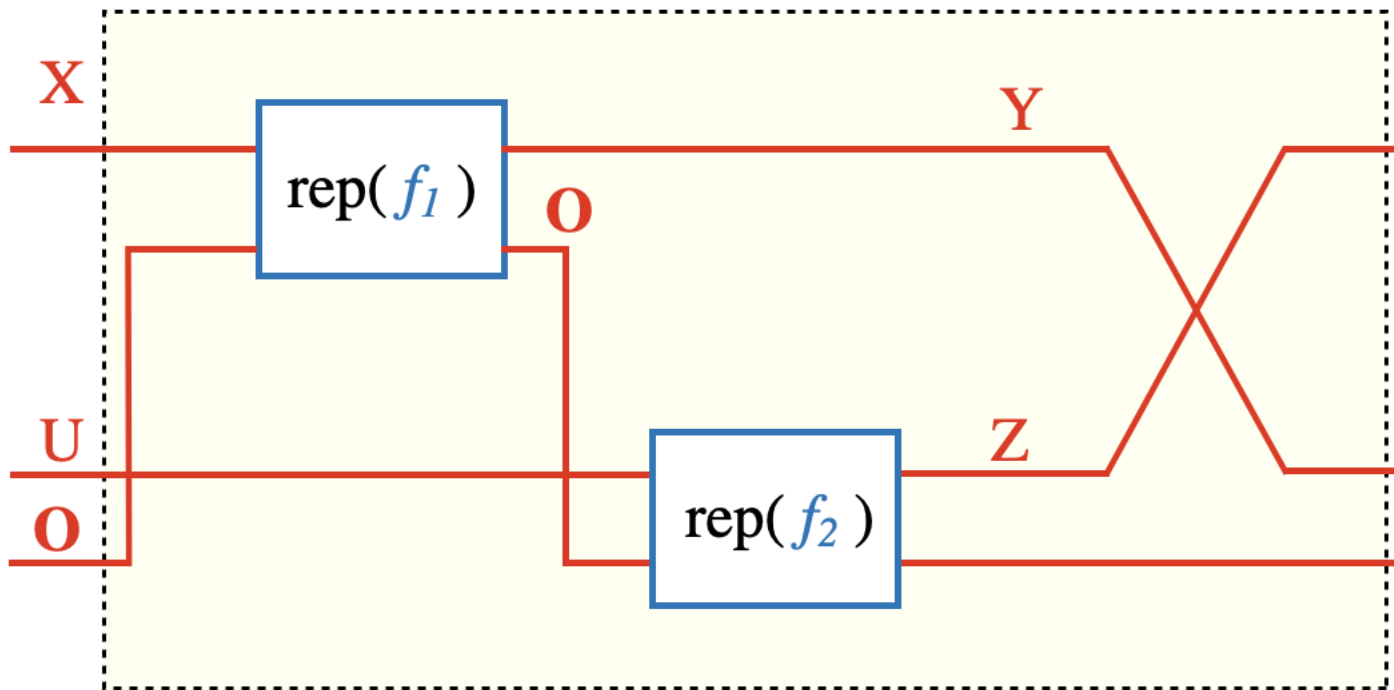


Figure 8.: Non-symmetry of effects

## Solutions to selected exercises

**Solution of Exercise 45.** To prove the statement we first check that the stacking operations satisfy Def. 30.2, we then show that they are compatible, and finally show associativity. The stacking operation on objects was already checked for  $\langle \mathbf{Set} \rangle$ . The stacking operation on morphisms clearly returns a valid relation. Furthermore, compatibility is satisfied:

$$\frac{R : \langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle \rightarrow_{\langle \mathbf{Rel} \rangle} \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle \quad S : \langle \mathbf{C}_1, \dots, \mathbf{C}_o \rangle \rightarrow_{\langle \mathbf{Rel} \rangle} \langle \mathbf{D}_1, \dots, \mathbf{D}_p \rangle}{R \otimes S : \langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle \circ_{\langle \rangle} \langle \mathbf{C}_1, \dots, \mathbf{C}_o \rangle \rightarrow_{\langle \mathbf{Rel} \rangle} \langle \mathbf{B}_1, \dots, \mathbf{B}_n \rangle \circ_{\langle \rangle} \langle \mathbf{D}_1, \dots, \mathbf{D}_p \rangle} \quad (30)$$

Finally, associativity for the operation on objects was already shown for  $\langle \mathbf{Set} \rangle$ .

**Solution of Exercise 46.**





# PART I. NATURALITY



---

32. Naturality	463
33. Adjunctions	479
34. Traced Monoidal categories	495

---

The *Sechseläuten* is a traditional spring holiday in the Zurich, Switzerland, usually happening on the 3rd monday of April. The old city guilds meet in the city center for a parade, climax of which is the burning of the “Böögg”, a snowman prepared with explosives, considered a weather oracle for the summer.

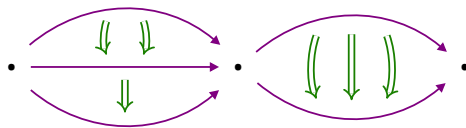




## 32. Naturality

We have seen that functors are “morphisms between categories”. It turns out that there is an important third layer to the world of categories: there are also “morphisms between functors”, and these are known as *natural transformations*.

To represent the three layers of structure involved in the world of categories, it is common to draw diagrams like this: Points repre-



sent categories, single arrows represent functors, and double arrows represent natural transformations.

32.1 Visual Example and Warmup . .	464
32.2 Natural transformations . . . . .	465
32.3 Morphisms in a Category of Functors . . . . .	469
32.4 Data migration . . . . .	473
32.5 More examples . . . . .	475
32.6 Yoneda Embedding Theorem & Lemma . . . . .	478

When one thinks about Switzerland, one of the symbols that comes to mind is usually cows. Recent statistics show that Switzerland has around 1.6 million cows (roughly one cow per five residents). The canton of Bern leads the rankings, being the one owning the most cows. Interestingly, in the canton of Appenzell Innerrhoden, the ratio of cows and humans is close to 1:1.

### 32.1. Visual Example and Warmup

Since mappings between functors may be rather abstract, we will first start with a concrete example. Imagine you are in Boston, and can go to New York or Providence. We may now define a category  $\mathbf{C}$  of cities, where morphisms are trips between the cities. Let  $\mathbf{D}$  be the category of pairs of real numbers under element-wise addition, and  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  be functors between them.  $F$  gives us the distance and altitude difference between our starting point (Boston) and our desired destination, and  $G$  gives us the gas price relative to our starting point.

Now, we can create a mapping  $\alpha$  mapping the distances to gas prices. Thinking about it, it relates the elements of the category of distances to those of the category of prices. If we write out our categories in typical diagrams, we notice that they commute (see Def. 13.15).

$$\begin{array}{ccc}
 F(\text{Boston}) & \xrightarrow{F(\text{trip})} & F(\text{New York}) \\
 \downarrow \alpha_{\text{Boston}} & & \downarrow \alpha_{\text{New York}} \\
 G(\text{Boston}) & \xrightarrow{G(\text{trip})} & G(\text{New York})
 \end{array}$$

In other words, for each pair of (distance, altitude) relative to Boston, we can compute the gas price of the trip.

## 32.2. Natural transformations

To formally define natural transformations, the general situation we will start from is when we have two functors  $F : \mathbf{C} \rightarrow \mathbf{D}$  and  $G : \mathbf{C} \rightarrow \mathbf{D}$ , sharing the same source and target, respectively. In the previous example above,  $F$  and  $G$  were database instances.

A natural transformation from  $F$  to  $G$  is then a kind of “map” that relates the two functors. How might one define such a thing?

First, let’s look at the situation only on the level of objects. Each object  $X$  of  $\mathbf{C}$  is mapped by  $F$  and  $G$  to an object  $F(X)$  and  $G(X)$  of  $\mathbf{D}$ , respectively. One straightforward way to relate  $F(X)$  to  $G(X)$  is to choose a morphism  $F(X) \rightarrow G(X)$  in  $\mathbf{D}$ . We call this morphism  $\alpha_X$ , using the subscript  $X$  since  $\alpha_X$  relates the respective images of the object  $X$  under  $F$  and  $G$ . If we choose such a morphism for each object in  $\mathbf{C}$ , then we have collection  $\{\alpha_X\}_{X \in \text{Ob}_{\mathbf{C}}}$  of morphisms in  $\mathbf{D}$ , indexed by the objects of  $\mathbf{C}$ .

Next, consider a morphism  $f : X \rightarrow Y$  in the category  $\mathbf{C}$ . Under the functor  $F$  it will be mapped to some morphism  $F(f) : F(X) \rightarrow F(Y)$  in  $\mathbf{D}$ , and under the functor  $G$  it will be mapped to some other morphism  $G(f) : G(X) \rightarrow G(Y)$ , also in  $\mathbf{D}$ .

We can think of  $F(f) : F(X) \rightarrow F(Y)$  and  $G(f) : G(X) \rightarrow G(Y)$  as each being very small diagrams (directed graphs) in  $\mathbf{D}$ . If we have already chosen morphisms  $\alpha_X : F(X) \rightarrow G(X)$  and  $\alpha_Y : F(Y) \rightarrow G(Y)$  in  $\mathbf{D}$ , then these will connect the two diagrams, as depicted in the figure to the side.

In  $\mathbf{D}$ , this gives rise to the square diagram shown to the side. We’ll require, as a condition on the morphisms  $\alpha_X$  and  $\alpha_Y$ , that they make the diagram commutative:

$$F(f) \circ \alpha_Y = \alpha_X \circ G(f). \quad (1)$$

Now consider not only a single morphism  $f : X \rightarrow Y$  in  $\mathbf{C}$  being mapped by  $F$  and  $G$ , respectively, but *all* of the category  $\mathbf{C}$ . Under  $F$ , the category  $\mathbf{C}$  is mapped to a – possibly very complicated – diagram in  $\mathbf{D}$  (a directed graph of objects and morphism comprising the image of  $F$ ), and similarly, under  $G$ , the category  $\mathbf{C}$  is mapped to another diagram in  $\mathbf{D}$  (the image of  $G$ ).

To relate the image of  $F$  to the image of  $G$  we can proceed in the same way as above: for each object  $X$  in  $\mathbf{C}$ , we choose a morphism  $\alpha_X : F(X) \rightarrow G(X)$  in  $\mathbf{D}$ . In other words, we have a collection of morphisms  $(\alpha_X, X \in \text{Ob}_{\mathbf{C}})$  indexed by the objects of  $\mathbf{C}$ . These gives rise to lots of squares of the kind in Fig. 1, which we will require to be commutative. It is because of this commutativity condition, which is a condition on the collection  $(\alpha_X, X \in \text{Ob}_{\mathbf{C}})$ , that some mathematicians would say that the collection  $(\alpha_X, X \in \text{Ob}_{\mathbf{C}})$  is a “coherent” or “natural” way to relate the image of  $F$  to the image of  $G$ . (This does not mean, however, that there is at most one natural transformation between any two given functors—on the contrary, there might be many!)

In Fig. 2 we have illustrated a situation involving three objects and two morphisms in  $\mathbf{C}$ , giving rise to two squares. We have “glued” the two squares together since they share an edge (this a more compact way of drawing them). Note that because each of the two component squares in the diagram commute, so does the entire diagram.

### Definition 32.1 (Natural transformation)

Let  $\mathbf{C}$  and  $\mathbf{D}$  be categories, and let  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  be functors. A *natural transformation*  $\alpha : F \Rightarrow G$  is specified by:

#### Constituents

Watch: *Diagrams* (11 minutes).

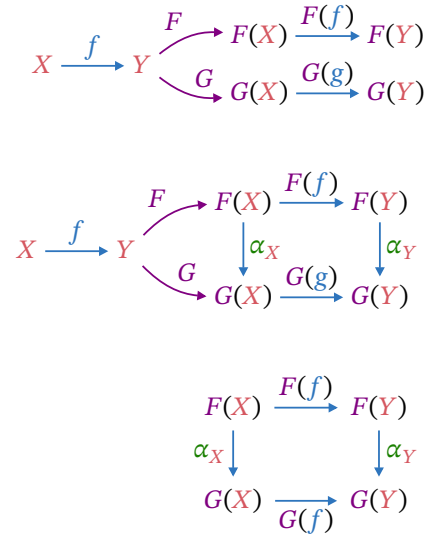
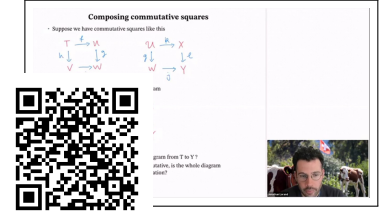


Figure 1.

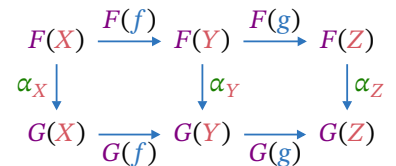
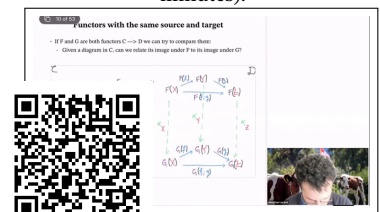


Figure 2.

Watch: *Definition of natural transformation* (8 minutes).



1. For each object  $X \in \text{Ob}_{\mathbf{C}}$ , a morphism  $\alpha_X : F(X) \rightarrow G(X)$  in  $\mathbf{D}$ , called the  $X$ -component of  $\alpha$ .

#### Conditions

1. For every morphism  $f : X \rightarrow Y$  in  $\mathbf{C}$ , the components of  $\alpha$  must satisfy the *naturality condition*

$$F(f) \circ \alpha_Y = \alpha_X \circ G(f). \quad (2)$$

In other words, the following diagram must commute:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \alpha_X \downarrow & & \downarrow \alpha_Y \\ G(X) & \xrightarrow{G(f)} & G(Y) \end{array} \quad (3)$$

To reiterate: a natural transformation  $\alpha$  is a *collection*  $(\alpha_X)_{X \in \text{Ob}_{\mathbf{C}}}$  of morphisms (called the *components* of the natural transformation) which satisfy the naturality conditions. The name “components” is analogous to how a vector  $v = (v_1, \dots, v_n)$  has *components* or a sequence  $a = (a_n)_{n \in \mathbb{N}}$  has *terms*.

The diagrams (3) are often called *naturality squares*, and a natural transformation  $\alpha : F \Rightarrow G$  is often depicted concisely in this manner:

$$\begin{array}{ccc} & F & \\ \mathbf{C} & \Downarrow \alpha & \mathbf{D} \\ & G & \end{array} \quad (4)$$

Figure 3 shows a diagram that describes the property of functors and of natural transformations. The diagram in  $\mathbf{D}$  is a “commuting prism”: all faces of the prism commute.

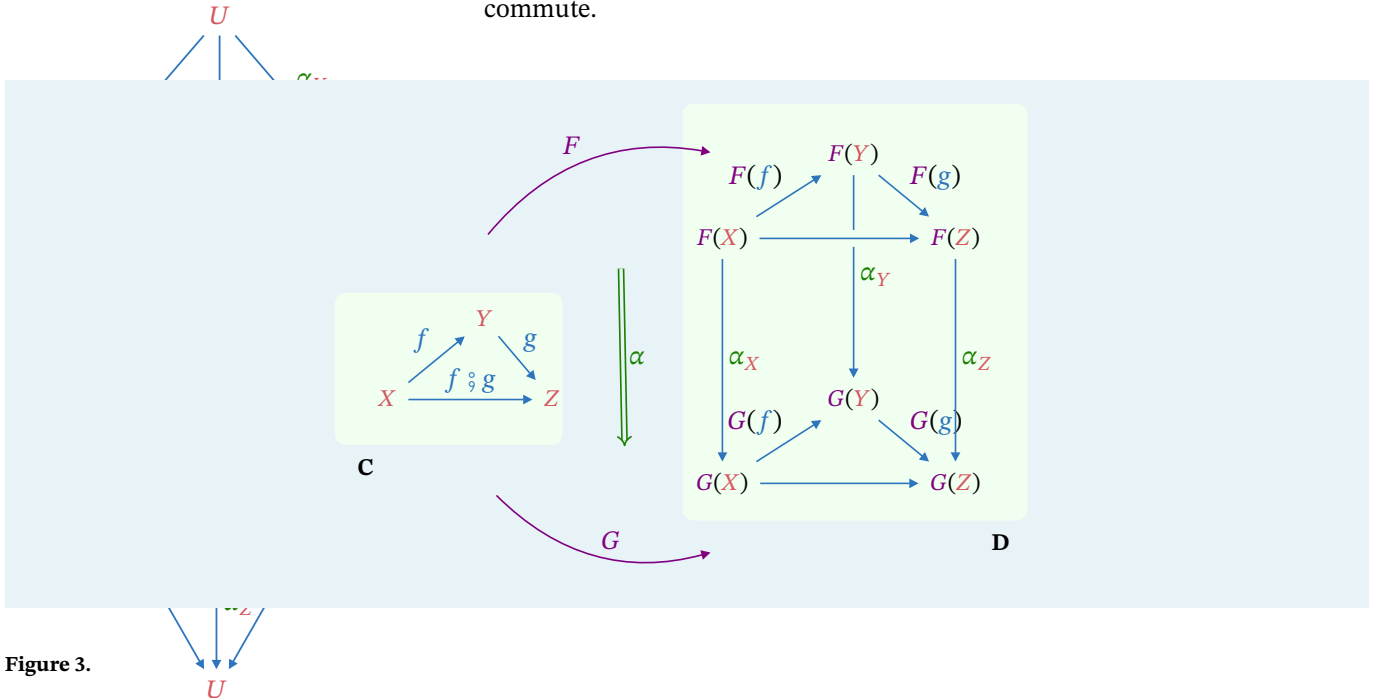


Figure 3.

Figure 5.: Cone under  $F$ .

But perhaps this prism is better explained with a few simple examples:



## Basic Examples

**Example 32.2** (Constant Functors and Cones). Let  $\mathbf{C}, \mathbf{D}$  be categories, and  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  be functors, where  $F$  is constant. That means that  $F$  maps all objects  $X_i$  of  $\mathbf{C}$  to a single object  $Y \in \mathbf{D}$ . Clearly the image of  $F$  is a single object. The natural transformation from  $F(X)$  to  $G(Y)$  thus forms what we call the cone over  $G$ , illustrated in figure Section 32.2. Analogously, if we consider  $G$  to be a constant functor, we talk about the cone under  $F$  (or cocone), as seen in figure Fig. 4.

**Example 32.3.** Consider the powerset functor from Example 19.3, and denote it  $F$ . As a reminder, the functor maps a set  $A$  to its powerset, and a function  $f : A \rightarrow B$  to the map which sends each subset of  $A$  to its image under  $f$ . We now look at a natural transformation  $\alpha : \text{id}_{\text{Set}} \Rightarrow F$  whose components are the functions

$$\begin{aligned} \alpha_A : \text{id}_{\text{Set}}(A) = A &\rightarrow F(A) = \text{Pow}(A) \\ a &\mapsto \{a\}. \end{aligned} \quad (5)$$

In other words, the natural transformation embeds each element of  $A$  into the power set  $\text{Pow}(A)$ . To check that this is a natural transformation, consider  $f : A \rightarrow B$ . We have

$$\begin{aligned} (\alpha_A \circ F(f))(a) &= \{a\} \circ F(f) \\ &= \{f(a)\} \\ &= f(a) \circ \alpha_B \\ &= (f \circ \alpha_B)(a). \end{aligned} \quad (6)$$

**Example 32.4.** Consider the following two functors  $F, G : \text{Set} \times \text{Set} \times \text{Set} \rightarrow \text{Set}$ . We define  $F$  on objects by

$$F(\langle A, B, C \rangle) = (A \times B) \times C \quad (7)$$

and define  $G$  on objects by

$$G(\langle A, B, C \rangle) = A \times (B \times C). \quad (8)$$

For their actions on morphisms, consider a morphism

$$\langle f, g, h \rangle : \langle A, B, C \rangle \rightarrow \langle A', B', C' \rangle \quad (9)$$

in  $\text{Set} \times \text{Set} \times \text{Set}$ . Its image under  $F$  is

$$\langle \langle f, g \rangle, h \rangle : (A \times B) \times C \rightarrow (A' \times B') \times C' \quad (10)$$

and its image under  $G$  is

$$\langle f, \langle g, h \rangle \rangle : A \times (B \times C) \rightarrow A' \times (B' \times C'). \quad (11)$$

One way to see that  $F$  is indeed a functor is to note that it is equal to the following composition of functors

$$\begin{aligned} \text{Set} \times \text{Set} \times \text{Set} &\rightarrow \text{Set} \times \text{Set} \rightarrow \text{Set}, \\ \langle A, B, C \rangle &\mapsto \langle A \times B, C \rangle \mapsto (A \times B) \times C \end{aligned} \quad (12)$$

and recall from Graded Exercise F.8 that “ $\times$ ” is a functor. An analogous remark applies to  $G$ .

Now we define a natural transformation  $\alpha : F \Rightarrow G$  by specifying its components

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.



to be the functions

$$\begin{aligned} \alpha_{\langle \mathbf{A}, \mathbf{B}, \mathbf{C} \rangle} : (\mathbf{A} \times \mathbf{B}) \times \mathbf{C} &\rightarrow \mathbf{A} \times (\mathbf{B} \times \mathbf{C}), \\ \langle \langle x, y \rangle, z \rangle &\mapsto \langle x, \langle y, z \rangle \rangle, \end{aligned} \quad (13)$$

indexed by triples of sets  $\langle \mathbf{A}, \mathbf{B}, \mathbf{C} \rangle$ .

For the family of morphisms  $\alpha_{\langle \mathbf{A}, \mathbf{B}, \mathbf{C} \rangle}$  to be a natural transformation, we need to check that the diagrams

$$\begin{array}{ccc} (\mathbf{A} \times \mathbf{B}) \times \mathbf{C} & \xrightarrow{\langle \langle f, g \rangle, h \rangle} & \mathbf{A}' \times (\mathbf{B}' \times \mathbf{C}') \\ \alpha_{\langle \mathbf{A}, \mathbf{B}, \mathbf{C} \rangle} \downarrow & & \downarrow \alpha_{\langle \mathbf{A}', \mathbf{B}', \mathbf{C}' \rangle} \\ \mathbf{A} \times (\mathbf{B} \times \mathbf{C}) & \xrightarrow{\langle f, \langle g, h \rangle \rangle} & \mathbf{A}' \times (\mathbf{B}' \times \mathbf{C}') \end{array} \quad (14)$$

in **Set** commute for all morphisms  $\langle f, g, h \rangle$  in  $\mathbf{Set} \times \mathbf{Set} \times \mathbf{Set}$ . It is easily checked that this is true.

This natural transformation is an example of something called an *associator*, which we will discuss later when we define monoidal categories. The idea here is that the cartesian product of sets is not quite an associative operation, but almost: instead of an “equality” symbol in the usual equation for the associative law, we have the components of this associator natural transformation.

**Example 32.5.** For any two sets  $\mathbf{A}, \mathbf{B}$ , consider the function

$$\begin{aligned} \alpha_{\langle \mathbf{A}, \mathbf{B} \rangle} : \mathbf{A} \times \mathbf{B} &\rightarrow \mathbf{B} \times \mathbf{A}, \\ \langle x, y \rangle &\mapsto \langle y, x \rangle. \end{aligned} \quad (15)$$

When we think of (15) as a family of functions parametrized by  $\langle \mathbf{A}, \mathbf{B} \rangle$ , they are in fact the components of a natural transformation  $\alpha$ . Can you write down what the functors are that are the source and target of this natural transformation?

### 32.3. Morphisms in a Category of Functors

We have seen that natural transformations between two functors  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  map objects in  $\mathbf{C}$  to morphisms in  $\mathbf{D}$ , and map morphisms in  $\mathbf{C}$  to commutative diagrams. This is quite similar to the effect of functors on category objects and morphisms. What if there were a category where objects are functors, and morphisms are natural transformations?

#### Vertical Composition

As for any category, we would need to define the morphism composition law and the identity morphisms. The former would look like this:

$$\begin{array}{ccc}
 \begin{array}{ccc}
 & F & \\
 & \searrow & \\
 \mathbf{C} & \xrightarrow{G} & \mathbf{D} \\
 & \nearrow & \\
 & H & 
 \end{array}
 & \xrightarrow{\text{morphism composition}} &
 \begin{array}{ccc}
 & F & \\
 & \searrow & \\
 \mathbf{C} & \xrightarrow{\alpha \circ \beta} & \mathbf{D} \\
 & \nearrow & \\
 & H & 
 \end{array}
 \end{array}
 \quad (16)$$

Due to its diagrammatic form, we call this type of composition vertical composition.

**Definition 32.6** (Vertical Composition of natural transformations)

Let  $\mathbf{C}, \mathbf{D}$  be categories and let  $F, G, H : \mathbf{C} \rightarrow \mathbf{D}$  be functors from  $\mathbf{C}$  to  $\mathbf{D}$ . Suppose we are given natural transformations

$$\alpha : F \Rightarrow G, \quad (17)$$

$$\beta : G \Rightarrow H. \quad (18)$$

Their composition  $\alpha \circ \beta$  is a natural transformation

$$\alpha \circ \beta : F \Rightarrow H \quad (19)$$

defined in components by

$$(\alpha \circ \beta)_X := \alpha_X \circ \beta_X \quad \forall X \in \text{Ob}_{\mathbf{C}}. \quad (20)$$

Next up, we can trivially define the identity natural transformations.

**Definition 32.7** (Identity natural transformation)

Let  $\mathbf{C}, \mathbf{D}$  be categories and let  $F : \mathbf{C} \rightarrow \mathbf{D}$  be a functor. The identity natural transformation at  $F$  is the natural transformation  $\text{id}_F : F \Rightarrow F$  defined in components by

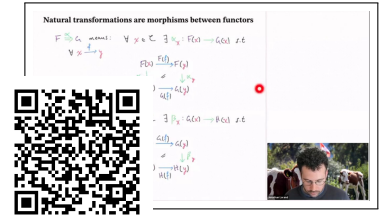
$$(\text{id}_F)_X := \text{id}_{F(X)} \quad \forall X \in \text{Ob}_{\mathbf{C}}. \quad (21)$$

This is exactly what we were looking for! We can now define a category of functors.

**Definition 32.8**

Let  $\mathbf{C}, \mathbf{D}$  be categories. The category  $[\mathbf{C}, \mathbf{D}]$  of functors from  $\mathbf{C}$  to  $\mathbf{D}$  is given by

Watch: Natural transformations are morphisms between functors (2 minutes).



1. *Objects*: functors  $\mathbf{C} \rightarrow \mathbf{D}$ .
2. *Morphisms*: natural transformations between functors  $\mathbf{C} \rightarrow \mathbf{D}$ .
3. *Composition*: composition of natural transformations.
4. *Identities*: identity natural transformations.

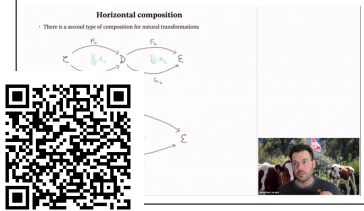
## Natural isomorphisms

### Definition 32.9 (Natural isomorphism)

A natural transformation  $\alpha : F \Rightarrow G$  is called a *natural isomorphism* if each component morphism  $\alpha_X$  in  $\mathbf{D}$  is an isomorphism.

**Lemma 32.10.** Let  $\mathbf{C}, \mathbf{D}$  be categories and let  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  be functors. A natural isomorphism  $\alpha : F \Rightarrow G$  is an isomorphism in the category of functors  $[\mathbf{C}, \mathbf{D}]$  if and only if  $\alpha$  is a natural isomorphism.

Watch: *Horizontal composition* (5 minutes).



## Horizontal Composition

But what about the following case? Suppose we have three categories:  $\mathbf{C}, \mathbf{D}$  and  $\mathbf{E}$  with functors  $F_1, F_2 : \mathbf{C} \rightarrow \mathbf{D}$  and  $G_1, G_2 : \mathbf{D} \rightarrow \mathbf{E}$  and finally two natural transformations  $\alpha : F_1 \Rightarrow F_2$  and  $\beta : G_1 \Rightarrow G_2$ . We would then have the following situation.

$$\begin{array}{ccc} \mathbf{A} & \xrightarrow{F_1} & \mathbf{B} \\ \Downarrow \alpha & & \Downarrow \beta \\ \mathbf{A} & \xrightarrow{G_1} & \mathbf{C} \end{array} \quad (22)$$

This looks suspiciously composable. Since we could compare have a natural transformation between  $F_1 \circ G_1$  and  $F_2 \circ G_2$ , why not observe the intermediate steps?

### Definition 32.11 (Horizontal Composition of natural transformations)

Let  $\mathbf{C}, \mathbf{D}, \mathbf{E}$  be categories and let  $F_1, F_2 : \mathbf{C} \rightarrow \mathbf{D}$  and  $G_1, G_2 : \mathbf{D} \rightarrow \mathbf{E}$  be functors. Suppose we are given the natural transformations

$$\alpha : F_1 \Rightarrow F_2, \quad (23)$$

$$\beta : G_1 \Rightarrow G_2. \quad (24)$$

Then their horizontal composition is given by the natural transformation

$$\alpha * \beta : (F_1 \circ G_1) \Rightarrow (F_2 \circ G_2) \quad (25)$$

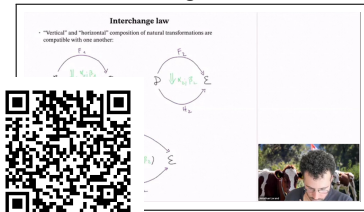
defined in components by

$$(\alpha * \beta)_X := \alpha_X * \beta_X \quad \forall X \in \text{Ob}_{\mathbf{C}} \quad (26)$$

as the composition

$$(\alpha * \beta)_X : (F_2(F_1(X))) \Rightarrow (G_2(F_1(X))) \Rightarrow (G_2(G_1(X))) \quad (27)$$

Watch: *Interchange law* (2 minutes).



## Interchange Law

The following statement is quite powerful. Although perhaps obvious-looking, it allows us to presume associativity of natural transformations.

**Proposition 32.12** (Interchange). Let  $\mathbf{C}, \mathbf{D}, \mathbf{E}$  be categories,  $F_1, G_1, H_1 : \mathbf{C} \rightarrow \mathbf{D}$ ,  $F_2, G_2, H_2 : \mathbf{D} \rightarrow \mathbf{E}$  be functors and  $\alpha_1 : F_1 \Rightarrow G_1$ ,  $\alpha_2 : F_2 \Rightarrow G_2$ ,  $\beta_1 : G_1 \Rightarrow H_1$ ,  $\beta_2 : G_2 \Rightarrow H_2$  be natural transformations. Then,

$$(\alpha_1 \circ \beta_1) * (\alpha_2 \circ \beta_2) = (\alpha_1 * \alpha_2) \circ (\beta_1 * \beta_2) \quad (28)$$

*Proof.* Consider the case where we have categories  $\mathbf{C}, \mathbf{D}, \mathbf{E}$ , with functors  $F_1, G_1 : \mathbf{C} \rightarrow \mathbf{D}$  and  $F_2, G_2 : \mathbf{D} \rightarrow \mathbf{E}$  relating them. We can thus have the following four diagrams:

$$\begin{array}{ccc} \mathbf{A} & \xrightarrow{F_1} & \mathbf{B} \\ \Downarrow \alpha_1 & & \Downarrow \alpha_2 \\ \mathbf{B} & \xrightarrow{G_1} & \mathbf{C} \end{array} \quad \begin{array}{ccc} \mathbf{B} & \xrightarrow{F_2} & \mathbf{C} \\ \Downarrow \alpha_2 & & \Downarrow \alpha_2 \\ \mathbf{C} & \xrightarrow{G_2} & \mathbf{C} \end{array} \quad (29)$$

$$\begin{array}{ccc} \mathbf{A} & \xrightarrow{G_1} & \mathbf{B} \\ \Downarrow \beta_1 & & \Downarrow \beta_2 \\ \mathbf{B} & \xrightarrow{H_1} & \mathbf{C} \end{array} \quad \begin{array}{ccc} \mathbf{B} & \xrightarrow{G_2} & \mathbf{C} \\ \Downarrow \beta_2 & & \Downarrow \beta_2 \\ \mathbf{C} & \xrightarrow{H_2} & \mathbf{C} \end{array} \quad (30)$$

We are now faced with the choice of either vertically composing first, or horizontally composing. By vertically composing, we get

$$\begin{array}{ccc} \mathbf{A} & \xrightarrow{F_1} & \mathbf{B} \\ \Downarrow \alpha_1 \circ \beta_1 & & \Downarrow \alpha_2 \circ \beta_2 \\ \mathbf{B} & \xrightarrow{H_1} & \mathbf{C} \end{array} \quad \begin{array}{ccc} \mathbf{B} & \xrightarrow{F_2} & \mathbf{C} \\ \Downarrow \alpha_2 \circ \beta_2 & & \Downarrow \alpha_2 \circ \beta_2 \\ \mathbf{C} & \xrightarrow{H_2} & \mathbf{C} \end{array} \quad (31)$$

Subsequently applying horizontal composition yields

$$\begin{array}{ccc} \mathbf{A} & \xrightarrow{F_1 \circ F_2} & \mathbf{C} \\ \Downarrow (\alpha_1 \circ \beta_1) * (\alpha_2 \circ \beta_2) & & \Downarrow (\alpha_1 \circ \beta_1) * (\alpha_2 \circ \beta_2) \\ \mathbf{B} & \xrightarrow{H_1 \circ H_2} & \mathbf{C} \end{array} \quad (32)$$

Otherwise, we apply horizontal composition first:

$$\begin{array}{ccc} \mathbf{C} & \xrightarrow{F_1 \circ F_2} & \mathbf{D} \\ \Downarrow \alpha_1 \circ \alpha_2 & & \Downarrow \alpha_1 \circ \alpha_2 \\ \mathbf{C} & \xrightarrow{G_1 \circ G_2} & \mathbf{D} \\ \Downarrow \beta_1 \circ \beta_2 & & \Downarrow \beta_1 \circ \beta_2 \\ \mathbf{C} & \xrightarrow{H_1 \circ H_2} & \mathbf{D} \end{array} \quad (33)$$

th's proving that vertical and horizontal composition are interchangeable.  $\square$

**Remark 32.13.** The proof is also interesting to do by observing the commuting squares, and putting them together.

## Whiskering

A variant of horizontal composition is the case where we compose a natural transformation with a functor. We can have two cases: right or left whiskering.

$$\begin{array}{ccc} & F & \\ A & \xrightarrow{\quad} & B \\ & G & \\ & \Downarrow \alpha & \end{array} \quad (34)$$

### Definition 32.14 (Right Whiskering)

Let  $\mathbf{C}, \mathbf{D}$  and  $\mathbf{E}$  be categories,  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  be functors from  $\mathbf{C}$  to  $\mathbf{D}$ ,  $H : \mathbf{D} \rightarrow \mathbf{E}$  be a functor from  $\mathbf{D}$  to  $\mathbf{E}$  and  $\alpha : F \Rightarrow G$  be a natural transformation from  $F$  to  $G$ .

The right whiskering of  $H$  and  $\alpha$  is given by the natural transformation

$$H\alpha : (F \circ H) \Rightarrow (G \circ H) \quad (35)$$

### Definition 32.15 (Left Whiskering)

Let  $\mathbf{C}, \mathbf{D}$  and  $\mathbf{E}$  be categories,  $F, G : \mathbf{C} \rightarrow \mathbf{D}$  be functors from  $\mathbf{C}$  to  $\mathbf{D}$ ,  $H : \mathbf{B} \rightarrow \mathbf{C}$  be a functor from  $\mathbf{B}$  to  $\mathbf{C}$  and  $\beta : F \Rightarrow G$  be a natural transformation from  $F$  to  $G$ .

The left whiskering of  $\beta$  and  $H$  is given by the natural transformation

$$\beta H : (H \circ F) \Rightarrow (H \circ G) \quad (36)$$

## 32.4. Data migration

Now that we have seen some definitions and toy examples, let's observe a real-world example. As mentioned in *Categorical Databases* (see 19.5 for the full context), we can model instances of databases due to Spivak [ref / cite] as functors from the category of the architecture of the database to the category of sets.

**An alumni database** The architecture of the database in our example might be encoded by a category with four objects and three non-identity morphisms, as depicted in Fig. 6. The object  $S$  stands for *Student*,  $D$  stands for *Discipline*,  $N$  stands for *Name*, and  $Y$  stands for *Year*. Call this category  $\mathbf{C}$ . A database instance  $F : \mathbf{C} \rightarrow \mathbf{Set}$  entails specifying a set  $F(S)$  of all student IDs, a set  $F(D)$  of university disciplines such as mechanical engineering, civil engineering, applied mathematics, pure mathematics, *etc.* It also entails defining functions for each of the arrows in  $\mathbf{C}$ . For example,  $F(\text{studied}) : F(S) \rightarrow F(D)$  is the function that assigns to each student ID the name of the discipline that that student studied.

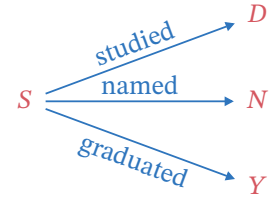


Figure 6.: The schema of an alumni database.

**Updating the database** For simplicity, we focus on two aspects of the data: student ID numbers and the disciplines of study. We assume that the university updates its alumni database once a year. This means, for example, adding the graduates of that year to the total list of graduates.

To model the situation, let  $F : \mathbf{C} \rightarrow \mathbf{Set}$  be the database instance for the year 2021, and let  $G : \mathbf{C} \rightarrow \mathbf{Set}$  be the database instance for the year 2022. For concreteness, suppose the student IDs are in some standardized format, for example a code of the kind 17-371-802, where each of the three parts of the code are calculated/assigned by some rule (e.g., the “17” here stands for 2017, the year the student registered with the university, *etc.*).

Since new students register to the university each year, the set  $F(S)$  of all student IDs registered up to the end of 2020 is a subset of the set  $G(S)$  of student IDs up to the end of 2022. This means there is an inclusion function  $\alpha_S : F(S) \rightarrow G(S)$ .

Now suppose that in 2022 the university decides to simplify the way it attributes disciplines to students in the database.

For instance, instead of the discipline names  $F(D) = \{\text{mechanical engineering, civil engineering, applied physics, theoretical physics, pure math, applied math}\}$ , the new discipline names are just

$$G(D) = \{\text{engineering, physics, math}\}. \quad (37)$$

In order to implement these changes, we use a function  $\alpha_D : F(D) \rightarrow G(D)$  which maps the old discipline names to the corresponding new ones in an obvious way:

$$\alpha_D(\text{civil engineering}) = \text{engineering}, \alpha_D(\text{applied physics}) = \text{physics}, \text{etc.} \quad (38)$$

The functions  $\alpha_S$  and  $\alpha_D$  allow us to check whether the new database instance  $G$  relates coherently with the older database instance  $F$ . Concretely, we want that if a student ID in  $G(S)$  is inherited from  $F(S)$  – in other words, if it is in the image of  $\alpha_S$  – then we want that its associated discipline in the database instance  $G$  is the same as if we first computed the student's discipline in the older database instance  $F$ , and then mapped it to  $G$  using the function  $\alpha_D$ .

This can be formulated succinctly by saying that we want the diagram in figure Fig. 7 to commute.

What we have defined is a collection of morphisms: for each object  $X$  in  $\mathbf{C}$ , we have a morphism  $\alpha_X : F(X) \rightarrow G(X)$  that obeys the commutativity property

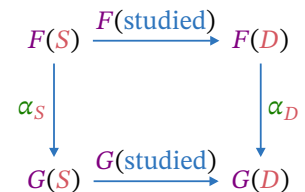


Figure 7.

corresponding to the diagram. Such transformations appear in many places and are formalized by the notion of natural transformations.

## 32.5. More examples

**Example 32.16.** Consider the category  $\mathbf{Vect}_{\mathbb{R}}$  whose objects are real vector spaces and whose morphisms are linear maps. (For convenience, in the following we sometimes omit reference to the ground field.) Recall that the *dual* of a vector space  $V$  is the vector space describing all linear maps from  $V$  to  $\mathbb{R}$ :

$$V^* := \mathbf{Hom}_{\mathbf{Vect}_{\mathbb{R}}}(V; \mathbb{R}), \quad (39)$$

Also, recall that if  $f : V \rightarrow W$  is a linear map, then its dual is the linear map  $f^* : W^* \rightarrow V^*$  which maps any  $\xi \in W^*$  to the element of  $V^*$  given by

$$f^*(\xi) : V \rightarrow \mathbb{R}, v \mapsto \xi(f(v)). \quad (40)$$

Applying the above duality construction twice to a vector space or a linear map gives their double dual. It turns out that this is a functorial operation. That is, there is a functor

$$\text{Double dual} : \mathbf{Vect}_{\mathbb{R}} \rightarrow \mathbf{Vect}_{\mathbb{R}} \quad (41)$$

that maps every vector space and every linear map to its double dual.

Furthermore, for any vector space  $V$ , there is a “canonical” or “natural” map

$$\alpha_V : V \Rightarrow V^{**} \quad (42)$$

defined by

$$\alpha_V(v)(l) = l(v), \quad v \in V, l \in V^*. \quad (43)$$

These form the components of a natural transformation from the identity functor on  $\mathbf{Vect}_{\mathbb{R}}$  to the double dual functor.

$$\begin{array}{ccc} & \text{id} & \\ & \curvearrowright & \\ \mathbf{Vect}_{\mathbb{R}} & \Downarrow \alpha & \mathbf{Vect}_{\mathbb{R}} \\ & \curvearrowleft & \\ & \text{Double dual} & \end{array} \quad (44)$$

**Example 32.17.** Fix a set  $S$ . There are functors  $F, G : \mathbf{Set}^{\text{op}} \times \mathbf{Set} \rightarrow \mathbf{Set}$  whose respective actions on objects are

$$F_{\bullet} : \langle \mathbf{A}, \mathbf{B} \rangle \mapsto \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A} \times \mathbf{S}, \mathbf{B}) \quad (45)$$

and

$$G_{\bullet} : \langle \mathbf{A}, \mathbf{B} \rangle \mapsto \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}, \mathbf{B}^S). \quad (46)$$

These functors may be understood as built up using compositions of functors of the kind discussed in Graded Exercise F.4, Graded Exercise F.5 and Example 19.11.

Recall that we can “curry” any function  $f : \mathbf{A} \times \mathbf{S} \rightarrow \mathbf{B}$  to get a function  $\hat{f} : \mathbf{A} \rightarrow \mathbf{B}^S$ , where  $\hat{f}(x)$  may be thought of as a partial evaluation of  $f$ .

There is a natural transformation  $\alpha : F \Rightarrow G$  whose components are the functions

$$\alpha_{\langle \mathbf{A}, \mathbf{B} \rangle} : \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A} \times \mathbf{S}, \mathbf{B}) \rightarrow \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}, \mathbf{B}^S), f \mapsto \hat{f}, \quad (47)$$

where  $\hat{f}$  is the “curried” version of  $f$ .

Watch: *Double dual* (10 minutes).

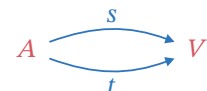
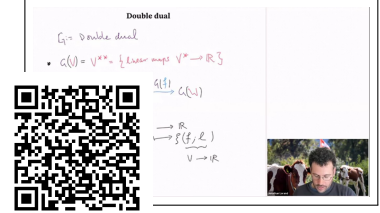


Figure 8.

### Graded exercise I.1 (NatTrafosGraphs)



This exercise builds on Graded Exercise F.6. There, we defined a category  $\mathbf{G}$  which has precisely two objects and four morphisms, see Fig. 8 (the two identity morphisms are not drawn). The task there was to understand how specifying a functor from this category  $\mathbf{CatSymbolG}$  into the category of sets is “the same thing” as specifying a directed graph.

Now consider two functors  $F_1, F_2 : \mathbf{G} \rightarrow \mathbf{Set}$ . Spell out what it means to have a natural transformation  $\alpha : F_1 \Rightarrow F_2$ . What does this correspond to in the language of directed graphs?

### Graded exercise I.2 (UpperSetsNatTrafos)

This exercise builds on Graded Exercise F.7. There we fixed a poset  $\mathbf{P}$ , viewed it as a category  $\mathbf{P}$ , and saw that functors  $\mathbf{P} \rightarrow \mathbf{Bool}$  encode upper sets in  $\mathbf{P}$ . Suppose we have two functors  $F_1, F_2 : \mathbf{P} \rightarrow \mathbf{Bool}$ . What does a natural transformation  $\alpha : F_1 \Rightarrow F_2$  correspond to in terms of the upper sets encoded by  $F_1$  and  $F_2$ , respectively?

### Graded exercise I.3 (DoubleDualNatTrafo)

This exercise builds on Graded Exercise F.3. Consider the category  $\mathbf{Vect}_{\mathbb{R}}$  whose objects are real vector spaces and whose morphisms are linear maps. For any vector space  $V$ , there is a “canonical” or “natural” map

$$\alpha_V : V \Rightarrow V^{**} \quad (48)$$

defined by

$$\alpha_V(v)(l) = l(v), \quad v \in V, l \in V^*. \quad (49)$$

Your tasks in this exercise:

1. Check that the operation of “taking the double dual” (see (41) above) defines a functor.
2. Verify that these components define a natural transformation from the identity functor on  $\mathbf{Vect}_{\mathbb{R}}$  to the double dual functor.

$$\begin{array}{ccc} & \text{id} & \\ & \curvearrowright & \\ \mathbf{Vect}_{\mathbb{R}} & & \mathbf{Vect}_{\mathbb{R}} \\ & \Downarrow \alpha & \\ & \curvearrowleft & \\ & \text{Double dual} & \end{array} \quad (50)$$

### Graded exercise I.4 (NaturalCurry)

Fix a set  $\mathbf{S}$ . There are functors  $F, G : \mathbf{Set}^{\text{op}} \times \mathbf{Set} \rightarrow \mathbf{Set}$  whose respective actions on objects are

$$F : \langle \mathbf{A}, \mathbf{B} \rangle \mapsto \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A} \times \mathbf{S}, \mathbf{B}) \quad (51)$$

and

$$G : \langle \mathbf{A}, \mathbf{B} \rangle \mapsto \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}, \mathbf{B}^{\mathbf{S}}). \quad (52)$$

On morphisms,  $F$  acts as follows. Given a morphism  $\langle f^{\text{op}}, g \rangle : \langle \mathbf{A}, \mathbf{B} \rangle \rightarrow \langle \mathbf{A}', \mathbf{B}' \rangle$  in  $\mathbf{Set}^{\text{op}} \times \mathbf{Set}$ , the function

$$F(\langle f^{\text{op}}, g \rangle) : \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A} \times \mathbf{S}, \mathbf{B}) \rightarrow \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}' \times \mathbf{S}, \mathbf{B}')$$

takes any function  $\varphi : \mathbf{A} \times \mathbf{S} \rightarrow \mathbf{B}$  and maps it to the function

$$\langle f, \text{id}_{\mathbf{S}} \rangle \circ \varphi \circ g : \mathbf{A}' \times \mathbf{S} \rightarrow \mathbf{B}'. \quad (53)$$

Let us also define the action of  $G$  on morphisms. Given again a morphism  $\langle f^{\text{op}}, g \rangle : \langle \mathbf{A}, \mathbf{B} \rangle \rightarrow \langle \mathbf{A}', \mathbf{B}' \rangle$  in  $\mathbf{Set}^{\text{op}} \times \mathbf{Set}$ , the function

$$G(\langle f^{\text{op}}, g \rangle) : \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}, \mathbf{B}^{\mathbf{S}}) \rightarrow \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}', \mathbf{B}'^{\mathbf{S}})$$

takes any function  $\varphi : \mathbf{A} \rightarrow \mathbf{B}^{\mathbf{S}}$  and maps it to the function

$$f \circ \varphi \circ g_* : \mathbf{A}' \rightarrow \mathbf{B}'^{\mathbf{S}} \quad (54)$$

where  $g_*$  is the function

$$g_* : \mathbf{B}^{\mathbf{S}} \rightarrow \mathbf{B}'^{\mathbf{S}}, \quad \psi \mapsto \psi \circ g. \quad (55)$$

Now, recall that we can “curry” any function  $f : \mathbf{A} \times \mathbf{S} \rightarrow \mathbf{B}$  to get a function  $\bar{f} : \mathbf{A} \rightarrow \mathbf{B}^{\mathbf{S}}$ , where  $\bar{f}(x)$  may be thought of as a partial evaluation of  $f$ .

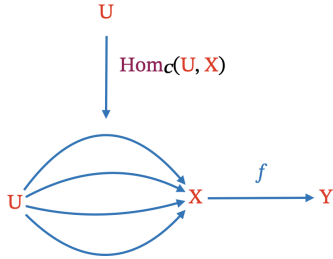
Your task in this exercise: show that the functions

$$\alpha_{\langle \mathbf{A}, \mathbf{B} \rangle} : \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A} \times \mathbf{S}, \mathbf{B}) \rightarrow \mathbf{Hom}_{\mathbf{Set}}(\mathbf{A}, \mathbf{B}^{\mathbf{S}}), \quad f \mapsto \bar{f}, \quad (56)$$

where  $\bar{f}$  is the “curried” version of  $f$ , are the components of a natural transformation  $\alpha : F \Rightarrow G$ .

## 32.6. Yoneda Embedding Theorem & Lemma

Here we will cover one of the most important results in Category Theory. It gives us insight on the role that objects play in a category, and how we can actually 'recognize' what their role is. We will first define certain types of functors for any (locally small) category  $\mathbf{C}$ , and see how they actually reveal more hidden structure on  $\mathbf{C}$ .



### Representable Functors

#### Definition 32.18 (Hom-Functor)

Let  $\mathbf{C}$  be a category, and  $X \in \text{Ob}_{\mathbf{C}}$ . We define the hom-functor of  $X$  as  $\text{Hom}_{\mathbf{C}}(X, -) : \mathbf{C} \rightarrow \mathbf{Set}$ . It maps  $U \in \text{Ob}_{\mathbf{C}}$  to the hom-set  $\text{Hom}_{\mathbf{C}}(X, U)$ .

#### Definition 32.19 (Representable Functor)

Let  $\mathbf{C}$  be a category, and  $X, Y \in \text{Ob}_{\mathbf{C}}$  be objects. A functor  $F : \mathbf{C} \rightarrow \mathbf{Set}$  is said representable, of representing object  $Z \in \text{Ob}_{\mathbf{C}}$ , if there exists a natural isomorphism between  $F$  and the functor  $\text{Hom}_{\mathbf{C}}[Z, -]$ . Moreover, any morphism  $f : X \rightarrow Y$  gets mapped to  $F \circ f$ .

That is, for each object  $X \in \text{Ob}_{\mathbf{C}}$ ,  $F(X)$  maps  $X$  to a set, which is naturally isomorphic to all the ways to map  $Z$  to  $X$  in  $\mathbf{C}$ .

Recall that we have defined a category of functors in the previous section. We can now consider the category of representable functors in  $\mathbf{C}$ , which we will denote  $[\mathbf{C}, \mathbf{Set}]$ . This new category will provide some more insight on 'unicity' of objects within  $\mathbf{C}$ . More specifically, we will see that objects are characterized by their "incoming" and "outgoing" morphisms.

**Theorem 32.20 (Yoneda Embedding).** Let  $\mathbf{C}$  be a category, and  $X, Y \in \text{Ob}_{\mathbf{C}}$ . Then there is a natural bijection of sets between the hom-set  $\text{Hom}_{\mathbf{C}}(X, Y)$  and the hom-set  $\text{Hom}_{[\mathbf{C}, \mathbf{Set}]}(\text{Hom}_{\mathbf{C}}(Y, -), \text{Hom}_{\mathbf{C}}(X, -))$  in the category of representable functors.

Let's break this down a bit. We have two objects  $X, Y$  representing two functors  $\text{Hom}_{\mathbf{C}}(X, -), \text{Hom}_{\mathbf{C}}(Y, -)$ . These functors can be also considered as objects in  $[\mathbf{C}, \mathbf{Set}]$ . We are essentially saying that the ways to map  $X$  to  $Y$  in  $\mathbf{C}$  is isomorphic to the ways to map  $\text{Hom}_{\mathbf{C}}(X, -)$  to  $\text{Hom}_{\mathbf{C}}(Y, -)$  in  $[\mathbf{C}, \mathbf{Set}]$ .

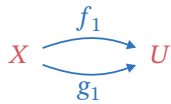


Figure 9.

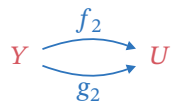


Figure 10.

**Example 32.21.** Let us consider  $X, Y, U \in \text{Ob}_{\mathbf{C}}$ , with the following hom-sets:  $\text{Hom}_{\mathbf{C}}(X, U) = \{f_1, f_2\}, \text{Hom}_{\mathbf{C}}(Y, U) = \{g_1, g_2\}$ . We can thus infer that  $\text{Hom}_{\mathbf{C}}(X, Y) = \{f_1 \circ g_1, f_1 \circ g_2, f_2 \circ g_1, f_2 \circ g_2\}$ .

Analogously, the ways to map  $\text{Hom}_{\mathbf{C}}(Y, U)$  to  $\text{Hom}_{\mathbf{C}}(X, U)$  is given by the set  $\{g_1 \mapsto f_1, g_1 \mapsto f_2, g_2 \mapsto f_1, g_2 \mapsto f_2\}$ . Both sets are bijective. Moreover, we see that in this case, both objects  $X, Y$  have the same number of "ins" and "outs" with respect to  $U$ . A general result in Category Theory is that if the hom-sets of  $X$  and  $Y$  are isomorphic for all  $U \in \text{Ob}_{\mathbf{C}}$ , then  $X$  and  $Y$  are isomorphic, and essentially play the same role within  $\mathbf{C}$ .



## 33. Adjunctions

33.1 Formal concept analysis . . . . .	480
33.2 Galois connections . . . . .	486
33.3 Adjunctions: hom-set definition	488
33.4 Adjunctions: (co)unit definition	489
33.5 Product-Hom adjunction . . . . .	491
33.6 Free-forgetful adjunction . . . . .	492
33.7 Relating the two definitions . . .	494

The *Battle of Surfaces* was a men's tennis exhibition match that was held on May 2, 2007, between the Swiss Roger Federer and Rafael Nadal, respectively number 1 and 2 in the world in men's singles. Federer preferred grass—he was 5 years unbeaten on that terrain. Nadal preferred clay—he was 3 years unbeaten. To check who would win when averaging out the terrain, the match was played on a unique court with a clay surface on one side of the net and grass on the other.

### 33.1. Formal concept analysis

Formal concept analysis (FCA) is a mathematical theory which formalizes the relationships, and in particular hierarchies, that appear when we consider a set of *things* – these are called *objects* in FCA – together with a set of *attributes* that these things may or may not have. (We will use the word “things” instead of “objects”, because we are already using the word “objects” in the category-theory sense.)

The most basic set-up for formal concept analysis is to start with a triple  $\langle \mathbf{G}, \mathbf{M}, I \rangle$ , where  $\mathbf{G}$  is a set of things (“G” stands for the German word “Gegenstände”),  $\mathbf{M}$  is a set of attributes (“M” stands for the German word “Merkmale”), and  $I \subseteq \mathbf{G} \times \mathbf{M}$  is a relation that encodes which objects are associated with which attributes (“I” stands for the German word “Inzidenz”). The triple  $\langle \mathbf{G}, \mathbf{M}, I \rangle$  is called a *formal context*.

Here is a (very simplified) example in the context of “private means of transportation”. We consider the set of things to be the following means of transportation

$$\mathbf{G} = \{\text{classic car, hybrid car, electric car, classic bike, hybrid bike, electric scooter, skateboard}\} \quad (1)$$

and we consider the set of attributes

$$\mathbf{M} = \{\text{fast, electric, gas, muscle, cheap}\}, \quad (2)$$

which describe aspects such how the means of transportation are powered, their relative cost, or if they can go fast enough to move on a highway, for example. We define the relation  $I \subseteq \mathbf{G} \times \mathbf{M}$  via the following table

	fast	electric	gas	muscle	cheap
classic car	×		×		
hybrid car	×	×	×		
electric car	×	×			
classic bike				×	×
hybrid bike		×		×	
electric scooter		×			×
skateboard				×	×

where a cross “×” indicates when a thing and an attribute are associated with each other.

#### Induced monotone maps

For each element  $x$  of  $\mathbf{G}$ , we can consider the set  $I_{\#}(\{x\})$  of attributes that are associated with  $x$ . This corresponds to reading off in the the above table where there are crosses “×” in the row labeled by  $x$ . For example

$$I_{\#}(\{\text{hybrid car}\}) = \{\text{fast, electric, gas}\}. \quad (3)$$

More generally, given a subset  $\mathbf{A} \subseteq \mathbf{G}$ , can can consider the largest set  $I_{\#}(\mathbf{A})$  of attributes which all elements of  $\mathbf{A}$  have in common. For example

$$I_{\#}(\{\text{hybrid car, electric car}\}) = \{\text{fast, electric}\}. \quad (4)$$

Here, “gas” is not an element of  $I_{\#}(\{\text{hybrid car, electric car}\})$  because “gas” is associated with hybrid car but not with electric car.

Similarly,

$$I_{\#}(\{\text{classic car, hybrid car, electric car}\}) = \{\text{fast}\}. \quad (5)$$



Also, we have, for example,

$$I_{\#}(\{\text{classic car, classic bike}\}) = \emptyset \quad (6)$$

because “classic car” and “classic bike” have *no* attributes from the set **M** in common. In general we have

$$I_{\#}(\mathbf{A}) = \bigcap_{x \in \mathbf{A}} I_{\#}(\{x\}). \quad (7)$$

and the operation “ $I_{\#}$ ” defines a function

$$I_{\#} : \text{Pow } \mathbf{G} \rightarrow \text{Pow } \mathbf{M}. \quad (8)$$

Observe that the larger **A** is, the smaller  $I_{\#}(\mathbf{A})$  will be. Formulated more mathematically, we have

$$\mathbf{A} \subseteq \mathbf{A}' \implies I_{\#}(\mathbf{A}) \supseteq I_{\#}(\mathbf{A}'). \quad (9)$$

Another way of saying this is to say that  $I_{\#}$  is a monotone map of posets

$$I_{\#} : \langle \text{Pow } \mathbf{G}, \subseteq \rangle \rightarrow \langle \text{Pow } \mathbf{M}, \supseteq \rangle. \quad (10)$$

Or, equivalently, we can say that  $I_{\#}$  is a monotone map

$$I_{\#} : \langle \text{Pow } \mathbf{G}, \subseteq \rangle \rightarrow \langle \text{Pow } \mathbf{M}, \subseteq \rangle^{\text{op}}. \quad (11)$$

Note that we can also define a similar map in the other direction: there is a function

$$I_b : \text{Pow } \mathbf{M} \rightarrow \text{Pow } \mathbf{G} \quad (12)$$

defined such that for any subset  $\mathbf{B} \subseteq \mathbf{M}$ , the set  $I_b(\mathbf{B})$  is the largest set of elements of **G** such that the attributes in **B** apply to all of the elements of  $I_b(\mathbf{B})$ . For example,

$$I_b(\{\text{muscle, cheap}\}) = \{\text{classic bike, skateboard}\}. \quad (13)$$

The map  $I_b$  is also order-reversing with respect to inclusion of sets: if we start with a larger set of attributes, then set of things that these all apply to will be smaller. Thus we have a monotone map

$$I_b : \langle \text{Pow } \mathbf{M}, \subseteq \rangle^{\text{op}} \rightarrow \langle \text{Pow } \mathbf{G}, \subseteq \rangle. \quad (14)$$

We will also want to use its opposite, the monotone map

$$I_b^{\text{op}} : \langle \text{Pow } \mathbf{M}, \subseteq \rangle \rightarrow \langle \text{Pow } \mathbf{G}, \subseteq \rangle^{\text{op}}. \quad (15)$$

In the following, we will try to keep track of when there is an superscript “ $(-)^{\text{op}}$ ”; however sometimes it will be convenient to use the notations  $I_{\#}$  and  $I_b$  both for these maps *and* their opposites (in particular, on the level of objects they are the same function).

A key observation is that  $I_{\#}$  and  $I_b$  are “complementary” in the following sense. For any  $\mathbf{A} \subseteq \mathbf{G}$  and any  $\mathbf{B} \subseteq \mathbf{M}$  we have

$$I_{\#}(\mathbf{A}) \supseteq \mathbf{B} \iff \mathbf{A} \subseteq I_b(\mathbf{B}). \quad (16)$$

This equivalence formalizes the (nearly tautological-seeming) statement that a set **B** is contained in the largest set of attributes which apply to all members of **A** (meaning: the attributes **B** apply to all elements of **A**) if, and only if, **A** is contained in the largest set of things to which all attributes **B** apply (again meaning: the

attributes  $\mathbf{B}$  apply to all elements of  $\mathbf{A}$ ).

Despite seeming tautological, we can use the equivalence (16) to make non-trivial observations. One consequence of (16) is that the monotone maps

$$I_{\#} \circ I_b : \langle \mathbf{Pow} \mathbf{G}, \subseteq \rangle \rightarrow \langle \mathbf{Pow} \mathbf{G}, \subseteq \rangle \quad (17)$$

and

$$I_b^{\text{op}} \circ I_{\#}^{\text{op}} : \langle \mathbf{Pow} \mathbf{M}, \subseteq \rangle \rightarrow \langle \mathbf{Pow} \mathbf{M}, \subseteq \rangle. \quad (18)$$

are examples of what are called a *closure operator* and *interior operator*, respectively.

## Closure and interior operators

### Definition 33.1

Let  $\mathbf{P} = \langle \mathbf{P}, \leq \rangle$  be poset. A *closure operator* on  $\mathbf{P}$  is

#### Constituents

1. a monotone map  $f : \mathbf{P} \rightarrow \mathbf{P}$ ;

#### Conditions

1. Extensivity:  $x \leq f(x) \quad \forall x \in \mathbf{P}$ ;
2. Idempotence:  $f(f(x)) = f(x) \quad \forall x \in \mathbf{P}$ .

### Definition 33.2

Let  $\mathbf{P} = \langle \mathbf{P}, \leq \rangle$  be poset. An *interior operator* on  $\mathbf{P}$  is

#### Constituents

1. a monotone map  $f : \mathbf{P} \rightarrow \mathbf{P}$ ;

#### Conditions

1. Intensivity:  $f(x) \leq x \quad \forall x \in \mathbf{P}$ ;
2. Idempotence:  $f(f(x)) = f(x) \quad \forall x \in \mathbf{P}$ .

The notions of closure and interior operator are dual in the following sense.

**Lemma 33.3.** If  $f : \mathbf{P} \rightarrow \mathbf{P}$  is a closure (interior) operator, then  $f^{\text{op}} : \mathbf{P}^{\text{op}} \rightarrow \mathbf{P}^{\text{op}}$  is an interior (closure) operator.

In this section, for simplicity, we will work mainly in terms of closure operators.

**Lemma 33.4.** The monotone maps

$$I_{\#} \circ I_b : \langle \mathbf{Pow} \mathbf{G}, \subseteq \rangle \rightarrow \langle \mathbf{Pow} \mathbf{G}, \subseteq \rangle \quad (19)$$

and

$$I_b^{\text{op}} \circ I_{\#}^{\text{op}} : \langle \mathbf{Pow} \mathbf{M}, \subseteq \rangle \rightarrow \langle \mathbf{Pow} \mathbf{M}, \subseteq \rangle. \quad (20)$$

are closure operators.

*Proof.* Let's check that  $I_{\#} \circ I_b : \langle \mathbf{Pow} \mathbf{G}, \subseteq \rangle \rightarrow \langle \mathbf{Pow} \mathbf{G}, \subseteq \rangle$  is a closure operator, using (16). We omit the proof for  $I_b^{\text{op}} \circ I_{\#}^{\text{op}}$ , which may be done analogously.

To show the first condition in the definition of closure operator, fix a set of things  $\mathbf{A} \subseteq \mathbf{G}$ . In the situation of (16), choose  $\mathbf{B} = I_{\#}(\mathbf{A})$ . Since  $I_{\#}(\mathbf{A}) \supseteq I_{\#}(\mathbf{A})$  is true, (16) implies that  $\mathbf{A} \subseteq I_b(I_{\#}(\mathbf{A})) = (I_{\#} \circ I_b)(\mathbf{A})$ .

Now let's consider the second condition. Applying the monotone map  $I_{\#} \circ I_b$

to the relation  $\mathbf{A} \subseteq (I_{\#} \circ I_b)(\mathbf{A})$ , we have

$$(I_{\#} \circ I_b)(\mathbf{A}) \subseteq (I_{\#} \circ I_b \circ I_{\#} \circ I_b)(\mathbf{A}). \quad (21)$$

Thus we are finished when we show the inclusion in the other direction. By the first condition, we know that

$$I_{\#}(\mathbf{A}) \subseteq I_{\#}(I_b(I_{\#}(\mathbf{A}))) = (I_{\#} \circ I_b \circ I_{\#})(\mathbf{A}). \quad (22)$$

Applying the order-reversing map  $I_b$  to both sides of this inclusion then gives

$$(I_{\#} \circ I_b)(\mathbf{A}) \supseteq (I_{\#} \circ I_b \circ I_{\#} \circ I_b)(\mathbf{A}) \quad (23)$$

as desired.  $\square$

Closure and interior operators arise in various contexts in mathematics. Often we are interested in the elements which are in the images of these operators. These are called *closed elements* and *open elements*, respectively. We will use the term *fixed-points* to refer to both of these cases without needing to specify whether we are working with a closure or an interior operator.

### Definition 33.5

Let  $\mathbf{P} = \langle \mathbf{P}, \leq \rangle$  be a poset,  $f : \mathbf{P} \rightarrow \mathbf{P}$  a monotone map, and  $x \in \mathbf{P}$  an arbitrary element of  $\mathbf{P}$ .

If  $f$  is a closure (interior) operator, then  $f(x) \in \mathbf{P}$  is called the *closure* (*interior*) of  $x$ , and  $x$  is called *closed* (*open*) if  $f(x) = x$ . In both cases, when  $f(x) = x$ , we say that  $x$  is a fixed-point of  $f$ .

The set of fixed-points of  $f$  will be denoted  $\mathbf{P}_f$ , or by  $\mathbf{P}_{\text{fix}}$  when the operator  $f$  in question is clear.

**Remark 33.6.** Note that if  $f : \mathbf{P} \rightarrow \mathbf{P}$  is a closure or interior operator, then the set of fixed points  $\mathbf{P}_{\text{fix}}$  coincides with the image of  $f$ .

On the one hand, any element  $y$  of the form  $y = f(x)$  is a fixed-point, because

$$f(f(x)) = f(x) \quad (24)$$

by the idempotence property.

On the other hand, if  $y \in \mathbf{P}$  is a fixed point, then by definition  $y = f(y)$  is in the image of  $f$ .

**Remark 33.7.** If  $x$  is a fixed-point of a closure/interior operator  $f : \mathbf{P} \rightarrow \mathbf{P}$ , then  $x$  is also a fixed-point of  $f^{\text{op}} : \mathbf{P}^{\text{op}} \rightarrow \mathbf{P}^{\text{op}}$ .

Returning now to formal concept analysis, let's look at closures and closed elements for the closure operators

$$I_{\#} \circ I_b : \langle \text{Pow } \mathbf{G}, \subseteq \rangle \rightarrow \langle \text{Pow } \mathbf{G}, \subseteq \rangle$$

and

$$I_b^{\text{op}} \circ I_{\#}^{\text{op}} : \langle \text{Pow } \mathbf{M}, \subseteq \rangle \rightarrow \langle \text{Pow } \mathbf{M}, \subseteq \rangle$$

in terms of our simple example.

For example, let

$$\mathbf{A} = \{\text{classic car}, \text{electric car}\}. \quad (25)$$

Then

$$I_{\#}(\mathbf{A}) = \{\text{fast}\} \quad (26)$$



and

$$I_b(I_{\#}(\mathbf{A})) = I_b(\{\text{fast}\}) = \{\text{classic car, hybrid car, electric car}\}. \quad (27)$$

So  $\mathbf{A} = \{\text{classic car, electric car}\}$  is not a closed element of  $\langle \text{Pow } \mathbf{G}, \subseteq \rangle$ . Its closure contains the element “hybrid car” which is not in  $\mathbf{A}$ .

Or consider

$$\mathbf{B} = \{\text{electric, muscle}\}. \quad (28)$$

Then

$$I_b(\mathbf{B}) = \{\text{hybrid bike}\} \quad (29)$$

and

$$I_{\#}(I_b(\mathbf{B})) = I_{\#}(\{\text{hybrid bike}\}) = \{\text{electric, muscle}\} = \mathbf{B}. \quad (30)$$

We find here that  $\{\text{hybrid bike}\}$  is a closed element of  $\langle \text{Pow } \mathbf{M}, \subseteq \rangle$ .

In general, given a set  $\mathbf{A}$  of things, its closure  $(I_{\#} \circ I_b)(\mathbf{A})$  is the largest set of things that share the attributes in  $I_{\#}(\mathbf{A})$ . And  $I_{\#}(\mathbf{A})$  is the largest set of attributes shared by  $\mathbf{A}$ . Thus we may say:

“( $I_{\#} \circ I_b$ )( $\mathbf{A}$ ) is the maximal set of things that share the same attributes as are shared by  $\mathbf{A}$ .”

Or, put another way, taking the closure of  $\mathbf{A}$  is a way of enlarging  $\mathbf{A}$  without decreasing the set of associated shared attributes. Closing  $\mathbf{A}$  is adding those things to  $\mathbf{A}$  that come “for free” in the sense that, by adding them, we are not losing shared attributes.

A similar point of view of course also applies to closing sets of attributes with respect to the closure operator  $I_b \circ I_{\#}$ .

## Concepts

### Definition 33.8

Let  $\langle \mathbf{G}, \mathbf{M}, I \rangle$  be a formal context in the sense of formal concept analysis. A *concept* is a pair  $\langle \mathbf{A}, \mathbf{B} \rangle \in \text{Pow } \mathbf{G} \times \text{Pow } \mathbf{M}$  such that

$$I_{\#}(\mathbf{A}) = \mathbf{B} \quad \text{and} \quad I_b(\mathbf{B}) = \mathbf{A}. \quad (31)$$

For a concept  $\langle \mathbf{A}, \mathbf{B} \rangle$ , the set  $\mathbf{A}$  of things is called the *extent* of the concept, and the set  $\mathbf{B}$  of attributes is called the *intent* of the concept.

We denote the set of all concepts for the context  $\langle \mathbf{G}, \mathbf{M}, I \rangle$  by  $\mathcal{B}\langle \mathbf{G}, \mathbf{M}, I \rangle$ . (Here “ $\mathcal{B}$ ” comes from the German term “Begriffe”.)

The set  $\mathcal{B}\langle \mathbf{G}, \mathbf{M}, I \rangle$  of concepts for a formal context has a natural partial order structure. We set

$$\langle \mathbf{A}_1, \mathbf{B}_1 \rangle \leq \langle \mathbf{A}_2, \mathbf{B}_2 \rangle \quad (32)$$

if  $\mathbf{A}_1 \subseteq \mathbf{A}_2$  and  $\mathbf{B}_1 \supseteq \mathbf{B}_2$ . (In fact, by the definition of a concept, if one of the latter inclusions holds, then so must the other, so we only need to require one of them.) When (32) holds, we say that  $\langle \mathbf{A}_1, \mathbf{B}_1 \rangle$  is a *subconcept* of  $\langle \mathbf{A}_2, \mathbf{B}_2 \rangle$ .

**Lemma 33.9.** If  $\langle \mathbf{A}, \mathbf{B} \rangle$  is a concept, then  $\mathbf{A}$  and  $\mathbf{B}$  are closed elements of  $\langle \text{Pow } \mathbf{G}, \subseteq \rangle$  and  $\langle \text{Pow } \mathbf{M}, \subseteq \rangle$ , respectively.

*Proof.* For  $\mathbf{A}$  we have

$$I_b(I_{\#}(\mathbf{A})) = I_b(\mathbf{B}) = \mathbf{A} \quad (33)$$

using both the equations (31). The case for  $\mathbf{B}$  is analogous.  $\square$

**Lemma 33.10.** If  $\mathbf{A} \in \langle \text{Pow } \mathbf{G}, \subseteq \rangle$  is closed, then  $I_{\#}(\mathbf{A})$  is closed and  $\langle \mathbf{A}, I_{\#}(\mathbf{A}) \rangle$  is a concept.

Similarly, if  $\mathbf{B} \in \langle \text{Pow } \mathbf{M}, \subseteq \rangle$  is closed, then  $I_b(\mathbf{B})$  is closed and  $\langle I_b(\mathbf{B}), \mathbf{B} \rangle$  is a concept.

*Proof.* We show only the first statement. We have

$$(I_b ; I_{\#})(I_{\#}(\mathbf{A})) = I_{\#}(I_b(I_{\#}(\mathbf{A}))) = I_{\#}((I_{\#} ; I_b)(\mathbf{A})) = I_{\#}(\mathbf{A}), \quad (34)$$

so  $I_{\#}(\mathbf{A})$  is closed. That  $\langle \mathbf{A}, I_b(\mathbf{A}) \rangle$  is a concept is clear, since  $I_b(I_{\#}(\mathbf{A})) = \mathbf{A}$ .  $\square$

**Lemma 33.11.** The posets of fixed points  $\langle \text{Pow } \mathbf{G}_{\text{fix}}, \subseteq \rangle$  and  $\langle \text{Pow } \mathbf{M}_{\text{fix}}, \subseteq \rangle^{\text{op}}$  are isomorphic via the restrictions of  $I_{\#}$  and  $I_b$ , and each is isomorphic to the poset  $\langle \mathcal{B}(\mathbf{G}, \mathbf{M}, I), \leq \rangle$  via its projections onto its first and second factors, respectively.

*Proof.* This follows from Lemma 33.9, Lemma 33.10, and the definition of the ordering on  $\langle \mathcal{B}(\mathbf{G}, \mathbf{M}, I), \leq \rangle$ .  $\square$

### 33.2. Galois connections

**Definition 33.12** (Monotone Galois Connection)

A (monotone) *Galois connection* between posets  $\mathbf{P}$  and  $\mathbf{Q}$  is a pair of monotone maps  $f : \mathbf{P} \rightarrow \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow \mathbf{P}$  such that for all  $p \in \mathbf{P}, q \in \mathbf{Q}$ :

$$\underline{\underline{f(p) \leq_Q q}} \quad . \quad \underline{\underline{p \leq_P g(q)}} \quad . \quad (35)$$

In this case  $f$  is called the *left adjoint* (or *lower adjoint*) and  $g$  is called the *right adjoint* (or *upper adjoint*). We use the short-hand notation  $f \dashv g$  to say that  $f$  and  $g$  form a Galois connection, or we draw a globular diagram like so:

$$\begin{array}{ccc} & f & \\ \mathbf{P} & \xrightarrow{\quad} & \mathbf{Q} \\ & g & \\ & \perp & \end{array} \quad (36)$$

**Lemma 33.13.** Monotone maps  $f : \mathbf{P} \rightarrow \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow \mathbf{P}$  form a Galois connection if and only if the following hold:

1.  $p \leq_P g(f(p)) \quad \forall p \in \mathbf{P};$
2.  $f(g(q)) \leq_Q q \quad \forall q \in \mathbf{Q}.$

**Definition 33.14** (Antitone Galois Connection)

An *antitone Galois connection* between  $\mathbf{P}$  and  $\mathbf{Q}$  is a pair of antitone maps  $f : \mathbf{P} \rightarrow \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow \mathbf{P}$  such that for all  $p \in \mathbf{P}, q \in \mathbf{Q}$ :

$$\underline{\underline{q \leq_Q f(p)}} \quad . \quad \underline{\underline{p \leq_P g(q)}} \quad . \quad (37)$$

**Remark 33.15.** The underlying function of an antitone map  $f : \mathbf{P} \rightarrow \mathbf{Q}$  defines a monotone map  $f : \mathbf{P} \rightarrow \mathbf{Q}^{\text{op}}$  (or a monotone map  $f : \mathbf{P}^{\text{op}} \rightarrow \mathbf{Q}$ ). Every antitone Galois connection  $f : \mathbf{P} \rightarrow \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow \mathbf{P}$  defines a Galois connection  $f : \mathbf{P} \rightarrow \mathbf{Q}^{\text{op}}$  and  $g : \mathbf{Q}^{\text{op}} \rightarrow \mathbf{P}$ .

Because of the above remark, and because we prefer to work with monotone maps (since they are morphisms of posets), we will mainly focus on (monotone) Galois connections. However, it is useful to be aware of the antitone definition, since it is sometimes used in the literature and sometimes more natural in the context of certain examples.

**Lemma 33.16.** Antitone maps  $f : \mathbf{P} \rightarrow \mathbf{Q}$  and  $g : \mathbf{Q} \rightarrow \mathbf{P}$  form an antitone Galois connection if and only if the following hold:

1.  $p \leq_P g(f(p)) \quad \forall p \in \mathbf{P};$
2.  $q \leq_Q f(g(q)) \quad \forall q \in \mathbf{Q}.$

## Examples

### Induced closure and interior operators

**Lemma 33.17.** If  $f : P \rightarrow Q$  and  $g : Q \rightarrow P$  form a Galois connection, then

$$f \circ g : P \rightarrow P \quad (38)$$

is a closure operator and

$$g \circ f : Q \rightarrow Q \quad (39)$$

is an interior operator.

### 33.3. Adjunctions: hom-set definition

In this section we give a definition of adjunction which can be viewed as an analogy with the following situation in linear algebra. Suppose  $V$  and  $W$  are finite-dimensional real vector spaces, equipped with inner products  $(-, -)_V$  and  $(-, -)_W$ , respectively. The adjoint of a linear map  $F : V \rightarrow W$  is a linear map  $F^* : W \rightarrow V$  such that

$$(Fv, w)_W = (v, F^*w)_V, \quad \forall v \in V, w \in W. \quad (40)$$

**Definition 33.18** (Adjunction, Version 1)

Let  $\mathbf{C}$  and  $\mathbf{D}$  be categories. An *adjunction* from  $\mathbf{C}$  to  $\mathbf{D}$  is given by the following data:

1. A functor  $L : \mathbf{C} \rightarrow \mathbf{D}$ , called the *left adjoint*;
2. A functor  $R : \mathbf{D} \rightarrow \mathbf{C}$ , called the *right adjoint*;
3. A natural isomorphism  $\tau : \text{Hom}_{\mathbf{D}}(L-, -) \Rightarrow \text{Hom}_{\mathbf{C}}(-, R-)$  between functors  $\mathbf{C}^{\text{op}} \times \mathbf{D} \rightarrow \mathbf{Set}$ .

We use the notation  $L \dashv R$  to indicate that  $L$  and  $R$  form an adjunction, with  $L$  the left adjoint and  $R$  the right adjoint.

### 33.4. Adjunctions: (co)unit definition

Recall from Def. 35.1: in a category  $\mathbf{C}$ , a morphism  $f : X \rightarrow Y$  is an isomorphism if there exists a morphism  $g : Y \rightarrow X$  such that

$$f \circ g = \text{id}_X \quad \text{and} \quad g \circ f = \text{id}_Y. \quad (41)$$

Now let's think about this definition in the case where  $\mathbf{C}$  is the category  $\mathbf{Cat}$  of categories. We will consider weakenings of the notion of isomorphism in this setting, and this will lead to a second (but equivalent) definition of adjunction. The precise relationship between the two definitions will be spelled out Section 33.7.

The idea of “weakening” the notion of isomorphism of categories is as follows. Given functors

$$\begin{array}{ccc} & L & \\ \mathbf{C} & \xrightarrow{\quad} & \mathbf{D} \\ & R & \end{array} \quad (42)$$

instead of requiring the equations

$$\text{id}_{\mathbf{C}} = L \circ R \quad \text{and} \quad R \circ L = \text{id}_{\mathbf{D}}, \quad (43)$$

we replace the equality symbols with 2-morphisms, in this way:

$$\text{id}_{\mathbf{C}} \xRightarrow{\text{un}} L \circ R \quad \text{and} \quad R \circ L \xRightarrow{\text{co}} \text{id}_{\mathbf{D}}. \quad (44)$$

The last two relationships can also be depicted in the following more geometric manner:

$$\begin{array}{ccc} & \mathbf{D} & \\ L \nearrow & & \searrow R \\ \mathbf{C} & & \mathbf{C} \\ & \text{id}_{\mathbf{C}} \searrow & \\ & \uparrow \text{un} & \end{array} \quad , \quad \begin{array}{ccc} & \mathbf{C} & \\ R \nearrow & & \searrow L \\ \mathbf{D} & & \mathbf{D} \\ & \text{id}_{\mathbf{D}} \searrow & \\ & \downarrow \text{co} & \end{array} \quad (45)$$

**Definition 33.19** (Equivalence of categories)

Let  $\mathbf{C}$  and  $\mathbf{D}$  be categories. An *equivalence of categories* between  $\mathbf{C}$  and  $\mathbf{D}$  is the following data:

1. A functor  $L : \mathbf{C} \rightarrow \mathbf{D}$ ;
2. A functor  $R : \mathbf{D} \rightarrow \mathbf{C}$ ;
3. Natural isomorphisms  $\text{un} : \text{id}_{\mathbf{C}} \Rightarrow L \circ R$  and  $\text{co} : R \circ L \Rightarrow \text{id}_{\mathbf{D}}$ .

**Definition 33.20** (Adjunction, Version 2)

Let  $\mathbf{C}$  and  $\mathbf{D}$  be categories. An *adjunction* from  $\mathbf{C}$  to  $\mathbf{D}$  is given by the following data, satisfying the following conditions.

Data:

1. A functor  $L : \mathbf{C} \rightarrow \mathbf{D}$  (the *left adjoint*);
2. A functor  $R : \mathbf{D} \rightarrow \mathbf{C}$  (the *right adjoint*);
3. Two natural transformations  $\text{un} : \text{id}_{\mathbf{C}} \Rightarrow L \circ R$  and  $\text{co} : R \circ L \Rightarrow \text{id}_{\mathbf{D}}$

Conditions:

1. For all objects  $X$  of  $\mathbf{C}$ , it holds that

$$L\mathbf{un}_X \circ \mathbf{co}_{LX} = \mathbf{id}_{LX} \quad \text{and} \quad \mathbf{un}_{RY} \circ R\mathbf{co}_Y = \mathbf{id}_{RY}, \quad (46)$$

which means that the following diagrams commute:

$$\begin{array}{ccc} LX & \xrightarrow{L\mathbf{un}_X} & LRLX \\ & \searrow \mathbf{id}_{LX} & \downarrow \mathbf{co}_{LX} \\ & & LX \end{array} \qquad \begin{array}{ccc} RY & \xrightarrow{\mathbf{un}_{RY}} & RLRY \\ & \searrow \mathbf{id}_{RY} & \downarrow R\mathbf{co}_Y \\ & & RY \end{array} \quad (47)$$

The natural transformations  $\mathbf{un}$  and  $\mathbf{co}$  are called the *unit* and *counit* of the adjunction.

**Definition 33.21** (Adjoint equivalence)

An adjunction is called an *adjoint equivalence* if the unit and counit are natural isomorphisms.

**Remark 33.22.** The conditions (triangle identities) from Def. 33.20 are “hidden” in Def. 33.18 in the condition that  $\tau$  be a natural isomorphism. In Section 33.7 we spell out how the two definitions are related.

### 33.5. Example of a “Product-Hom” adjunction

We will consider an adjunction between the category **Set** and itself which is a basic representative of a certain “type” of adjunction that appears all over mathematics. This type of adjunction might be called a “Product-Hom” adjunction.

Fix a set **B** and consider the functors **F** and **G** which act as follows. Given a set **A**,

$$F(\mathbf{A}) = \mathbf{B} \times \mathbf{A} \quad (48)$$

and

$$G(\mathbf{A}) = \text{Hom}_{\mathbf{Set}}(\mathbf{B}; \mathbf{A}) =: \mathbf{A}^{\mathbf{B}}. \quad (49)$$

Given a morphism  $f : \mathbf{A} \rightarrow \mathbf{A}'$ ,

$$F(f) = \text{id}_{\mathbf{B}} \times f \quad (50)$$

and

$$\begin{aligned} G(f) : \mathbf{A}^{\mathbf{B}} &\rightarrow \mathbf{A}'^{\mathbf{B}} \\ g &\mapsto g \circ f. \end{aligned} \quad (51)$$

These functors are part of an adjunction

$$\begin{array}{ccc} & \mathbf{B} \times - & \\ \text{Set} & \begin{array}{c} \curvearrowright \\ \perp \\ \curvearrowleft \end{array} & \text{Set} \\ & (-)^{\mathbf{B}} & \end{array} \quad (52)$$

In terms of Def. 33.18, there is a natural isomorphism

$$\tau : \text{Hom}_{\mathbf{Set}}(F(-); -) \Longrightarrow \text{Hom}_{\mathbf{Set}}(-; G(-)) \quad (53)$$

whose component at  $\langle \mathbf{A}, \mathbf{C} \rangle$  is the isomorphism

$$\tau_{\mathbf{A}, \mathbf{C}} : \text{Hom}_{\mathbf{Set}}(\mathbf{B} \times \mathbf{A}; \mathbf{C}) \rightarrow \text{Hom}_{\mathbf{Set}}(\mathbf{A}; \mathbf{C}^{\mathbf{B}}) \quad (54)$$

given by “partial evaluation”. Namely, given  $f : \mathbf{B} \times \mathbf{A} \rightarrow \mathbf{C}$ , this is mapped by  $\tau_{\mathbf{A}, \mathbf{C}}$  to the function  $\tau f : \mathbf{A} \rightarrow \mathbf{C}^{\mathbf{B}}, a \mapsto f(-, a)$ .

In terms of Def. 33.20, the component at **A** of the unit and co-unit, respectively, are

$$\begin{aligned} \text{un}_{\mathbf{A}} : \mathbf{A} &\rightarrow (\mathbf{B} \times \mathbf{A})^{\mathbf{B}} \\ a &\mapsto (b \mapsto \langle a, b \rangle) \end{aligned} \quad (55)$$

and

$$\begin{aligned} \text{co}_{\mathbf{A}} : \mathbf{B} \times (\mathbf{A}^{\mathbf{B}}) &\rightarrow \mathbf{A} \\ \langle b, f \rangle &\mapsto f(b) \end{aligned} \quad (56)$$



### 33.6. Example of a “Free-Forgetful” adjunction

Another “type” of adjunction that appears frequently can be called a “Free-Forgetful” adjunction. Such adjunctions are composed of a “free functor” and a “forgetful functor”. These terms are informal, but the idea is as follows.

A free functor  $\mathbf{C} \rightarrow \mathbf{D}$  typically takes an object  $X$  of  $\mathbf{C}$  and “freely” adds some structure to it. “Free” means that only those structures and conditions are added that are absolutely necessary to make  $X$  an object of  $\mathbf{D}$ , and otherwise the functor does not impose any constraints or relations.

Conversely, a “forgetful functor” usually starts from an object  $Y$  on  $\mathbf{D}$  which has some structure, and “forgets” some of this structure, which results in us being able to view  $Y$  as an object in  $\mathbf{C}$ .

**Example 33.23.** Any real vector space is built from an underlying set, together with extra structure given by operations (vector addition and scalar multiplication). There is a forgetful functor

$$\mathbf{Vect}_{\mathbb{R}} \rightarrow \mathbf{Set} \quad (57)$$

which maps any vector space to its underlying set of vectors. On the other hand, there is a “free” functor

$$\mathbf{Set} \rightarrow \mathbf{Vect}_{\mathbb{R}}. \quad (58)$$

Given a set  $A$ , we can build the “free real vector space generated by  $A$ ”. To do this, we think of the elements of  $A$  as basis vectors, and we build a vector space by taking formal finite  $\mathbb{R}$ -linear combinations of them.

In the following we consider an example in detail where we “freely” generate a category from a directed graph.

**Example 33.24.** Let  $\mathbf{Grph}$  be the category of directed graphs and  $\mathbf{Cat}$  the category of (small) categories.

There is a functor  $F : \mathbf{Grph} \rightarrow \mathbf{Cat}$  which turns any directed graph  $D = \langle V, E, s, t \rangle$  into a category whose objects are the vertices  $V$  and whose morphisms are finite directed paths between vertices. This is called the *free category generated by the graph  $D$*  (Section 13.6).

There is also a functor  $G : \mathbf{Cat} \rightarrow \mathbf{Grph}$  which turns a category  $C$  into a graph where the set of vertices is  $\mathbf{Ob}_C$  and there is a directed edge between vertices for every morphism in  $C$  between the corresponding vertices.

Let’s first describe this adjunction via Def. 33.18. The natural isomorphism

$$\tau : \mathbf{Hom}_{\mathbf{Cat}}(F(-); -) \Rightarrow \mathbf{Hom}_{\mathbf{Grph}}(-; G(-)) \quad (59)$$

is the one whose component at  $\langle D, C \rangle$  is the isomorphism

$$\tau_{D,C} : \mathbf{Hom}_{\mathbf{Cat}}(F(D); C) \Rightarrow \mathbf{Hom}_{\mathbf{Grph}}(D; G(C)) \quad (60)$$

which assigns to any functor  $F : F(D) \rightarrow C$  the morphism of graphs  $D : G(C)$  given by restricting  $F$  to  $D$  and only keeping track of its action on vertices and edges (in other words, we ignore its compositional properties and think of it just as a graph morphism).

Now let’s consider this adjunction from the perspective of Def. 33.20. The component at  $D$  of the counit is the morphism of graphs

$$\mathbf{un}_D : D \rightarrow G(F(D)) \quad (61)$$

which includes  $D$  into the graph  $G(F(D))$ . The latter has an edge from the source

to the target of every finite path in  $D$ . The paths of length zero are what corresponded to identity morphisms in  $F(D)$ , and the paths of length one constitute a copy of  $D$  inside  $G(F(D))$ .

What does the unit look like? Its component at  $\mathbf{C}$  is a functor

$$\text{co}_{\mathbf{C}} : F(G(\mathbf{C})) \rightarrow \mathbf{C}. \quad (62)$$

The category  $F(G(\mathbf{C}))$  is larger than  $\mathbf{C}$ : starting with  $\mathbf{C}$ , the graph  $G(\mathbf{C})$  will contain edges for all the morphisms in  $\mathbf{C}$ , but it will forget their compositional interlinking. In particular, for example, it will forget which loops denote identity morphisms (in other words, which morphisms act neutrally) and, more generally, it will forget when different compositions of morphism give the same result. In  $F(G(\mathbf{C}))$ , then, morphism compositions that might have given the same result in  $\mathbf{C}$  will now be distinct. The functor  $\text{co}_{\mathbf{C}}$  in a sense “remembers” those relations that were true in  $\mathbf{C}$  and it “implements” them by “projecting”  $F(G(\mathbf{C}))$  back to  $\mathbf{C}$ .

### 33.7. Relating the two definitions

We start first with the “hom-set definition” of adjunction, and show how to obtain the “(co)unit definition”. Given an adjunction  $F \dashv G$  from a category  $\mathbf{C}$  to a category  $\mathbf{D}$ , we have, by Def. 33.18 a natural isomorphism  $\tau$  with components

$$\tau_{X,Y} : \text{Hom}_{\mathbf{D}}(F(X); Y) \rightarrow \text{Hom}_{\mathbf{C}}(X; G(Y)). \quad (63)$$

From this data we can construct the unit and counit of the adjunction as follows.

Given an object  $X$  of  $\mathbf{C}$ , we define

$$\eta_X : X \rightarrow G(F(X)) \quad (64)$$

to be the image under  $\tau_{X,F(X)}$  of  $\text{id}_{F(X)} \in \text{Hom}_{\mathbf{D}}(F(X); F(X))$ .

Given an object  $Y$  of  $\mathbf{D}$ , we define

$$\varepsilon_Y : F(G(Y)) \rightarrow Y \quad (65)$$

to the image under  $\tau_{G(Y),Y}^{-1}$  of  $\text{id}_{G(Y)} \in \text{Hom}_{\mathbf{D}}(G(Y); G(Y))$ .

**Exercise 47.** Show that if we define  $\eta$  and  $\varepsilon$  in terms of their components as above, then they do indeed define natural transformations

$$\eta : \text{id}_{\mathbf{C}} \Rightarrow F \circ G \quad (66)$$

and

$$\varepsilon : F \circ G \Rightarrow \text{id}_{\mathbf{D}} \quad (67)$$

respectively. In other words, check the naturality conditions for  $\eta$  and  $\varepsilon$ .

*See solution on page 513.*

**Exercise 48.** Show that  $\eta$  and  $\varepsilon$ , as defined above, satisfy the triangle identities stated in Def. 33.20.

*See solution on page 513.*

Now let’s start with the “(co)unit definition” of adjunction and see how to obtain the “hom-set definition”.

Given the unit  $\eta$  and counit  $\varepsilon$ , we can construct the components  $\tau_{X,Y}$  of the natural transformation  $\tau$  as follows. Given  $f \in \text{Hom}_{\mathbf{D}}(F(X), Y)$ , we define

$$\tau_{X,Y}(f) = \eta_X \circ G(f). \quad (68)$$

Similarly, given  $g \in \text{Hom}_{\mathbf{C}}(X, G(Y))$ , the inverse component is given by

$$\tau_{X,Y}^{-1}(g) = F(g) \circ \varepsilon_Y. \quad (69)$$

**Exercise 49.** Show that  $\tau_{X,Y}$  and  $\tau_{X,Y}^{-1}$  are indeed functions which are inverses of each other.

*See solution on page 513.*

**Exercise 50.** Show that the functions  $\tau_{X,Y}$  do assemble to a natural transformation

$$\tau : \text{Hom}_{\mathbf{D}}(F(-), -) \Rightarrow \text{Hom}_{\mathbf{C}}(-, G(-)) \quad (70)$$

between functors  $\mathbf{C}^{\text{op}} \times \mathbf{D} \rightarrow \mathbf{Set}$ .

*See solution on page 513.*

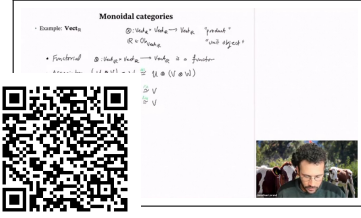


## 34. Traced Monoidal categories

This chapter describes the classical notions of monoidal categories and traced monoidal categories. In our exposition in Chapter 30, we assumed that the categories were associative stacking. That is,  $f \circ (g \circ h) = (f \circ g) \circ h$ . One can construct the notion of parallelism with less strict conditions, by asking only that the two results are related by an isomorphism:  $f \circ (g \circ h) \simeq (f \circ g) \circ h$ . This requires bookkeeping of natural transformations to describe the needed isomorphisms.

<b>34.1 Monoidal categories . . . . .</b>	<b>496</b>
<b>34.2 Examples of monoidal categories</b>	<b>499</b>
<b>34.3 Symmetric monoidal categories</b>	<b>501</b>
<b>34.4 Monoidal functors . . . . .</b>	<b>503</b>
<b>34.5 Dual objects and morphisms . .</b>	<b>505</b>
<b>34.6 Trace for monoidal categories . .</b>	<b>507</b>

Watch: *Definition of monoidal categories* (8 minutes).



## 34.1. Monoidal categories

### Definition 34.1 (Monoidal category)

A *monoidal structure* on a category  $\mathbf{C}$  is specified by:

#### Constituents

1. A functor  $\otimes : \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}$ , called the *monoidal product*.
2. An object  $\mathbf{1} \in \text{Ob}_{\mathbf{C}}$ , called the *monoidal unit*.
3. A natural isomorphism, called the *associator*, whose components are of the type

$$\text{as}_{X,Y,Z} : (X \otimes Y) \otimes Z \xrightarrow{\text{ir}} X \otimes (Y \otimes Z) \quad X, Y, Z \in \text{Ob}_{\mathbf{C}}. \quad (1)$$

4. A natural isomorphism, called the *left unitor*, whose components are of the type

$$\text{lu}_X : \mathbf{1} \otimes X \xrightarrow{\text{ir}} X \quad X \in \text{Ob}_{\mathbf{C}}. \quad (2)$$

5. A natural isomorphism, called the *right unitor*, whose components are of the type

$$\text{ru}_X : X \otimes \mathbf{1} \xrightarrow{\text{ir}} X \quad X \in \text{Ob}_{\mathbf{C}}. \quad (3)$$

#### Conditions

For all  $X, Y, Z, U \in \text{Ob}_{\mathbf{C}}$ , the following diagrams must commute:

1. Triangle identities.

$$\begin{array}{ccc} (X \otimes \mathbf{1}) \otimes Y & \xrightarrow{\text{as}_{X,\mathbf{1},Y}} & X \otimes (\mathbf{1} \otimes Y) \\ & \searrow \text{ru}_X \otimes \text{id}_Y \quad \swarrow \text{id}_X \otimes \text{lu}_Y & \\ & X \otimes Y & \end{array} \quad (4)$$

2. Pentagon identities.

$$\begin{array}{ccccc} & & (X \otimes Y) \otimes (Z \otimes U) & & \\ & \text{as}_{X \otimes Y, Z, U} \nearrow & & \searrow \text{as}_{X, Y, Z \otimes U} & \\ ((X \otimes Y) \otimes Z) \otimes U & & & & (X \otimes (Y \otimes (Z \otimes U))) \\ \downarrow \text{as}_{X, Y, Z} \otimes \text{id}_U & & & & \uparrow \text{id}_X \otimes \text{as}_{Y, Z, U} \\ (X \otimes (Y \otimes Z)) \otimes U & \xrightarrow{\text{as}_{X, Y \otimes Z, U}} & X \otimes ((Y \otimes Z) \otimes U) & & \end{array} \quad (5)$$

A category equipped with a monoidal structure is called a *monoidal category*. If the components of the associator, left unitor, and right unitor are all equalities, one calls the category *strict monoidal*.

**Remark 34.2.** Note that in the constituents listed in Def. 34.1 we specified natural isomorphisms  $\text{as}$ ,  $\text{lu}$ , and  $\text{ru}$  simply in terms of their components. You may be wondering: which functors are the respective source and target of these natural transformations? Since it is a mouthful to write, this information is often left to be inferred from the components given. Let us quickly illustrate how to see, from the components, which functors are involved. Take, for example, the left unitor. Its components are

$$\text{lu}_X : \mathbf{1} \otimes X \xrightarrow{\text{ir}} X \quad X \in \text{Ob}_{\mathbf{C}}, \quad (6)$$

so, if  $F$  and  $G$  denote the functors which are the source and target of  $\text{lu}$ , the functor  $F$  must act on objects by  $F(X) = \mathbf{1} \otimes X$  and  $G$  must act by  $G(X) = X$ . The “obvious” or “canonical” choice then (given that we are considering *any* monoidal category) is that  $G$  is the identity functor and that  $F$  is the functor which acts on morphisms by mapping  $f : X \rightarrow Y$  to

$$\text{id}_{\mathbf{1}} \otimes f : \mathbf{1} \otimes X \rightarrow \mathbf{1} \otimes Y. \quad (7)$$

Note that the components of the left unitor  $\text{lu}$  are indexed by one variable  $X \in \text{Ob}_{\mathbf{C}}$ , while the associator  $\text{as}$  is indexed by *three* variables! The associator is therefore a natural transformation between two functors of the type

$$\mathbf{C} \times \mathbf{C} \times \mathbf{C} \rightarrow \mathbf{C}. \quad (8)$$

Can you guess which functors of this type are meant in Def. 34.1 to be the source and target of  $\text{as}$ ?

**Example 34.3.** We digest the definition of monoidal category with an explanatory example. We consider the structure  $\langle \mathbf{Set}, \times, \mathbf{1} \rangle$  and show that it indeed forms a monoidal category. First, we specify how the monoidal product (cartesian product here) acts on objects and morphisms in  $\mathbf{Set}$  (it is a functor). Given  $\mathbf{A}, \mathbf{B} \in \text{Ob}_{\mathbf{Set}}$ ,  $\mathbf{A} \times \mathbf{B}$  is the cartesian product of sets, and given  $f : \mathbf{A} \rightarrow \mathbf{A}'$ ,  $g : \mathbf{B} \rightarrow \mathbf{B}'$ , we have:

$$\begin{aligned} (f \times g) : \mathbf{A} \times \mathbf{B} &\xrightarrow{\cong} \mathbf{A}' \times \mathbf{B}' \\ \langle a, b \rangle &\mapsto \langle f(a), g(b) \rangle. \end{aligned} \quad (9)$$

Furthermore, given any  $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \text{Ob}_{\mathbf{Set}}$ , we specify the associator  $\text{as}_{\mathbf{A}, \mathbf{B}, \mathbf{C}}$ :

$$\begin{aligned} \text{as}_{\mathbf{A}, \mathbf{B}, \mathbf{C}} : (\mathbf{A} \times \mathbf{B}) \times \mathbf{C} &\rightarrow \mathbf{A} \times (\mathbf{B} \times \mathbf{C}) \\ \langle \langle a, b \rangle, c \rangle &\mapsto \langle a, \langle b, c \rangle \rangle \end{aligned} \quad (10)$$

This defines an isomorphism (I can go “back and forth”, by switching the tuple separation). We now need to check whether  $\text{as}$  is natural. We check this graphically, using the commutative diagram in Fig. 1.

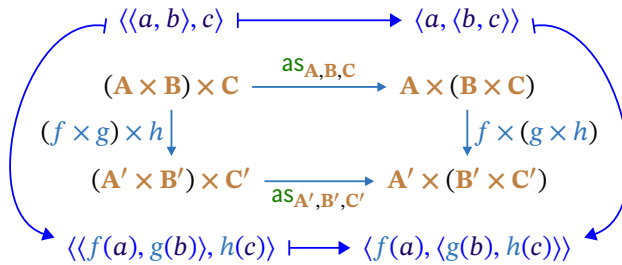


Figure 1.

Given an object  $\mathbf{A} \in \text{Ob}_{\mathbf{Set}}$ , the unitor  $\text{lu}_{\mathbf{A}}$  is given by:

$$\begin{aligned} \text{lu}_{\mathbf{A}} : \mathbf{1} \times \mathbf{A} &\xrightarrow{\cong} \mathbf{A} \\ \langle \bullet, a \rangle &\mapsto a. \end{aligned} \quad (11)$$

Again, this defines an isomorphism. Consider a morphism  $f : \mathbf{A} \rightarrow \mathbf{A}'$ . We now prove naturality graphically (Fig. 2).

Analogously, given an object  $\mathbf{A} \in \text{Ob}_{\mathbf{Set}}$ , the unitor isomorphism  $\text{ru}_{\mathbf{A}}$  is given by:

$$\begin{aligned} \text{ru}_{\mathbf{A}} : \mathbf{A} \times \mathbf{1} &\xrightarrow{\cong} \mathbf{A} \\ \langle a, \bullet \rangle &\mapsto a. \end{aligned} \quad (12)$$

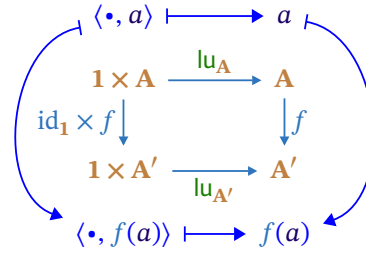


Figure 2.

The proof for naturality is analogous to the one of  $lu_A$ . We now need to check whether the triangle and pentagon identities are satisfied. We start by the triangle. Given  $A, B \in \mathbf{Ob}_{\mathbf{Set}}$ , the proof is displayed in Fig. 3.

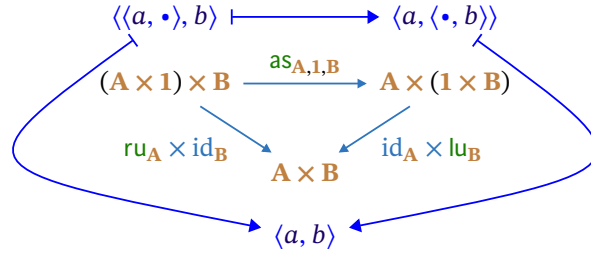


Figure 3.

We now prove the pentagon identity. Given  $A, B, C, D \in \mathbf{Ob}_{\mathbf{Set}}$ , the proof is reported in Fig. 4.

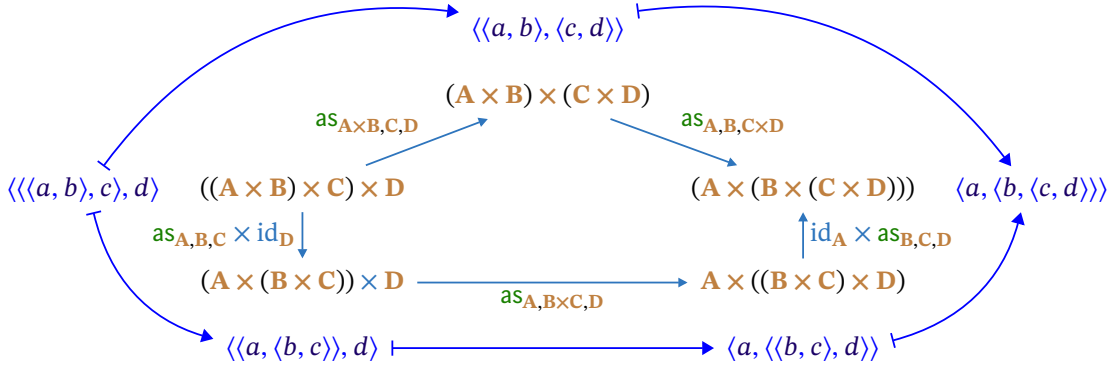


Figure 4.

**Remark 34.4** (We cannot define monoidal *semi*-categories). Note that in Def. 34.1 we used the category's identities in the diagrams. It is not possible to “patch” the definition to work with semicategories.

## 34.2. Examples of monoidal categories

**Example 34.5.** The category  $\mathbf{Vect}_{\mathbb{R}}$  can be equipped with a monoidal structure where the monoidal product is the tensor product of real vector spaces. It can also be equipped with a different monoidal structure where the monoidal product is the direct sum of real vector spaces.

**Example 34.6** (Robot configurations). Consider  $\mathbb{R}^2$ , discretized as a two-dimensional grid, representing locations (cells) which a robot can reach. The configuration space of the robot is  $\mathbb{R}^2 \times \Theta$ , where  $\Theta = [0, 2\pi)$ . A specific configuration  $\langle x, y, \theta \rangle$  is characterized at each time by the position of the robot  $x, y \in \mathbb{R}$  and its orientation  $\theta \in \Theta$ . The action space of the robot is  $\mathcal{A} = \{\text{stay}, \leftarrow, \rightarrow, \uparrow, \downarrow\}$ . This is a category, where each configuration of the robot is an object, and morphisms are robot actions which change configurations. Each configuration has the identity morphism which does not change it (stay). Composition of morphisms is given by concatenation of actions (Fig. 5). Assuming the existence of multiple robots  $r_i = \langle x_i, y_i, \theta_i \rangle$ , it is possible to define a product  $r_i \otimes r_j$ , which is to be intended as “we have a robot at configuration  $r_i$  and another one at configuration  $r_j$ ”. However, this cannot be a proper monoidal product, because two robots cannot have the same configuration (physically, they cannot lie on each other), and hence  $r_i \otimes r_i$  does not exist. By assuming that two robots could share the same configuration, this would be a valid monoidal product.

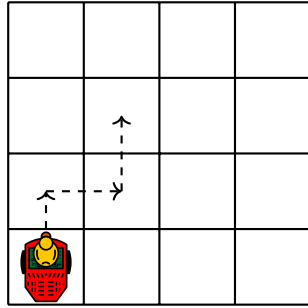


Figure 5.: Example of the robot category.

We will provide more examples of monoidal structures on the previous categories that we have seen.



### Graded exercise I.5 ( $\mathbf{VectTensorMonStructure}$ )

What are straightforward choices of monoidal unit, associator, and left/right unitors which, together with the tensor product as monoidal product, equip  $\mathbf{Vect}_{\mathbb{R}}$  with a monoidal structure?

In this exercise, simply write down how you think each of these pieces of data would be defined – it is not asked that you prove that they do indeed form a monoidal structure (that would be much more involved).

### Graded exercise I.6 ( $\mathbf{MonoidalProductVectDirectSum}$ )

Let  $\mathbf{C}$  denote the category of real vector spaces. Given real vector spaces  $X = \langle X, +_X, \cdot_X \rangle$  and  $Y = \langle Y, +_Y, \cdot_Y \rangle$ , their *direct sum*  $X \oplus Y$  is defined as follows. The underlying set of  $X \oplus Y$  is the cartesian product  $X \times Y$  of the underlying sets of  $X$  and  $Y$ . For  $X \oplus Y$ , vector addition is defined (using infix notation)



by

$$\langle x_1, y_1 \rangle + \langle x_2, y_2 \rangle := \langle x_1 +_X x_2, y_1 +_Y y_2 \rangle \quad x_1, x_2 \in X; y_1, y_2 \in Y; \quad (13)$$

and scalar multiplication is defined (using infix notation) by

$$\lambda \cdot \langle x, y \rangle := \langle \lambda \cdot_X x, \lambda \cdot_Y y \rangle, \quad \lambda \in \mathbb{R}; x \in X; y \in Y. \quad (14)$$

There is a monoidal structure  $(\otimes, 1, \text{as}, \text{lu}, \text{ru})$  on  $\mathbf{C}$  where the monoidal product  $\otimes$ , on objects, is the direct sum of real vector spaces. Your tasks:

1. Guess what the definition of  $\otimes$  on morphisms is, and check that  $\otimes$  really does define a functor.
2. Guess what the definitions of the monoidal unit, associator, and the unitors are for this monoidal product. Are the components of the associator and unitors really isomorphisms in  $\mathbf{C}$ ? Justify, briefly, why.
3. Check that the left-unitor is indeed a natural transformation. What are the functors that it maps between?
4. Check the coherence condition in the definition of a monoidal category that involves the unitors.

### 34.3. Symmetric monoidal categories

#### Definition 34.7 (Braided monoidal category)

A *braided monoidal category* is a monoidal category  $\langle \mathbf{C}, \otimes, \mathbf{1}, \text{as}, \text{lu}, \text{ru} \rangle$  equipped with a *braiding*, which is specified by

##### Constituents

1. A natural isomorphism  $\text{br}$ , called the *braiding*, whose components are of the type

$$\text{br}_{X,Y} : (X \otimes Y) \xrightarrow{\cong} (Y \otimes X), \quad X, Y \in \text{Ob}_{\mathbf{C}}. \quad (15)$$

Explicitly, this means that for any  $f_1 : X_1 \rightarrow Y_1$  and  $f_2 : X_2 \rightarrow Y_2$ , the following diagram commutes:

$$\begin{array}{ccc} X_1 \otimes X_2 & \xrightarrow{f_1 \otimes f_2} & Y_1 \otimes Y_2 \\ \text{br}_{X_1, X_2} \downarrow & & \downarrow \text{br}_{Y_1, Y_2} \\ X_2 \otimes X_1 & \xrightarrow{f_2 \otimes f_1} & Y_2 \otimes Y_1 \end{array} \quad (16)$$

##### Conditions

1. *Hexagon identities*: Given any objects  $X, Y, Z \in \text{Ob}_{\mathbf{C}}$ , the following diagrams must commute.

$$\begin{array}{ccccc} (X \otimes Y) \otimes Z & \xrightarrow{\text{br}_{X,Y} \otimes \text{id}_Z} & (Y \otimes X) \otimes Z & \xrightarrow{\text{as}_{Y,X,Z}} & Y \otimes (X \otimes Z) \\ \text{as}_{X,Y,Z} \downarrow & & & & \downarrow \text{id}_Y \otimes \text{br}_{X,Z} \\ X \otimes (Y \otimes Z) & \xrightarrow{\text{br}_{X,Y \otimes Z}} & (Y \otimes Z) \otimes X & \xrightarrow{\text{as}_{Y,Z,X}} & Y \otimes (Z \otimes X) \end{array} \quad (17)$$

$$\begin{array}{ccccc} X \otimes (Y \otimes Z) & \xrightarrow{\text{id}_X \otimes \text{br}_{Y,Z}} & X \otimes (Z \otimes Y) & \xrightarrow{\text{as}_{Y,X,Z}^{-1}} & (X \otimes Z) \otimes Y \\ \text{as}_{X,Y,Z}^{-1} \downarrow & & & & \downarrow \text{br}_{X,Z} \otimes \text{id}_Y \\ (X \otimes Y) \otimes Z & \xrightarrow{\text{br}_{X \otimes Y, Z}} & Z \otimes (X \otimes Y) & \xrightarrow{\text{as}_{Z,X,Y}^{-1}} & (Z \otimes X) \otimes Y \end{array} \quad (18)$$

**Remark 34.8.** If  $\langle \mathbf{C}, \otimes, \mathbf{1}, \text{as}, \text{lu}, \text{ru}, \text{br} \rangle$  is a braided monoidal category, we can show that the following diagram commutes for all  $X \in \text{Ob}_{\mathbf{C}}$ .

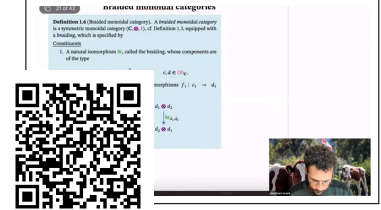
$$\begin{array}{ccc} \mathbf{1} \otimes X & \xrightarrow{\text{br}_{\mathbf{1}, X}} & X \otimes \mathbf{1} \\ \text{lu}_X \searrow & & \swarrow \text{ru}_X \\ & X & \end{array} \quad (19)$$

#### Definition 34.9 (Symmetric monoidal category)

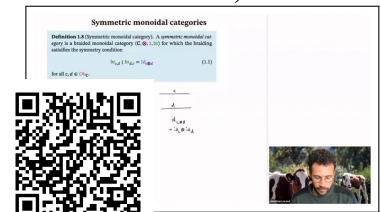
A *symmetric monoidal category* is a braided monoidal category  $\langle \mathbf{C}, \otimes, \mathbf{1}, \text{as}, \text{lu}, \text{ru}, \text{br} \rangle$  for which the braiding satisfies the symmetry condition

$$\text{br}_{X,Y} \circ \text{br}_{Y,X} = \text{id}_{X \otimes Y} \quad (20)$$

Watch: *Braided monoidal categories* (2 minutes).



Watch: *Symmetric monoidal categories* (5 minutes).



for all  $X, Y \in \text{Ob}_{\mathbf{C}}$ .

**Remark 34.10.** If  $\text{br}$  is a natural isomorphism such that it is a candidate to be a braiding on a given monoidal category, and if, additionally, it satisfies (20), then the two hexagon identities are equivalent, and so only one of them needs to be checked.

### DP is a symmetric monoidal category

We define a monoidal product  $\otimes$  for **DP** on objects by

$$P \otimes Q = P \times Q \quad (21)$$

and on morphisms by

$$\frac{\mathbf{d} : P^{\text{op}} \times R \rightarrow_{\text{Pos}} \mathbf{Bool} \quad \mathbf{e} : Q^{\text{op}} \times S \rightarrow_{\text{Pos}} \mathbf{Bool}}{\mathbf{d} \otimes \mathbf{e} : (P \times Q)^{\text{op}} \times (R \times S) \rightarrow_{\text{Pos}} \mathbf{Bool},} \quad (22)$$

$$\langle \langle a, c \rangle^*, \langle b, d \rangle \rangle \mapsto \mathbf{d}(a^*, b) \wedge \mathbf{e}(c^*, d),$$

**Lemma 34.11.** There is a symmetric monoidal category  $\langle \mathbf{DP}, \otimes, \mathbf{1}, \text{as}, \text{lu}, \text{ru}, \text{br} \rangle$  where the braiding is given by the design problem  $\text{br}_{P,Q} : P \times Q \rightarrow Q \times P$  with

$$\text{br}_{P,Q}(\langle p_1, q_1 \rangle^*, \langle q_2, p_2 \rangle) := (p_1 \leq_P p_2) \wedge (q_1 \leq_Q q_2) \quad (23)$$

for any  $P, Q \in \text{Ob}_{\mathbf{DP}}$ .

### 34.4. Monoidal functors

**Definition 34.12** (Strong monoidal functor)

Let  $\langle \mathbf{C}, \otimes_{\mathbf{C}}, \mathbf{1}_{\mathbf{C}} \rangle$  and  $\langle \mathbf{D}, \otimes_{\mathbf{D}}, \mathbf{1}_{\mathbf{D}} \rangle$  be two monoidal categories.

A *strong monoidal functor* between  $\mathbf{C}$  and  $\mathbf{D}$  is given by:

1. A functor

$$F : \mathbf{C} \rightarrow \mathbf{D}; \quad (24)$$

2. An isomorphism

$$\text{iso} : \mathbf{1}_{\mathbf{D}} \rightarrow F(\mathbf{1}_{\mathbf{C}}); \quad (25)$$

3. A natural isomorphism  $\mu$

$$\mu_{X,Y} : F(X) \otimes_{\mathbf{D}} F(Y) \rightarrow F(X \otimes_{\mathbf{C}} Y), \quad \forall X, Y \in \mathbf{C}, \quad (26)$$

satisfying the following conditions:

- (a) *Associativity*: For all objects  $X, Y, Z \in \mathbf{C}$ , there are *associators*  $\text{as}^{\mathbf{C}}$  and  $\text{as}^{\mathbf{D}}$  such that the diagram in Fig. 6a commutes.
- (b) *Unitality*: For all  $X \in \mathbf{C}$ , there exist left and right *unitors*  $\text{lu}^{\mathbf{C}}$  and  $\text{ru}^{\mathbf{C}}$ , the diagram in Fig. 6b commutes.

$$\begin{array}{ccc}
 (F(X) \otimes_D F(Y)) \otimes_D F(Z) & \xrightarrow{\text{as}_D^{F(X), F(Y), F(Z)}} & F(X) \otimes_D (F(Y) \otimes_D F(Z)) \\
 \downarrow \mu_{X,Y} \otimes_D \text{id}(F(Z)) & & \downarrow \text{id}(F(X)) \otimes_D \mu_{Y,Z} \\
 F(X \otimes_C Y) \otimes_D F(Z) & & F(X) \otimes_D F(Y \otimes_C Z) \\
 \downarrow \mu_{X \otimes_D Y, Z} & & \downarrow \mu_{X, Y \otimes_D Z} \\
 F((X \otimes_C Y) \otimes_C Z) & \xrightarrow{F(\text{as}_{X,Y,Z}^C)} & F(X \otimes_C (Y \otimes_C Z))
 \end{array}$$

(a) Natural associativity

$$\begin{array}{ccc}
 1_D \otimes_D F(X) & \xrightarrow{\text{iso} \otimes_D \text{id}_{F(X)}} & F(1_C) \otimes_D F(X) \\
 \downarrow \text{lu}_D & & \downarrow \mu_{1_C, X} \\
 F(X) & \xleftarrow{F(\text{lu}_X^C)} & F(1_C \otimes_C X) \\
 \\ 
 F(X) \otimes_D 1_D & \xrightarrow{\text{id}_{F(X)} \otimes_D \text{iso}} & F(X) \otimes_D F(1_C) \\
 \downarrow \text{ru}_D & & \downarrow \mu_{X, 1_C} \\
 F(X) & \xleftarrow{F(\text{ru}_X^C)} & F(X \otimes_C 1_C)
 \end{array}$$

(b) Natural unitality

**Figure 6.:** Commuting diagrams using in Def. 34.12

## 34.5. Dual objects and morphisms

There is a concept of “duality” for objects in monoidal category which we will introduce with an illustrative example.

We have seen in Example 34.5 that the category  $\mathbf{C} = \mathbf{Vect}_{\mathbb{R}}$  of real vector spaces is symmetric monoidal, with tensor product as the monoidal product. Given a vector space  $V$ , its *linear dual* is the real vector space

$$V^* := \{\text{linear maps } V \rightarrow \mathbb{R}\} = \text{Hom}_{\mathbf{C}}(V; \mathbb{R}). \quad (27)$$

Recall from linear algebra the following fact about any vector space  $V$ :

$$V \simeq (V^*)^* \text{ if and only if } \dim V < \infty. \quad (28)$$

One might say that the finite-dimensional real vector spaces are characterizable based on their behavior in this way with respect to the operation of taking the linear dual.

We will develop an alternative formulation of this fact, based on the notion of a *dualizable object*. This notion will make sense in the setting of any (symmetric) monoidal category, and we will see then, that (28) translates to the statement

$$V \in \text{Ob}_{\mathbf{Vect}_{\mathbb{R}}} \text{ is dualizable if and only if } \dim V < \infty. \quad (29)$$

Key protagonists in this reformulation are *evaluation* and *coevaluation* maps.

In the following,  $V$  denotes a *finite-dimensional* real vector space. The evaluation map  $\text{ev}_V$  associated to  $V$  is

$$\begin{aligned} \text{ev}_V : V^* \otimes V &\rightarrow \mathbb{R}, \\ \langle l, v \rangle &\mapsto l(v). \end{aligned} \quad (30)$$

In other words, given  $\langle l, v \rangle$ , the map  $\text{ev}_V$  evaluates  $l$  at  $v$ .

The coevaluation map  $\text{coev}_V$  associated to  $V$  is slightly trickier to describe. Let  $\{e_1, \dots, e_n\}$  be a basis of  $V$ , and let  $\{e_1^*, \dots, e_n^*\}$  be the corresponding dual basis of  $V^*$ . Then

$$\begin{aligned} \text{coev}_V : \mathbb{R} &\rightarrow V \otimes V^*, \\ \lambda &\mapsto \lambda \sum_{i=1}^n e_i \otimes e_i^*. \end{aligned} \quad (31)$$

It turns out that this map is independent of the choice of basis. One way to think of this coevaluation map is to recall that  $V \otimes V^* \simeq \text{Hom}(V, V)$ . Under this identification,  $\text{coev}_V$  maps the scalar  $\lambda$  to the linear endomorphism of  $V$  which is “multiplication by  $\lambda$ ”. (In terms of matrices, this is a diagonal matrix, with  $\lambda$  at every entry of the diagonal.)

Recall that as part of the monoidal structure on  $\mathbf{Vect}_{\mathbb{R}}$  we have the left and right unitors

$$\text{lu}_V : \mathbf{1} \otimes V \xrightarrow{\cong} V \quad V \in \text{Ob}_{\mathbf{Vect}_{\mathbb{R}}} \quad (32)$$

$$\text{ru}_V : V \otimes \mathbf{1} \xrightarrow{\cong} V \quad V \in \text{Ob}_{\mathbf{Vect}_{\mathbb{R}}}. \quad (33)$$

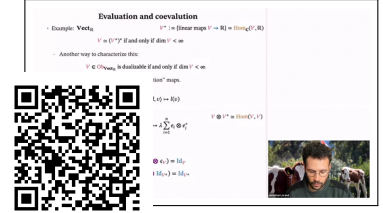
The evaluation and coevaluation maps defined above satisfy the following equations:

$$\text{lu}_V^{-1} \circ (\text{coev}_V \otimes \text{id}_V) \circ \text{as}_{V, V^*, V} \circ (\text{id}_V \otimes \text{ev}_V) \circ \text{ru}_V = \text{id}_V \quad (34)$$

and

$$\text{ru}_{V^*}^{-1} \circ (\text{id}_{V^*} \otimes \text{coev}_V) \circ \text{as}_{V^*, V, V^*}^{-1} \circ (\text{ev}_V \otimes \text{id}_{V^*}) \circ \text{lu}_{V^*} = \text{id}_{V^*}. \quad (35)$$

Watch: *Dualizability* (10 minutes).



**Graded exercise I.7** (VectSnakeEquations)

Check (34) and (35) by direct calculation, assuming that  $V$  is a finite-dimensional real vector space.

The equations (34) and (35) form the basis for the general notion of dualizability in a monoidal category.

**Definition 34.13** (Dual object)

Let  $\langle \mathbf{C}, \otimes_{\mathbf{C}}, \mathbf{1}_{\mathbf{C}} \rangle$  be a monoidal category, and let  $X \in \text{Ob}_{\mathbf{C}}$ . A *right dual object* of  $X$  is specified by:

Constituents

1. an object  $X^{\vee} \in \text{Ob}_{\mathbf{C}}$ ;
2. a morphism  $\text{ev}_X : X^{\vee} \otimes X \rightarrow \mathbf{1}$ , called *evaluation*;
3. a morphism  $\text{coev}_X : \mathbf{1} \rightarrow X \otimes X^{\vee}$ , called *coevaluation*;

Conditions

1.

$$\text{lu}_X^{-1} \circ (\text{coev}_X \otimes \text{id}_X) \circ \text{as}_{X, X^{\vee}, X} \circ (\text{id}_X \otimes \text{ev}_X) \circ \text{ru}_X = \text{id}_X; \quad (36)$$

2.

$$\text{ru}_{X^{\vee}}^{-1} \circ (\text{id}_{X^{\vee}} \otimes \text{coev}_X) \circ \text{as}_{X^{\vee}, X, X^{\vee}}^{-1} \circ (\text{ev}_X \otimes \text{id}_{X^{\vee}}) \circ \text{lu}_{X^{\vee}} = \text{id}_{X^{\vee}}. \quad (37)$$

**Definition 34.14**

An object  $X$  in a monoidal category is called *right dualizable* if there exists, in the category, a right dual object of  $X$ .

**Remark 34.15.** There is an analogous definition of *left dual object* and *left dualizability*. One can show that when the monoidal category in question is symmetric, then left dual objects can be seen as right dual objects, and vice versa. In this case, we then speak simply of *dualizability*.

**Definition 34.16**

[Compact closed category] A symmetric monoidal category is called *compact closed* if every object is dualizable.

**Definition 34.17** (Dual morphism)

Let  $f : X \rightarrow Y$  be a morphism in a monoidal category  $\langle \mathbf{C}, \otimes_{\mathbf{C}}, \mathbf{1}_{\mathbf{C}} \rangle$  and suppose that  $X$  and  $Y$  have right duals  $X^{\vee}$  and  $Y^{\vee}$ , respectively. The *right dual* of  $f$  is the morphism  $f^{\vee} : Y^{\vee} \rightarrow X^{\vee}$  given by the composition

$$\begin{aligned} Y^{\vee} &\xrightarrow{\text{ru}_Y^{-1}} Y^{\vee} \otimes \mathbf{1} \xrightarrow{\text{id}_{Y^{\vee}} \otimes \text{coev}_X} Y^{\vee} \otimes (X \otimes X^{\vee}) \\ &\xrightarrow{\text{as}^{-1}} (Y^{\vee} \otimes X) \otimes X^{\vee} \xrightarrow{(\text{id}_{Y^{\vee}} \otimes f) \otimes \text{id}_{X^{\vee}}} (Y^{\vee} \otimes Y) \otimes X^{\vee} \\ &\xrightarrow{\text{ev}_Y \otimes \text{id}_{X^{\vee}}} \mathbf{1} \otimes X^{\vee} \xrightarrow{\text{lu}_{X^{\vee}}} X^{\vee} \end{aligned} \quad (38)$$

## 34.6. Trace for monoidal categories

**Definition 34.18** (Trace of an endomorphism)

Let  $\langle \mathbf{C}, \otimes, \mathbf{1}, \text{as}, \text{lu}, \text{ru}, \text{br} \rangle$  be a symmetric monoidal category. Let  $X \in \text{Ob}_{\mathbf{C}}$  be dualizable and let

$$f : X \rightarrow X. \quad (39)$$

The *trace* of  $f$  is the morphism

$$\text{Tr}(f) : \mathbf{1} \rightarrow \mathbf{1} \quad (40)$$

defined by

$$\mathbf{1} \xrightarrow{\text{coev}_X} X \otimes X^\vee \xrightarrow{f \otimes \text{id}_{X^\vee}} X \otimes X^\vee \xrightarrow{\text{br}} X^\vee \otimes X \xrightarrow{\text{ev}_X} \mathbf{1} \quad (41)$$

**Graded exercise I.8** (LinearAlgebraTrace)

Let  $\mathbf{C}$  be the category of finite-dimensional real vector spaces and  $\mathbb{R}$ -linear maps. We have seen that this category is symmetric monoidal when equipped with the usual tensor product as monoidal product. Furthermore, in Section 34.5 we saw that every object in this category is dualizable.

Fix a finite-dimensional real vector space  $V$ , and let  $\{e_1, \dots, e_n\}$  be a basis of it. We saw that a choice of dual object for  $V$  is given by  $V^* = \text{Hom}(V, \mathbb{R})$ , together with the evaluation map

$$\begin{aligned} \text{ev}_V : V^* \otimes V &\rightarrow \mathbb{R}, \\ \langle l, v \rangle &\mapsto l(v). \end{aligned} \quad (42)$$

and the co-evaluation map

$$\begin{aligned} \text{coev}_V : \mathbb{R} &\rightarrow V \otimes V^*, \\ \lambda &\mapsto \lambda \sum_{i=1}^n e_i \otimes e_i^*. \end{aligned} \quad (43)$$

where  $\{e_1^*, \dots, e_n^*\}$  is the basis dual to the one we chose for  $V$ .

Let  $f : V \rightarrow V$  be a linear endomorphism – that is,  $f \in \text{Hom}_{\mathbf{C}}(V, V)$ . Compute the trace  $\text{Tr}(f) \in \text{Hom}_{\mathbf{C}}(\mathbb{R}, \mathbb{R})$  of  $f$  according to Def. 34.18, and explain why it is the linear map “multiplication by the trace of  $f$ ”, where “trace” in this latter phrase is the usual notion that we know from linear algebra.

**Graded exercise I.9** (DPSnakeTrace)

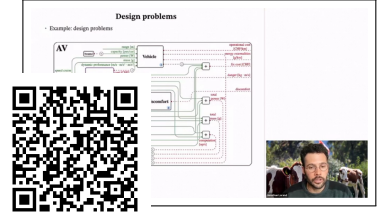
In this exercise we work with the category  $\mathbf{DP}$  of posets and design problems, equipped with the symmetric monoidal structure where the monoidal product is the cartesian product of posets. The braiding

$$\text{br}_{\mathbf{P}, \mathbf{Q}} : \mathbf{P} \times \mathbf{Q} \rightarrow \mathbf{Q} \times \mathbf{P}$$

is defined by

$$\text{br}_{\mathbf{P}, \mathbf{Q}}(\langle p_1, q_1 \rangle^*, \langle q_2, p_2 \rangle) := (p_1 \leq_{\mathbf{P}} p_2) \wedge (q_1 \leq_{\mathbf{Q}} q_2). \quad (44)$$

Watch: *Feedback* (8 minutes).





In the following you are free to use the identification

$$(\mathbf{P} \times \mathbf{Q})^{\text{op}} = \mathbf{P}^{\text{op}} \times \mathbf{Q}^{\text{op}} \quad (45)$$

for any posets  $\mathbf{P}, \mathbf{Q}$ . Also, recall that  $(\mathbf{P}^{\text{op}})^{\text{op}} = \mathbf{P}$ .

Let us define the following duality data:

$$\triangleright \mathbf{P}^{\vee} := \mathbf{P}^{\text{op}}$$

$$\triangleright \text{ev}_{\mathbf{P}} : (\mathbf{P}^{\text{op}} \times \mathbf{P})^{\text{op}} \times \{\bullet\} \rightarrow \mathbf{Bool}, \quad \langle \langle x^*, y \rangle^*, \bullet \rangle \mapsto y \leq_{\mathbf{P}} x$$

$$\triangleright \text{coev}_{\mathbf{P}} : \{\bullet\}^{\text{op}} \times (\mathbf{P} \times \mathbf{P}^{\text{op}}) \rightarrow \mathbf{Bool}, \quad \langle \bullet, \langle x, y^* \rangle \rangle \mapsto y \leq_{\mathbf{P}} x$$

Your tasks:

1. Guess the definitions of the associator **as** and the unitors **lu**, **ru** for the monoidal category  $\mathbf{DP}$ , check that each has the correct type, and justify why each of them does indeed define a morphism in the category  $\mathbf{DP}$ .
2. Guess the definitions of  $\text{lu}^{-1}$  and  $\text{ru}^{-1}$  and check for one of them that it does indeed define the inverse morphism.
3. Check that  $\text{ev}_{\mathbf{P}}$  and  $\text{coev}_{\mathbf{P}}$ , as defined above, are morphisms in  $\mathbf{DP}$ .
4. For an arbitrary poset  $\mathbf{P}$  and the duality data given above, prove that this snake equation

$$\text{lu}_{\mathbf{P}}^{-1} \circ (\text{coev}_{\mathbf{P}} \otimes \text{id}_{\mathbf{P}}) \circ \text{as}_{\mathbf{P}, \mathbf{P}^{\text{op}}, \mathbf{P}} \circ (\text{id}_{\mathbf{P}} \otimes \text{ev}_{\mathbf{P}}) \circ \text{ru}_{\mathbf{P}} = \text{id}_{\mathbf{P}} \quad (46)$$

holds.

**Definition 34.19** (Trace of a generalized endomorphism)

Let  $\langle \mathbf{C}, \otimes, \mathbf{1}, \text{as}, \text{lu}, \text{ru}, \text{br} \rangle$  be a symmetric monoidal category. Let  $X \in \text{Ob}_{\mathbf{C}}$  be dualizable and let

$$f : (Y \otimes X) \rightarrow (Z \otimes X). \quad (47)$$

The trace over  $X$  of  $f$  is the morphism

$$\text{Tr}_{Y,Z}^X(f) : Y \rightarrow Z \quad (48)$$

defined by

$$\begin{aligned} Y &\xrightarrow{\text{ru}_Y^{-1}} Y \otimes \mathbf{1} \xrightarrow{\text{id}_Y \otimes \text{coev}_X} Y \otimes (X \otimes X^{\vee}) \xrightarrow{\text{as}^{-1}} (Y \otimes X) \otimes X^{\vee} \xrightarrow{f \otimes \text{id}_{X^{\vee}}} (Z \otimes X) \otimes X^{\vee} \\ &\xrightarrow{\text{as}} Z \otimes (X \otimes X^{\vee}) \xrightarrow{\text{id}_Z \otimes \text{br}} Z \otimes (X^{\vee} \otimes X) \xrightarrow{\text{id}_Z \otimes \text{ev}_X} Z \otimes \mathbf{1} \xrightarrow{\text{ru}_Z} Z \end{aligned} \quad (49)$$

**Definition 34.20** (Traced monoidal category)

We say that a symmetric monoidal category  $\langle \mathbf{C}, \otimes, \mathbf{1}, \text{as}, \text{lu}, \text{ru}, \text{br} \rangle$  is *traced* if it is equipped with a family of functions

$$\text{Tr}_{X,Y}^Z : \text{Hom}_{\mathbf{C}}(X \otimes Z; Y \otimes Z) \rightarrow \text{Hom}_{\mathbf{C}}(X; Y), \quad (50)$$

satisfying the following axioms:

1. *Naturality in  $X$* : For any morphisms  $f : X \otimes Z \rightarrow Y \otimes Z$  and  $g : X' \rightarrow X$ ,

$$\text{Tr}_{X',Y}^Z((g \otimes \text{id}_Z) \circ f) = g \circ \text{Tr}_{X,Y}^Z(f) \quad (51)$$

2. *Naturality in  $Y$* : For any morphisms  $f : X \otimes Z \rightarrow Y \otimes Z$  and  $g : Y \rightarrow Y'$ ,

$$\text{Tr}_{X,Y'}^Z(f \circ (g \otimes \text{id}_Z)) = \text{Tr}_{X,Y}^Z(f) \circ g \quad (52)$$

3. *Dinaturality in  $Z$* : For any morphisms  $f : X \otimes Z \rightarrow Y \otimes Z'$  and  $g : Z' \rightarrow Z$ ,

$$\mathrm{Tr}_{X,Y}^Z(f \circ (\mathrm{id}_Y \otimes g)) = \mathrm{Tr}_{X,Y}^{Z'}((\mathrm{id}_X \otimes g) \circ f). \quad (53)$$

4. *Vanishing I*: For any morphisms  $f : X \rightarrow Y$  in  $\mathbf{C}$ ,

$$\mathrm{Tr}_{X,Y}^1(f) = \mathrm{ru}_X^{-1} \circ f \circ \mathrm{ru}_Y. \quad (54)$$

5. *Vanishing II*: For any morphism  $f : (X \otimes Z) \otimes U \rightarrow (Y \otimes Z) \otimes U$  in  $\mathbf{C}$ ,

$$\mathrm{Tr}_{X,Y}^Z(\mathrm{Tr}_{X \otimes Z, Y \otimes Z}^U(f)) = \mathrm{Tr}_{X,Y}^{Z \otimes U}(\mathrm{as}_{X,Z,U} \circ f \circ \mathrm{as}_{Y,Z,U}^{-1}). \quad (55)$$

6. *Superposing*: For any morphism  $f : X \otimes Z \rightarrow Y \otimes Z$  in  $\mathbf{C}$ ,

$$\mathrm{Tr}_{V \otimes X, V \otimes Y}^Z(\mathrm{as}_{V,X,Z} \circ \mathrm{id}_V \otimes f \circ \mathrm{as}_{V,Y,Z}^{-1}) = \mathrm{id}_V \otimes \mathrm{Tr}_{X,Y}^Z(f). \quad (56)$$

7. *Yanking*:

$$\mathrm{Tr}_{Z,Z}^Z(\mathrm{br}_{Z,Z}) = \mathrm{id}_Z. \quad (57)$$

**Remark 34.21.** Other variants of the definition of a traced monoidal category can be found in the literature. For instance, some include a more general version of the superposing law, see Lemma 34.22 below.

**Lemma 34.22.** Let  $\langle \mathbf{C}, \otimes, \mathbf{1}_{\mathbf{C}}, \mathrm{br}, \mathrm{Tr} \rangle$  be a traced monoidal category. Then a more general version of the superposing law holds: for any morphisms  $f : X \otimes Z \rightarrow Y \otimes Z$  and  $g : U \rightarrow V$ ,

$$\mathrm{Tr}_{U \otimes X, V \otimes Y}^Z(g \otimes f) = g \otimes \mathrm{Tr}_{X,Y}^Z(f). \quad (58)$$

Proof is missing.



### Graded exercise I.10 (TracingRelations)

Given the associative stacking category  $\langle \langle \mathbf{Rel} \rangle, \otimes, \boxtimes \rangle$ , as defined in Def. 30.14, consider the trace operation

$$\mathrm{Tr}_{X,Y}^Z : \mathrm{Hom}_{\langle \mathbf{Rel} \rangle}(X \circledast Z; Y \circledast Z) \rightarrow \mathrm{Hom}_{\langle \mathbf{Rel} \rangle}(X; Y)$$

which is defined, for a morphism  $R \in \mathrm{Hom}_{\mathbf{C}}(X \circledast Z; Y \circledast Z)$ , by

$$\mathrm{Tr}_{X,Y}^Z(R) = \{ \langle x, y \rangle \in X \times Y \mid \exists z \in Z : \langle x \circledast z, y \circledast z \rangle \in R \}. \quad (59)$$

Your task is to check that this definition satisfies the following two trace axioms:

1. *Vanishing II*:

For any relation  $R : X \circledast Z \circledast U \rightarrow Y \circledast Z \circledast U$  in  $\langle \mathbf{Rel} \rangle$ ,

$$\mathrm{Tr}_{X,Y}^{Z \circledast U}(R) = \mathrm{Tr}_{X,Y}^Z(\mathrm{Tr}_{X \circledast Z, Y \circledast Z}^U(R)). \quad (60)$$

2. *Superposing*:

For any relations  $R : X \bowtie Z \rightarrow Y \bowtie Z$  and  $S : V \rightarrow W$  in  $\langle \mathbf{Rel} \rangle$ ,

$$\mathrm{Tr}_{V \bowtie X, W \bowtie Y}^Z (S \otimes R) = S \otimes \mathrm{Tr}_{X, Y}^Z (R). \quad (61)$$

**Graded exercise I.11** (RelDualsTrace)

In this exercise we work with the category **Rel** of sets and relations, equipped with the symmetric monoidal structure where the monoidal product is the cartesian product of sets. The braiding is

$$\mathrm{br}_{A, B} : \{ \langle \langle x, y \rangle, \langle y', x' \rangle \rangle \in (A \times B) \times (B \times A) \mid x = x', y = y' \}. \quad (62)$$

This symmetric monoidal category is compact closed when we let the dual  $A^\vee$  of any set  $A$  be the set itself,  $A^\vee = A$ , and we define evaluation and co-evaluation by

$$\mathrm{ev}_A : A \times A \rightarrow 1, \quad \mathrm{ev}_A = \{ \langle \langle x, y \rangle, \bullet \rangle \in (A \times A) \times 1 \mid x = y \} \quad (63)$$

and

$$\mathrm{coev}_A : 1 \rightarrow A \times A, \quad \mathrm{coev}_A = \{ \langle \bullet, \langle x, y \rangle \rangle \in 1 \times (A \times A) \mid x = y \} \quad (64)$$

respectively.

Your tasks:

1. Let  $S : A \rightarrow B$  be a (generic) morphism in **Rel**. Compute the dual morphism  $S^\vee : B^\vee \rightarrow A^\vee$ .
2. Let  $R : A \times C \rightarrow B \times C$  be a morphism in **Rel**. Show that the trace of  $R$ , given by the composition

$$\mathrm{ru}_A^{-1} \circ (\mathrm{id}_A \otimes \mathrm{coev}_C) \circ \mathrm{as}_{A, C}^{-1} \circ (R \otimes \mathrm{id}_C) \circ \mathrm{as}_{B, C} \circ (\mathrm{id}_B \otimes \mathrm{br}_{A, A}) \circ (\mathrm{id}_B \otimes \mathrm{ev}_C) \circ \mathrm{ru}_B \quad (65)$$

is equal to the relation

$$\{ \langle x, y \rangle \in A \times B \mid \exists z \in C : \langle \langle x, z \rangle, \langle y, z \rangle \rangle \in R \}. \quad (66)$$

**Graded exercise I.12** (DPSnakeTracePart2)

In this exercise we work again with the category **DP** of posets and design problems, equipped with the symmetric monoidal structure where the monoidal product is the cartesian product of posets. In the following we make the identification

$$(P \times Q)^{\mathrm{op}} = P^{\mathrm{op}} \times Q^{\mathrm{op}} \quad (67)$$

for any posets  $P, Q$ . Also, recall that  $(P^{\mathrm{op}})^{\mathrm{op}} = P$ .

In components, the associator for **DP** is

$$\mathrm{as}_{P, Q, R} : ((P \times Q) \times R)^{\mathrm{op}} \times (P \times (Q \times R)) \rightarrow \mathbf{Bool} \quad (68)$$

with

$$\mathrm{as}_{P, Q, R}(\langle \langle p_1^*, q_1^* \rangle, r_1^* \rangle, \langle p_2, \langle q_2, r_2 \rangle \rangle) = p_1 \leq p_2 \wedge q_1 \leq q_2 \wedge r_1 \leq r_2, \quad (69)$$

and the left unitor is

$$\mathrm{lu}_P : (1 \times P)^{\mathrm{op}} \times P \rightarrow \mathbf{Bool} \quad (70)$$

with

$$\text{lu}_{\mathbf{P}}(\langle \langle \bullet^*, p_1^* \rangle, p_2 \rangle) = p_1 \leq p_2. \quad (71)$$

The right unitor is analogous.

The braiding

$$\text{br}_{\mathbf{P}, \mathbf{Q}} : \mathbf{P} \times \mathbf{Q} \rightarrow \mathbf{Q} \times \mathbf{P}$$

is

$$\text{br}_{\mathbf{P}, \mathbf{Q}}(\langle p_1^*, q_1^* \rangle, \langle q_2, p_2 \rangle) := (p_1 \leq p_2) \wedge (q_1 \leq q_2). \quad (72)$$

We define the following duality data, with respect to which  $\mathbf{DP}$  is compact closed:

$$\triangleright \mathbf{P}^\vee := \mathbf{P}^{\text{op}}$$

$$\triangleright \text{ev}_{\mathbf{P}} : (\mathbf{P}^{\text{op}} \times \mathbf{P})^{\text{op}} \times \{\bullet\} \rightarrow \mathbf{Bool}, \quad \langle \langle x^*, y \rangle^*, \bullet \rangle \mapsto y \leq_P x$$

$$\triangleright \text{coev}_{\mathbf{P}} : \{\bullet\}^{\text{op}} \times (\mathbf{P} \times \mathbf{P}^{\text{op}}) \rightarrow \mathbf{Bool}, \quad \langle \bullet, \langle x, y^* \rangle \rangle \mapsto y \leq_P x$$

Your task: given a morphism  $\mathbf{d} : \mathbf{P} \times \mathbf{R} \rightarrow \mathbf{Q} \times \mathbf{R}$  in  $\mathbf{DP}$ , show that the design problem  $\mathbf{P} \rightarrow \mathbf{Q}$  given by the following composition

$$\text{ru}_{\mathbf{P}}^{-1} \circ (\text{id}_{\mathbf{P}} \otimes \text{coev}_{\mathbf{R}}) \circ \text{as}_{\mathbf{P}, \mathbf{R}, \mathbf{R}^{\text{op}}}^{-1} \circ (\mathbf{d} \otimes \text{id}_{\mathbf{R}^{\text{op}}}) \circ \text{as}_{\mathbf{Q}, \mathbf{R}, \mathbf{R}^{\text{op}}} \circ (\text{id}_{\mathbf{Q}} \otimes \text{br}_{\mathbf{R}, \mathbf{R}^{\text{op}}}) \circ (\text{id}_{\mathbf{Q}} \otimes \text{ev}_{\mathbf{R}}) \circ \text{ru}_{\mathbf{Q}} \quad (73)$$

is equal to the design problem  $\mathbf{P} \rightarrow \mathbf{Q}$  given by

$$\begin{aligned} & \mathbf{P}^{\text{op}} \times \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{Bool}, \\ & \langle p^*, q \rangle \mapsto \bigvee_{r \in \mathbf{R}} \mathbf{d}(\langle p, r \rangle^*, \langle q, r \rangle). \end{aligned} \quad (74)$$



## Solutions to selected exercises

### Solution of Exercise 47.

Solution is missing.



### Solution of Exercise 48.

Solution is missing.



### Solution of Exercise 49.

Solution is missing.



### Solution of Exercise 50.

Solution is missing.













## 35. Sameness

35.1 Sameness in category theory . . . 518

35.2 Isomorphism is not identity . . . 521

### 35.1. Sameness in category theory

One nice thing about the category of sets is that we are all used to working with sets and functions. And many concepts that are familiar in the setting of sets and functions can actually be reformulated in a way which makes sense for lots of other categories, if not for all categories. It can be fun, and insightful, to see known definitions transformed into “category theory language”. For example: the notion of a bijective function is a familiar concept. There are at least two ways of saying what it means for a function  $f : \mathbf{A} \rightarrow \mathbf{B}$  of sets to be bijective:

**Definition 1:** “ $f : \mathbf{A} \rightarrow \mathbf{B}$  is bijective if, for every  $y \in \mathbf{B}$  there exists precisely one  $x \in \mathbf{A}$  such that  $f(x) = y$ ;

**Definition 2:** “ $f : \mathbf{A} \rightarrow \mathbf{B}$  is bijective if there exists a function  $g : \mathbf{B} \rightarrow \mathbf{A}$  such that  $f \circ g = \text{id}_{\mathbf{B}}$  and  $g \circ f = \text{id}_{\mathbf{A}}$ ”.

It is a short proof to show that the above two definitions are equivalent. The first definition, however, does not lend itself well to generalization in category theory, because it is formulated using something that is very specific to sets: namely, it refers to *elements* of the sets  $\mathbf{A}$  and  $\mathbf{B}$ . And we have seen that the objects of a category need not be sets, and so in general we cannot speak of “elements” in the usual sense. Definition 2, on the other hand, can easily be generalized to work in any category. To formulate this version, all we need are morphisms, their composition, the notion of identity morphisms, and the notion of equality of morphisms (for equations such as “ $f \circ g = \text{id}_x$ ”). The generalization we obtain is the fundamental notion of an “isomorphism”.

#### Definition 35.1 (Isomorphism)

Let  $\mathbf{C}$  be a category, let  $X, Y \in \text{Ob}_{\mathbf{C}}$  be objects, and let  $f : X \rightarrow Y$  be a morphism. We say that  $f$  is an *isomorphism* if there exists a morphism  $g : Y \rightarrow X$  such that  $f \circ g = \text{id}_X$  and  $g \circ f = \text{id}_Y$ .

**Remark 35.2.** The morphism  $g$  in the above definition is called the **inverse** of  $f$ . Because of the symmetry in how the definition is formulated, it is easy to see that  $g$  is necessarily also an isomorphism, and its inverse is  $f$ .

**Exercise 51.** In the previous remark we wrote *the* inverse. We do this because inverses are in fact unique. Can you prove this? That is, show that if  $f : X \rightarrow Y$  is an isomorphism, and if  $g_1 : Y \rightarrow X$  and  $g_2 : Y \rightarrow X$  are morphisms such that  $f \circ g_1 = \text{id}_X$  and  $g_1 \circ f = \text{id}_Y$ , and  $f \circ g_2 = \text{id}_X$  and  $g_2 \circ f = \text{id}_Y$ , then necessarily  $g_1 = g_2$ .

*See solution on page 523.*

#### Definition 35.3 (Isomorphic objects)

Let  $X, Y$  be two objects in a category. We say that  $X$  and  $Y$  are *isomorphic* if there exists an isomorphism  $X \rightarrow Y$  or  $Y \rightarrow X$ .

For the formulation of the definition of “isomorphic”, mathematicians might often only require the existence of an isomorphism  $X \rightarrow Y$ , say, since by remark above we know there is then necessarily also an isomorphism in the opposing direction, namely the inverse. We choose here the longer, perhaps more cumbersome formulation just to emphasize the symmetry of the term “isomorphic”. Also note that the definition leaves unspecified whether there might be just one or perhaps many isomorphisms  $X \rightarrow Y$ .

When two objects are isomorphic, in some contexts we will want to think of them as “the same”, and in some contexts we will want to keep track of more information. In fact, in category theory, it is typical to think in terms of different

kinds of “sameness”. To give a sense of this, we look at some examples using sets.

**Example 35.4** (Semantic coherence). Suppose Francesca and Gabriel want to share a dish at a restaurant. Francesca only speaks Italian, and Gabriel only speaks German. Let  $M$  denote the set of dishes on the menu. For each dish, Francesca can say if she is willing to eat it, or not. This can be modeled by a function  $f : M \rightarrow \{\text{Sì}, \text{No}\}$  which maps a given dish  $m \in M$  to the statement “Sì” (yes, I’d eat it) or “No” (no, I wouldn’t eat it). Gabriel can do similarly, and this can be modeled as a function  $g : M \rightarrow \{\text{Ja}, \text{Nein}\}$ . Then, the subset of dishes of  $M$  that both Francesca and Gabriel are willing to eat (and thus able to share) is

$$\{m \in M \mid f(m) = \text{Sì} \quad \text{and} \quad g(m) = \text{Ja}\}. \quad (1)$$

Suppose the server at the restaurant knows no Italian and no German. To help with the situation, he introduces a new two-element set:  $\{\heartsuit, \clubsuit\}$ . Then Francesca and Gabriel can each map their respective positive answers (“Sì” and “Ja”) to “ $\heartsuit$ ”, and their respective negative answers to “ $\clubsuit$ ”. This defines isomorphisms

$$\{\text{Sì}, \text{No}\} \longleftrightarrow \{\heartsuit, \clubsuit\} \longleftrightarrow \{\text{Ja}, \text{Nein}\} \quad (2)$$

whose compositions provide a translation between the Italian and German two-element sets. Using these isomorphisms, we obtain, by composition, new functions

$$\tilde{f} : M \longrightarrow \{\heartsuit, \clubsuit\}, \quad \tilde{g} : M \longrightarrow \{\heartsuit, \clubsuit\}, \quad (3)$$

and the set of dishes that Francesca and Gabriel would be willing to share can be written as

$$\{m \in M \mid \tilde{f}(m) = \heartsuit \quad \text{and} \quad \tilde{g}(m) = \heartsuit\}. \quad (4)$$

This may all seem unnecessarily complicated. The main point of this example is the following. There are infinitely many two-element sets; commonly used ones might be, for example

$$\{0, 1\}, \{\text{true}, \text{false}\}, \{\perp, \top\}, \{\text{left}, \text{right}\}, \{-, +\}, \text{etc.} \quad (5)$$

They are all isomorphic (for any two such sets, there are precisely two possible isomorphisms between them) and we can in principle use any one in place of another. However, in most cases, we should keep precise track of the semantics of what each of the two elements mean in a given context, such as how they are being used in interaction with other mathematical constructs.

**Example 35.5** (Relabelling). Suppose we want to buy an electric stepper motor for a robot that we are building, and for this we consult a catalogue of electric stepper motors\*.

The catalogue might be organized as a large table, where on the left-hand side there is a column listing all available motors (identified with a model ID), and the remaining columns correspond to different attributes that each of the models of motor might have, such as the name of the company that manufactures the motor, the size dimensions, the weight, the maximum power, the price, *etc.* A simple illustration is provided in Table 35.1.

Suppose that your old way of listing models of motors has become outdated, and you need to change to a new system, where each model is identified, say, by a unique numerical 10-digit code. Relabelling each of the models with its numerical code corresponds to an isomorphism, say relabel, from the new set  $N$  of numerical codes to the old set  $M$  of model names. In contrast to the previous example,

\* See [pololu.com](http://pololu.com) for a standard catalogue of electric stepper motors.

**Table 35.1.:** A simplified catalogue of motors.

Motor ID	Company	Size [mm <sup>3</sup> ]	Weight [g]	Max Power [W]	Cost [USD]
1204	SOYO	20 × 20 × 30	60.0	2.34	19.95
1206	SOYO	28 × 28 × 45	140.0	3.00	19.95
1207	SOYO	35 × 35 × 26	130.0	2.07	12.95
2267	SOYO	42 × 42 × 38	285.0	4.76	16.95
2279	Sanyo Denki	42 × 42 × 31.5	165.0	5.40	164.95
1478	SOYO	56.4 × 56.4 × 76	1,000	8.96	49.95
2299	Sanyo Denki	50 × 50 × 16	150.0	5.90	59.95

however, it is of course absolutely necessary to keep track of the isomorphism relabel that defines the relabelling. This is what holds the information of which code denotes which model.

Note also that all the other labelling functionalities in our example database may be updated by precomposing with relabel. For example, the old “Company” label was described by a function

$$\text{Company} : M \rightarrow C. \quad (6)$$

The updated version of the “Company” label, using the new set  $N$  of model IDs, is obtained by the composition

$$N \xrightarrow{\text{relabel}} M \xrightarrow{\text{Company}} C. \quad (7)$$

**Example 35.6.** Going back to currency exchangers, recall that any currency exchanger  $\langle a, d \rangle$ , given by

$$\begin{aligned} f_{\langle a, b \rangle} : \mathbb{R} &\rightarrow \mathbb{R}, \\ x &\mapsto ax - d, \end{aligned} \quad (8)$$

is an isomorphism, since we can define a currency exchanger  $\langle a', b' \rangle$  such that

$$\langle a, b \rangle \circ \langle a', b' \rangle = \langle a', b' \rangle \circ \langle a, b \rangle = \langle 1, 0 \rangle. \quad (9)$$

**Example 35.7.** In **FinSet**, isomorphisms from a set to itself are automorphisms, and correspond to *permutations* of the set. Assuming a cardinality of  $n$  for the set (for instance, the set has  $n$  elements), the number of isomorphisms is given by the number of ways in which we can “rearrange”  $n$  elements of the set, which is  $n!$ .

**Example 35.8.** In **Set**, isomorphisms between  $\mathbb{R} \rightarrow \mathbb{R}$  correspond to invertible functions.

## 35.2. Isomorphism is not identity

**Example 35.9.** Consider the two currencies ₿ and satoshi. These are both objects of the category **Curr** and are isomorphic. Being isomorphic does not mean to be strictly “the same”. Indeed, even if the amounts correspond, 1 ₿ and 1,000,000 satoshi are different elements of different sets, but there exists an isomorphism between the two. For one direction, the isomorphism transforms ₿ into satoshi (multiplying the real number by 1,000,000); the other direction transforms satoshi into ₿ (dividing the real number by 1,000,000).

### Invertible functions are isomorphisms

**Definition 35.10** (Strict monotone functions)

A function  $f : \mathbb{R} \rightarrow \mathbb{R}$  is *strictly monotone* if for all  $a, b \in \mathbb{R}$ :

$$\frac{a < b}{f(a) < f(b)} . \quad (10)$$

**Exercise 52.** Show that strictly monotone maps.  $f : \mathbb{R} \rightarrow \mathbb{R}$  are invertible.

*See solution on page 523.*



## Solutions to selected exercises

### Solution of Exercise 51.

Solution is missing.



**Solution of Exercise 52.** Because of Def. 35.10, given  $a, b \in \mathbb{R}$ , if  $f(a) = f(b)$ , then we must have  $a = b$ . Suppose that  $f$  is strictly monotone, but there exist elements  $a, b \in \mathbb{R}$  such that  $f(a) = f(b)$  but  $a \neq b$ . In other words, we have  $f(a) - f(b) = 0$  but  $a - b \neq 0$ , which implies

$$(f(a) - f(b))(a - b) = 0. \quad (11)$$

However, from the definition of monotonicity, one should have

$$(f(a) - f(b))(a - b) > 0. \quad (12)$$

Therefore, we have a contradiction, implying that if  $f$  is strictly monotone, then  $f(a) = f(b)$  implies  $a = b$ .







36. DP queries as functors	527
37. Solving finite co-design problems	547
38. Monads	573

The *St. Bernard* is a pretty large working dog from the Western Alps in Italy and Switzerland. Originally, such dogs were bred for rescue work, by the hospice of the Great St. Bernard Pass on Italian-Swiss border.







## 36. DP queries as functors

In this chapter we show how the two types of queries **FixFunMinRes** and **FixResMaxFun** can be seen as *functors* from the category **DP** to two new categories to be defined that represent the types of “solutions”. The specification of these functors represents a complete solution for **DP** optimization at the “mathematical level”, without taking into account questions of computability or resource consumption, which will be explored in the successive chapters.

36.1 Queries are functors from problem statements to solutions . .	528
36.2 The $\text{Pos}_U$ and $\text{Pos}_L$ categories . .	530
36.3 Queries as functors . . . . .	542

### 36.1. Queries are functors from problem statements to solutions

In this and the following chapters we are going to build towards the solution of co-design problems. We will consider an arbitrary graph of design problems, in which nodes are design problems and edges are arbitrary interconnections between functionality and resources, obtained through the operations of a traced monoidal category (series, parallel, feedback) plus the lattice structure (and, or) of design problems. On this structure we want to solve the query **FixFunMinRes** (Section 25.3) or, symmetrically, **FixResMaxFun** (Section 25.3)

We look at this from a compositional point of view. We will assume that we know the solution to **FixFunMinRes** for each of the components. We think of the components as primitive blocks, because they are given in a catalogue format as a DPI, or they are special cases (+,  $\otimes$ , etc.) which we will solve as special cases. Given the solution for the primitive blocks, we want to know what is the solution for **FixFunMinRes** for the entire diagram.

What is the form of the solution that we expect? Given a DP  $\mathbf{d} : \mathbf{F} \leftrightarrow \mathbf{R}$  we expect the solution to **FixFunMinRes** to be a function that, given a fixed functionality  $f \in \mathbf{F}$ , returns the minimal resources, which form an upper set. We call this function  $H_{\mathbf{d}}$ .

#### Definition 36.1

Given a DP  $\mathbf{d} : \mathbf{F} \leftrightarrow \mathbf{R}$  we denote by  $H_{\mathbf{d}} : \mathbf{F} \rightarrow_{\text{Pos}} \mathbf{UR}$  the map that associates to each functionality  $f$  the set of minimal resources sufficient to realize  $f$ :

$$\begin{aligned} H_{\mathbf{d}} : \mathbf{F} &\rightarrow_{\text{Pos}} \mathbf{UR}, \\ f &\mapsto \{r \in \mathbf{R} : \mathbf{d}(f^*, r)\}. \end{aligned} \quad (1)$$

If a certain functionality  $f$  is infeasible, then  $H(f) = \emptyset$ .

**Remark 36.2** (Monotonicity). Consider a DP  $\mathbf{d} : \mathbf{F} \leftrightarrow \mathbf{R}$  and  $f \leq f'$ . We know

$$\begin{aligned} H_{\mathbf{d}}(f) &= \{r \in \mathbf{R} : \mathbf{d}(f^*, r)\} \\ &\supseteq \{r \in \mathbf{R} : \mathbf{d}(f'^*, r)\} \\ &= H_{\mathbf{d}}(f'), \end{aligned} \quad (2)$$

showing monotonicity.

Symmetrically, the solution to **FixResMaxFun** is given by a function that we call  $K_{\mathbf{d}}$ .

#### Definition 36.3

Given a DP  $\langle \mathbf{F}, \mathbf{R}, \mathbf{I}, \text{prov}, \text{req} \rangle$ , define the map  $K_{\mathbf{d}} : \mathbf{R} \rightarrow_{\text{Pos}} \mathbf{LF}$  that associates to each resource  $r$  the set of functionalities which can be realized with  $r$ :

$$\begin{aligned} K_{\mathbf{d}} : \mathbf{R} &\rightarrow_{\text{Pos}} \mathbf{LF}, \\ r &\mapsto \{f \in \mathbf{F} : \mathbf{d}(f^*, r)\}. \end{aligned} \quad (3)$$

If a certain resource  $r$  only leads to infeasible functionalities, then  $K(r) = \emptyset$ .

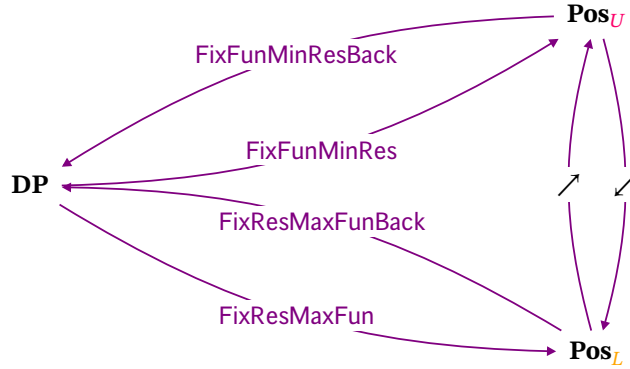
**Remark 36.4** (Monotonicity). Consider a DP  $\mathbf{d} : \mathbf{F} \leftrightarrow \mathbf{R}$  and  $r \leq r'$ . We know

$$\begin{aligned} K_{\mathbf{d}}(r) &= \{f \in \mathbf{F} : \mathbf{d}(f^*, r)\} \\ &\subseteq \{f \in \mathbf{F} : \mathbf{d}(f^*, r')\} \\ &= K_{\mathbf{d}}(r'), \end{aligned} \quad (4)$$

showing monotonicity.

A question that arises naturally is whether the map  $H_d$  is sufficient to reconstruct the original DP. The answer is yes. We will prove that  $H_d$  defines a morphism in a category called  $\mathbf{Pos}_U$ , and that this category is equivalent (Def. 33.19) to  $\mathbf{DP}$ , therefore being traced monoidal, with a lattice structure. In fact,  $\mathbf{FixFunMinRes}$  can be seen as a functor from  $\mathbf{DP}$  to  $\mathbf{Pos}_U$ . Symmetrically,  $K_d$  is a morphism in a category  $\mathbf{Pos}_L$  equivalent to  $\mathbf{DP}$  and  $\mathbf{FixResMaxFun}$  can be seen as the functor from  $\mathbf{DP}$  to  $\mathbf{Pos}_L$ .

This situation is represented in Fig. 1.



**Figure 1.:** In this chapter, we show that the queries  $\mathbf{FixResMaxFun}$  and  $\mathbf{FixFunMinRes}$  can be seen as functors from  $\mathbf{DP}$  to two new categories,  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$ . We show that  $\mathbf{DP}$  is equivalent to these categories: a  $\mathbf{DP}$  is univocally defined by the answers to the two queries.

In the course of this chapter, by defining the two functors  $\mathbf{FixFunMinRes}$  and  $\mathbf{FixResMaxFun}$ , we effectively have solved the problem of optimization for DPs in the “mathematical” way. However, this is only the first step, because it does not say anything about whether the functor is actually computable. In the next chapter (Chapter 37) we will look at finite approximations of DPs and the computational complexity of the solution. Then, we will introduce the theory of monads, and based on that, we will be able to show how to construct bounded finite approximations of any DPs.

## 36.2. The $\mathbf{Pos}_U$ and $\mathbf{Pos}_L$ categories

### Definition 36.5 (Category $\mathbf{Pos}_U$ )

The category  $\mathbf{Pos}_U$  consists of:

1. *Objects*: objects are posets;
2. *Morphisms*: given objects  $X, Y \in \mathbf{Ob}_{\mathbf{Pos}_U}$ , morphisms from  $f : X \rightarrow Y$  are monotone maps of the form  $f^\star : X \rightarrow_{\mathbf{Pos}} UY$ .
3. *Composition of morphisms*: Given morphisms  $f : X \rightarrow Y, g : Y \rightarrow Z$ , their composition  $f \circ g : X \rightarrow Z$  is given by

$$(f \circ g)^\star : X \rightarrow_{\mathbf{Pos}} UZ$$

$$x \mapsto \bigcup_{y \in f^\star(x)} g^\star(y); \quad (5)$$

4. *Identity morphism*: given an object  $X \in \mathbf{Ob}_{\mathbf{Pos}_U}$ , the identity morphism  $\text{id}_X : X \rightarrow X$  is given by the application of the upper closure operator:

$$\text{id}_X^\star(x) := \uparrow\{x\}. \quad (6)$$

**Remark 36.6.** Note that the composition of morphisms in this category corresponds to the generalization of the series operator for boolean profunctors.

Analogously, we can define the  $\mathbf{Pos}_L$  category.

### Definition 36.7 (Category $\mathbf{Pos}_L$ )

The category  $\mathbf{Pos}_L$  consists of:

1. *Objects*: objects are posets;
2. *Morphisms*: given objects  $X, Y \in \mathbf{Ob}_{\mathbf{Pos}_L}$ , morphisms  $f : X \rightarrow Y$  are monotone maps of the form  $f^\star : X \rightarrow_{\mathbf{Pos}} LY$ .
3. *Composition of morphisms*: Given morphisms  $f : X \rightarrow Y, g : Y \rightarrow Z$ , their composition  $f \circ g : X \rightarrow Z$  is given by

$$(f \circ g)^\star : X \rightarrow_{\mathbf{Pos}} LZ$$

$$x \mapsto \bigcup_{y \in f^\star(x)} g^\star(y); \quad (7)$$

4. *Identity morphism*: given an object  $X \in \mathbf{Ob}_{\mathbf{Pos}_L}$ , the identity morphism  $\text{id}_X : X \rightarrow X$  is given by the application of the lower closure operator:

$$\text{id}_X^\star(x) := \downarrow\{x\}. \quad (8)$$

We now show that  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$  are indeed categories.

**Lemma 36.8.**  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$  are categories.

*Proof.* We prove that  $\mathbf{Pos}_U$  is a category. The proof for  $\mathbf{Pos}_L$  is analogous. In the following, we show unitality and associativity.

**Unitality:** Given  $f : X \rightarrow Y$ , we have:

$$(f \circ \text{id}_Y)^\star(x) = \bigcup_{y \in f^\star(x)} \text{id}_Y^\star(y)$$

$$= \bigcup_{y \in f^\star(x)} \uparrow\{y\}$$

$$= \bigcup_{y \in f^\star(x)} \{y' \in Y : y \leq_Y y'\}. \quad (9)$$



We know that  $f^\star(x)$  is an upper set:

$$\begin{aligned} f^\star(x) &= \bigcup_{y \in f^\star(x)} \{y\} \\ &= \bigcup_{y \in f^\star(x)} \{y' \in Y : y \leq_Y y'\}. \end{aligned} \quad (10)$$

Therefore,  $(f \circ \text{id}_Y)^\star(x) = f^\star(x)$  for all  $x \in X$ . Similarly, we have:

$$\begin{aligned} (\text{id}_X \circ f)^\star(x) &= \bigcup_{x' \in \text{id}_X^\star(x)} f^\star(x') \\ &= \bigcup_{x' \in \uparrow\{x\}} f^\star(x') \\ &= f^\star(x), \end{aligned} \quad (11)$$

where the last equality holds since  $f^\star$  is a monotone function and  $f^\star(x') \subseteq f^\star(x)$  for all  $x' \in \uparrow\{x\}$ .

**Associativity:** Consider three morphisms  $f : X \rightarrow Y$ ,  $g : Y \rightarrow Z$ , and  $h : Z \rightarrow U$ . We have:

$$\begin{aligned} ((f \circ g) \circ h)^\star(x) &= \bigcup_{z \in \left(\bigcup_{y \in f^\star(x)} g^\star(y)\right)} h^\star(z) \\ &= \bigcup_{y \in f^\star(x)} \bigcup_{z \in g^\star(y)} h^\star(z) \\ &= (f \circ (g \circ h))^\star(x). \end{aligned} \quad (12)$$

Therefore,  $\mathbf{Pos}_U$  is a category.  $\square$

We can show that  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$  are equivalent categories (Def. 33.19).

**Lemma 36.9.**  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$  are isomorphic: there exists a pair of functors

$$\begin{aligned} \swarrow &: \mathbf{Pos}_U \rightarrow \mathbf{Pos}_L, \\ \nearrow &: \mathbf{Pos}_L \rightarrow \mathbf{Pos}_U, \end{aligned} \quad (13)$$

such that  $\swarrow \circ \nearrow = \text{id}_{\mathbf{Pos}_U}$  and  $\nearrow \circ \swarrow = \text{id}_{\mathbf{Pos}_L}$ , where  $\text{id}_{\mathbf{Pos}_U}$  and  $\text{id}_{\mathbf{Pos}_L}$  are the identity functors on  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$ , respectively.

*Proof.* To prove this, we need to define the needed functors and to show that they satisfy the listed properties. We choose the functors to be the ones that map a poset  $\mathbf{P}$  in a category to its opposite version  $\mathbf{P}^{\text{op}}$  in another category. Given a morphism  $f : X \rightarrow Y$  in  $\mathbf{Pos}_U$ , we have:

$$\begin{aligned} (\swarrow(f))^\star &: X^{\text{op}} \rightarrow_{\mathbf{Pos}} L Y^{\text{op}} \\ x &\mapsto f^\star(x). \end{aligned} \quad (14)$$

Given a morphism  $g : X \rightarrow Y$  in  $\mathbf{Pos}_L$ , we have:

$$\begin{aligned} (\nearrow(g))^\star &: X^{\text{op}} \rightarrow_{\mathbf{Pos}} U Y^{\text{op}} \\ x &\mapsto g^\star(x). \end{aligned} \quad (15)$$

$\swarrow$  and  $\nearrow$  are functors:



▷ *Preservation of identities:* Given  $X \in \mathbf{Ob}_{\mathbf{Pos}_U}$ , we have:

$$\begin{aligned} (\swarrow (\text{id}_X))^* &= \uparrow_X \{x\} \\ &= \downarrow_{X^{\text{op}}} \{x\} \\ &= \text{id}_{X^{\text{op}}}^*, \end{aligned} \tag{16}$$

where  $\text{id}_X$  is an identity morphism in  $\mathbf{Pos}_U$ , and  $\text{id}_{X^{\text{op}}}$  is an identity morphism in  $\mathbf{Pos}_L$ . Similarly, given  $X \in \mathbf{Ob}_{\mathbf{Pos}_L}$  we have:

$$\begin{aligned} (\nearrow (\text{id}_X))^* &= \downarrow_X \{x\} \\ &= \uparrow_{X^{\text{op}}} \{x\} \\ &= \text{id}_{X^{\text{op}}}^*. \end{aligned} \tag{17}$$

▷ *Preservation of composition:* This can be easily seen as follows. Given any  $f \in \mathbf{Hom}_{\mathbf{Pos}_U}(X; Y)$ ,  $g \in \mathbf{Hom}_{\mathbf{Pos}_U}(Y; Z)$ :

$$\begin{aligned} (\swarrow (f \circ g))^* &= (f \circ g)^* \\ &= (\swarrow (f)) \circ (\swarrow (g))^*. \end{aligned} \tag{18}$$

Similarly, given any  $f \in \mathbf{Hom}_{\mathbf{Pos}_L}(X; Y)$ ,  $g \in \mathbf{Hom}_{\mathbf{Pos}_L}(Y; Z)$ :

$$\begin{aligned} (\nearrow (f \circ g))^* &= (f \circ g)^* \\ &= (\nearrow (f)) \circ (\nearrow (g))^*. \end{aligned} \tag{19}$$

**Compositions return identity functors:** We want to show that by composing the two functors we obtain the identity functors in  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$ , respectively. Clearly, composing the two functors returns the identity on the objects, since for any poset  $\mathbf{P}$ , we have  $(\mathbf{P}^{\text{op}})^{\text{op}} = \mathbf{P}$ . The functors act on morphisms by “flipping the context”, and “flipping” twice is the “same” as not flipping.  $\square$

We can show that  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$  are monoidal categories.

**Lemma 36.10.**  $\mathbf{Pos}_U$  is a monoidal category with the following additional structure:

1. *Tensor product*  $\otimes$ : On objects, the tensor product corresponds to the product of posets. Given two morphisms  $f : X \rightarrow Y$  and  $g : Z \rightarrow U$ , we have  $f \otimes g : X \times Z \rightarrow Y \times U$ , with

$$\begin{aligned} (f \otimes g)^* &: X \times Z \rightarrow_{\mathbf{Pos}} U(Y \times U) \\ \langle x, z \rangle &\mapsto f^*(x) \times g^*(z). \end{aligned} \tag{20}$$

Note that the cartesian product of upper sets is an upper set.

2. *Unit:* The unit is the identity poset: the poset with a singleton carrier set and only the identity relation. We denote this by  $\mathbf{1}$ .
3. *Left unitor:* The left unitor is given by the pair of morphisms  $\text{lu}_X : \{\bullet\} \times X \rightarrow X$  and  $\text{lu}_X^{-1} : X \rightarrow \{\bullet\} \times X$ , with

$$\begin{aligned} \text{lu}_X^* &: \{\bullet\} \times X \rightarrow_{\mathbf{Pos}} UX \\ \langle \bullet, x \rangle &\mapsto \uparrow \{x\}, \end{aligned} \tag{21}$$

and

$$\begin{aligned} \text{lu}_X^{-1\star} : X &\rightarrow_{\mathbf{Pos}} U(\{\bullet\} \times X) \\ x &\mapsto \{\bullet\} \times \uparrow\{x\}, \end{aligned} \quad (22)$$

respectively.

4. *Right unitor*: The right unitor is given by the pair of morphisms  $\text{ru}_X : X \times \{\bullet\} \rightarrow X$  and  $\text{ru}_X^{-1} : X \rightarrow X \times \{\bullet\}$ , with

$$\begin{aligned} \text{ru}_X^{\star} : X \times \{\bullet\} &\rightarrow_{\mathbf{Pos}} UX \\ \langle x, \bullet \rangle &\mapsto \uparrow\{x\}, \end{aligned} \quad (23)$$

and

$$\begin{aligned} \text{ru}_X^{-1\star} : X &\rightarrow_{\mathbf{Pos}} U(X \times \{\bullet\}) \\ x &\mapsto \uparrow\{x\} \times \{\bullet\}, \end{aligned} \quad (24)$$

respectively.

5. *Associator*: The associator is given by the pair of morphisms  $\text{as}_{XY,Z} : (X \times Y) \times Z \rightarrow X \times (Y \times Z)$  and  $\text{as}_{X,YZ} : X \times (Y \times Z) \rightarrow (X \times Y) \times Z$ , given by

$$\begin{aligned} \text{as}_{XY,Z}^{\star} : (X \times Y) \times Z &\rightarrow_{\mathbf{Pos}} UX \times (UY \times UZ) \\ \langle \langle x, y \rangle, z \rangle &\mapsto \uparrow\{x\} \times (\uparrow\{y\} \times \uparrow\{z\}), \end{aligned} \quad (25)$$

and

$$\begin{aligned} \text{as}_{X,YZ}^{\star} : X \times (Y \times Z) &\rightarrow_{\mathbf{Pos}} (UX \times UY) \times UZ \\ \langle x, \langle y, z \rangle \rangle &\mapsto (\uparrow\{x\} \times \uparrow\{y\}) \times \uparrow\{z\}. \end{aligned} \quad (26)$$

We now want to show that  $\mathbf{Pos}_U$  can be equipped to become a symmetric monoidal category. To do so, we first need the following two facts.

**Lemma 36.11.** Given posets  $\mathbf{P}, \mathbf{Q}$ , a monotone maps  $f : \mathbf{P} \rightarrow \mathbf{Q}$ , and a family of singleton sets  $\{S_i\}_{i \in I}$ , with  $S_i = \{s_i\}$ ,  $s_i \in \mathbf{P}$ , the following equality holds:

$$\uparrow\left(\bigcup_{p \in \uparrow \bigcup_{i \in I} S_i} \{f(p)\}\right) = \uparrow\left(\bigcup_{i \in I} \{f(s_i)\}\right). \quad (27)$$

*Proof.* We first want to show that:

$$\underbrace{\uparrow\left(\bigcup_{p \in \uparrow \bigcup_{i \in I} S_i} \{f(p)\}\right)}_{\star} \subseteq \underbrace{\uparrow\left(\bigcup_{i \in I} \{f(s_i)\}\right)}_{\diamond}. \quad (28)$$

Take a

$$q \in \uparrow\left(\bigcup_{p \in \uparrow \bigcup_{i \in I} S_i} \{f(p)\}\right). \quad (29)$$

If we have such a  $q$ , it means that there exists a

$$q' \in \bigcup_{p \in \uparrow \bigcup_{i \in I} S_i} \{f(p)\} \quad (30)$$

such that  $q' \leq_Q q$ , and hence there is a  $p' \in \uparrow \bigcup_{i \in I} S_i$  such that  $q' = f(p')$ . Consequently, there must exist an  $i' \in I$  such that  $s_{i'} \leq_P p'$ . The monotonicity of  $f$  implies:

$$f(s_{i'}) \leq_P f(p') = q' \leq_Q q. \quad (31)$$

We know that  $s_{i'} \in \diamond$  and any  $q^* \in \mathbf{Q}$  satisfying  $f(s_{i'}) \leq_Q q^*$  belongs to  $\uparrow \diamond$ .

Therefore,  $\star \subseteq \uparrow \diamond$ , which proves the validity of (28).

We now want to show that:

$$\uparrow \left( \bigcup_{p \in \uparrow \bigcup_{i \in I} S_i} \{f(p)\} \right) \supseteq \uparrow \left( \bigcup_{i \in I} \{f(s_i)\} \right). \quad (32)$$

By now taking a

$$q \in \uparrow \left( \bigcup_{i \in I} \{f(s_i)\} \right), \quad (33)$$

we know that there is an  $i' \in I$  such that  $f(s_{i'}) \leq_Q q$ . Furthermore, we know that  $f(s_{i'}) \in \diamond$ . Therefore, any  $q^* \leq_Q f(s_{i'})$  must be in  $\uparrow \diamond$ , meaning that  $q \in \star$ , and proving the validity of (32).

The validity of (28) and (32) implies (27).  $\square$

**Remark 36.12.** Given posets  $\mathbf{P}, \mathbf{Q}$  and a monotone maps  $f : \mathbf{P} \rightarrow \mathbf{Q}$ , we have:

$$\uparrow \left( \bigcup_{p' \in \uparrow \{p\}} \{f(p')\} \right) = \uparrow \{f(p)\}. \quad (34)$$

This follows from Lemma 36.11, by considering a family of singleton sets consisting solely of the set  $\{p\}$ .

**Lemma 36.13.** The cartesian product of upper sets is an upper set. The cartesian product of lower sets is a lower set.

*Proof.* Consider two posets  $\mathbf{P}, \mathbf{Q}$  and two respective upper sets  $\mathbf{A}, \mathbf{B}$ . We have

$$\frac{a \in \mathbf{A} \quad a \leq_P a'}{a' \in \mathbf{A}}, \quad (35)$$

and

$$\frac{b \in \mathbf{B} \quad b \leq_Q b'}{b' \in \mathbf{B}}. \quad (36)$$

Therefore:

$$\frac{\langle a, b \rangle \in \mathbf{A} \times \mathbf{B} \quad \langle a, b \rangle \leq_{\mathbf{P} \times \mathbf{Q}} \langle a', b' \rangle}{\langle a', b' \rangle \in \mathbf{A} \times \mathbf{B}}, \quad (37)$$

which proves that  $\mathbf{A} \times \mathbf{B}$  is an upper set. The proof for the product of lower sets is analogous.  $\square$

**Lemma 36.14.**  $\langle \mathbf{Pos}_U, \otimes, \mathbf{1} \rangle$  from Lemma 36.10 equipped with the braiding isomorphism  $\text{br}_{X,Y} : X \times Y \xrightarrow{\cong} Y \times X$ , given by

$$\begin{aligned} \text{br}_{X,Y}^* : X \times Y &\rightarrow_{\mathbf{Pos}} U(Y \times X) \\ \langle x, y \rangle &\mapsto \uparrow \{y\} \times \uparrow \{x\}, \end{aligned} \quad (38)$$

defined for all  $X, Y \in \mathbf{Ob}_{\mathbf{Pos}_U}$ , forms a symmetric monoidal category.

*Proof.* We first show that the braiding defines an isomorphism. In other words, we want to show

$$(\text{br}_{X,Y} \circ \text{br}_{Y,X})^* = \text{id}_{X \times Y}^*. \quad (39)$$

We have

$$\begin{aligned}
 & (\text{br}_{X,Y} \circledast \text{br}_{Y,X})^*(x, y) \\
 &= \bigcup_{\langle x', y' \rangle \in \text{br}_{X,Y}^*(x, y)} \text{br}_{Y,X}^*(x', y') \\
 &= \bigcup_{\langle y', x' \rangle \in \uparrow\{y\} \times \uparrow\{x\}} \uparrow\{x'\} \times \uparrow\{y'\} \\
 &= \uparrow\{x\} \times \uparrow\{y\} \\
 &= \text{id}_{X \times Y}^*(x, y).
 \end{aligned} \tag{40}$$

Note that this comes from the fact that  $\text{br}$  is an involution. We now show naturality. Consider  $f : X \rightarrow Y$ ,  $g : Z \rightarrow U$ . We have

$$\begin{aligned}
 & ((f \otimes g) \circledast \text{br}_{Y,U})^*(x, z) \\
 &= \langle f^*(x), g^*(z) \rangle \circledast \text{br}_{Y,U} \\
 &= \left\langle \bigcup_{z' \in g^*(z)} \uparrow z', \bigcup_{x' \in f^*(x)} \uparrow x' \right\rangle.
 \end{aligned} \tag{41}$$

On the other hand:

$$\begin{aligned}
 & (\text{br}_{U,Y} \circledast (f \otimes g))^*(x, z) \\
 &= \langle \uparrow\{z\}, \uparrow\{x\} \rangle \circledast (f \otimes g)^* \\
 &= \left\langle \bigcup_{z' \in \uparrow\{z\}} g^*(z'), \bigcup_{x' \in \uparrow\{x\}} f^*(x') \right\rangle.
 \end{aligned} \tag{42}$$

Clearly, from Lemma 36.11 and Remark 36.12 we know that (41) and (42) are equivalent, proving naturality. We now just need to show hexagon identities. First, we want to show that

$$(\text{br}_{X,Y} \otimes \text{id}_Z) \circledast \text{as}_{Y,X,Z} \circledast (\text{id}_Y \otimes \text{br}_{X,Z}) = \text{as}_{X,Y,Z} \circledast \text{br}_{X,Y \otimes Z} \circledast \text{as}_{Y,Z,X} \tag{43}$$

To do so, we first look at the left-hand side of (43). We have

$$\begin{aligned}
 & ((\text{br}_{X,Y} \otimes \text{id}_Z) \circledast \text{as}_{Y,X,Z})^*(\langle x, y \rangle, z) \\
 &= \bigcup_{\langle y', x' \rangle, z' \in (\text{br}_{X,Y} \otimes \text{id}_Z)^*(\langle x, y \rangle, z)} \text{as}_{Y,X,Z}^*(y', x', z') \\
 &= \bigcup_{\langle y', x' \rangle, z' \in (\uparrow\{y\} \times \uparrow\{x\}) \times \uparrow\{z\}} \uparrow\{y'\} \times (\uparrow\{x'\} \times \uparrow\{z'\}) \\
 &= \uparrow\{y\} \times (\uparrow\{x\} \times \uparrow\{z\}).
 \end{aligned} \tag{44}$$

Furthermore, we have

$$\begin{aligned}
 & ((\text{br}_{X,Y} \otimes \text{id}_Z) \circledast \text{as}_{Y,X,Z} \circledast (\text{id}_Y \otimes \text{br}_{X,Z}))^*(\langle x, y \rangle, z) \\
 &= \bigcup_{\langle y', \langle x', z' \rangle \rangle \in \uparrow\{y\} \times (\uparrow\{x\} \times \uparrow\{z\})} (\text{id}_Y \otimes \text{br}_{X,Z})^*(y', \langle x', z' \rangle) \\
 &= \bigcup_{\langle y', \langle x', z' \rangle \rangle \in \uparrow\{y\} \times (\uparrow\{x\} \times \uparrow\{z\})} \uparrow\{y'\} \times (\uparrow\{z'\} \times \uparrow\{x'\}) \\
 &= \uparrow\{y\} \times (\uparrow\{z\} \times \uparrow\{x\}).
 \end{aligned} \tag{45}$$

We now look at the right-hand side of (43). We have

$$\begin{aligned}
 & \text{as}_{X,Y,Z} \circ \text{br}_{X,Y \otimes Z}^* (\langle x, y \rangle, z) \\
 &= \bigcup_{\langle x', \langle y', z' \rangle \rangle \in \text{as}_{X,Y,Z}^* (\langle x, y \rangle, z)} \text{br}_{X,Y \otimes Z}^* (x', \langle y', z' \rangle) \\
 &= \bigcup_{\langle x', \langle y', z' \rangle \rangle \in \uparrow\{x\} \times (\uparrow\{y\} \times \uparrow\{z\})} (\uparrow\{y'\} \times \uparrow\{z'\}) \times \uparrow\{x'\} \\
 &= (\uparrow\{y\} \times \uparrow\{z\}) \times \uparrow\{x\}.
 \end{aligned} \tag{46}$$

Furthermore, we have

$$\begin{aligned}
 & (\text{as}_{X,Y,Z} \circ \text{br}_{X,Y \otimes Z} \circ \text{as}_{Y,Z,X})^* (\langle x, y \rangle, z) \\
 &= \bigcup_{\langle \langle y', z' \rangle, x' \rangle \in (\uparrow\{y\} \times \uparrow\{z\}) \times \uparrow\{x\}} \text{as}_{Y,Z,X}^* (\langle y', z' \rangle, x') \\
 &= \bigcup_{\langle \langle y', z' \rangle, x' \rangle \in (\uparrow\{y\} \times \uparrow\{z\}) \times \uparrow\{x\}} \uparrow\{y'\} \times (\uparrow\{z'\} \times \uparrow\{x'\}) \\
 &= \uparrow\{y\} \times (\uparrow\{z\} \times \uparrow\{x\}).
 \end{aligned} \tag{47}$$

Clearly, since (45) and (47) are equal, the first hexagon identity is checked. The second hexagon identity can be checked analogously.  $\square$

**Definition 36.15** (Trace in  $\mathbf{Pos}_U$ )

Given a morphism  $f : X \times Z \rightarrow Y \times Z$  in  $\mathbf{Pos}_U$ , its trace in is defined as a morphism  $\text{Tr}_{X,Y}^Z(f) : X \rightarrow Y$ , given by

$$\begin{aligned}
 \text{Tr}_{X,Y}^Z(f)^* : X &\rightarrow UY \\
 x &\mapsto \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in f^*(x, z)\}.
 \end{aligned} \tag{48}$$

**Lemma 36.16.**  $\langle \mathbf{Pos}_U, \otimes, \mathbf{1}, \text{br} \rangle$  equipped with the trace operation defined in Def. 36.15 is a traced monoidal category.

*Proof.* We have already checked that  $\langle \mathbf{Pos}_U, \otimes, \mathbf{1}, \text{br} \rangle$  forms a symmetric monoidal category. First, we check that the trace indeed returns a valid morphism in  $\mathbf{Pos}_U$ . Given any  $X, Y, Z \in \mathbf{Ob}_{\mathbf{Pos}_U}$  and  $f : X \times Z \rightarrow Y \times Z$ , and any  $x \leq x' \in X$ , we need to prove that

$$\begin{aligned}
 & \text{Tr}_{X,Y}^Z(f)(x) \leq_{\mathbf{Pos}_U} \text{Tr}_{X,Y}^Z(f)(x') \\
 & \text{Tr}_{X,Y}^Z(f)^*(x) \supseteq \text{Tr}_{X,Y}^Z(f)^*(x')
 \end{aligned} \tag{49}$$

We know that  $f^*$  is a monotone map, meaning that

$$\begin{aligned}
 & \langle y, z \rangle \in f^*(x', z) \\
 & \hline
 & \langle y, z \rangle \in f^*(x, z)
 \end{aligned} \tag{50}$$

Therefore:

$$\begin{aligned}
 & y \in \text{Tr}_{X,Y}^Z(f)^*(x') \\
 & \hline
 & y \in \text{Tr}_{X,Y}^Z(f)^*(x)
 \end{aligned} \tag{51}$$

proving that  $\text{Tr}_{X,Y}^Z(f)^*$  is a monotone function. Furthermore, due to the

monotonicity of  $f^\star$ , for any  $y \leq y' \in Y$ ,  $x \in X$ ,  $z \in Z$ , we have:

$$\frac{\langle y, z \rangle \in f^\star(x, z)}{\langle y', z \rangle \in f^\star(x, z)} \quad (52)$$

proving that  $\text{Tr}_{X,Y}^Z(f)^\star(x)$  is an upper set for all  $x \in X$ . We now check the trace axioms one by one.

**Naturality I:** Given any object  $X, X', Y, Z \in \mathbf{Ob}_{\mathbf{Pos}_U}$ , a morphism  $f : X \times Z \rightarrow Y \times Z$ , and a morphism  $g : X' \rightarrow X$ , we have:

$$\begin{aligned} & \text{Tr}_{X',Y}^Z((g \otimes \text{id}_Z) \circ f)^\star(x') \\ &= \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in ((g \otimes \text{id}_Z) \circ f)^\star(x', z)\} \\ &= \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in \bigcup_{\langle x, z' \rangle \in g^\star(x') \times \uparrow\{z\}} f^\star(x, z')\} \\ &= \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in \bigcup_{x \in g^\star(x')} f^\star(x, z)\}. \end{aligned} \quad (53)$$

On the other hand, we have

$$\begin{aligned} (g \circ \text{Tr}_{X,Z}^Z(f))^\star(x') &= \bigcup_{x \in g^\star(x')} \text{Tr}_{X,Z}^Z(f)^\star(x) \\ &= \bigcup_{x \in g^\star(x')} \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in f^\star(x, z)\} \\ &= \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in \bigcup_{x \in g^\star(x')} f^\star(x, z)\}. \end{aligned} \quad (54)$$

Clearly (53) and (54) are equivalent, proving the first naturality condition.

**Naturality II:** Given any  $X, Y, Y', Z \in \mathbf{Ob}_{\mathbf{Pos}_U}$ ,  $f : X \times Z \rightarrow Y \times Z$ , and  $g : Y \rightarrow Y'$ , we have:

$$\begin{aligned} & \text{Tr}_{X,Y'}^Z(f \circ (g \otimes \text{id}_Z))^\star(x) \\ &= \{y' \in Y' \mid \bigvee_{z \in Z} \langle y', z \rangle \in (f \circ (g \otimes \text{id}_Z))^\star(x, z)\} \\ &= \{y' \in Y' \mid \bigvee_{z \in Z} \langle y', z \rangle \in \bigcup_{\langle y, z \rangle \in f^\star(x, z)} g^\star(y) \times \uparrow\{z\}\} \end{aligned} \quad (55)$$

On the other hand

$$(\text{Tr}_{X,Y}^Z(f) \circ g)^\star(x) = \bigcup_{y \in \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in f^\star(x, z)\}} g^\star(y) \quad (56)$$

**Vanishing:** Given any  $X, Y \in \mathbf{Ob}_{\mathbf{Pos}_U}$  and  $f : X \rightarrow Y$  in  $\mathbf{Pos}_U$ , we have

$$\begin{aligned} & \text{Tr}_{X,Y}^1(f)^\star(x) \\ &= \{y \in Y \mid \langle y, \bullet \rangle \in (f \otimes \text{id}_1)^\star(x, \bullet)\} \\ &= f^\star(x). \end{aligned} \quad (57)$$

Furthermore, given any  $X, Y, Z, U \in \mathbf{Ob}_{\mathbf{Pos}_U}$  and  $f : X \times Z \times U \rightarrow Y \times Z \times U$ , we have

$$\begin{aligned} & \text{Tr}_{X,Y}^{Z \times U}(f)^\star(x) \\ &= \{y \in Y \mid \bigvee_{\langle z, u \rangle \in Z \times U} \langle y, z, u \rangle \in f^\star(x, z, u)\} \end{aligned} \quad (58)$$

To check the second vanishing axiom, we also write:

$$\begin{aligned} & \text{Tr}_{X \times Z, Y \times Z}^U(f)^\star(x, z) \\ &= \{ \langle y, z \rangle \in Y \times Z \mid \bigvee_{u \in U} \langle y, z, u \rangle \in f^\star(x, z, u) \}. \end{aligned} \quad (59)$$

Therefore, we can write:

$$\begin{aligned} & \left( \text{Tr}_{X, Y}^Z \left( \text{Tr}_{X \times Z, Y \times Z}^U(f) \right) \right)^\star(x) \\ &= \{ y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in \text{Tr}_{X \times Z, Y \times Z}^U(f)^\star(x, z) \} \\ &= \{ y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in \{ \langle y', z' \rangle \in Y \times Z \mid \bigvee_{u \in U} \langle y', z', u \rangle \in f^\star(x, z', u) \} \} \\ &= \{ y \in Y \mid \bigvee_{z \in Z} \left( \bigvee_{u \in U} \langle y, z, u \rangle \in f^\star(x, z, u) \right) \} \\ &= \{ y \in Y \mid \bigvee_{\langle z, u \rangle \in Z \times U} \langle y, z, u \rangle \in f^\star(x, z, u) \}. \end{aligned} \quad (60)$$

Clearly, (58) and (60) are equivalent, proving the second vanishing axiom.

**Superposing:** Given any  $X, Y, Z \in \text{Ob}_{\text{Pos}_U}$  and  $f : X \times Z \rightarrow Y \times Z$ , we have:

$$\begin{aligned} & \text{Tr}_{U \times X, U \times Y}^Z(\text{id}_U \otimes f)^\star(u, x) \\ &= \{ \langle u, y \rangle \in U \times Y \mid \bigvee_{z \in Z} \langle u, y, z \rangle \in (\text{id}_U \otimes f)^\star(u, x, z) \} \\ &= \{ \langle u, y \rangle \in U \times Y \mid \bigvee_{z \in Z} (u \in \text{id}_U^\star(u)) \wedge (\langle y, z \rangle \in f^\star(x, z)) \} \\ &= \{ \langle u, y \rangle \in U \times Y \mid \bigvee_{z \in Z} (u \in \uparrow\{u\}) \wedge (\langle y, z \rangle \in f^\star(x, z)) \} \\ &= \{ \langle u, y \rangle \in \uparrow\{u\} \times Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in f^\star(x, z) \} \\ &= \uparrow\{u\} \times \{ y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in f^\star(x, z) \} \end{aligned} \quad (61)$$

On the other hand, we have:

$$(\text{id}_U \otimes \text{Tr}_{X, Y}^Z(f))^\star(u, x) = \uparrow\{u\} \times \{ y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in f^\star(x, z) \}. \quad (62)$$

Clearly, (61) and (62) are equivalent, proving the superposing axiom.

**Yanking:** Consider  $X \in \text{Ob}_{\text{Pos}_U}$ . We have

$$\begin{aligned} & \text{Tr}_{X, X}^X(\text{br}_{X, X})^\star(x) \\ &= \{ x' \in X \mid \bigvee_{x'' \in X} \langle x', x'' \rangle \in \text{br}_{X, X}^\star(x, x'') \} \\ &= \{ x' \in X \mid \bigvee_{x'' \in X} \langle x', x'' \rangle \in \uparrow\{x''\} \times \uparrow\{x\} \} \\ &= \{ x' \in X \mid \bigvee_{x'' \in X} (x' \in \uparrow\{x''\}) \wedge (x'' \in \uparrow\{x\}) \} \\ &= \{ x' \in X \mid x' \in \uparrow\{x\} \} \\ &= \uparrow\{x\} \\ &= \text{id}_X^\star(x), \end{aligned} \quad (63)$$

proving the yanking axiom.  $\square$

**Definition 36.17** (Order on morphisms in  $\mathbf{Pos}_U$ )

Given any two morphisms  $f, g : X \rightarrow Y$  in  $\mathbf{Pos}_U$ , we define an order between them as

$$\frac{f \leq_{\mathbf{Pos}_U} g}{f^*(x) \leq_{UY} g^*(x), \quad \forall x \in X} . \quad (64)$$

**Definition 36.18** (Order on morphisms in  $\mathbf{Pos}_L$ )

Given any two morphisms  $f, g : X \rightarrow Y$  in  $\mathbf{Pos}_L$ , we define an order between them as

$$\frac{f \leq_{\mathbf{Pos}_L} g}{f^*(x) \leq_{LY} g^*(x), \quad \forall x \in X} . \quad (65)$$

**Definition 36.19** (Intersection of morphisms in  $\mathbf{Pos}_L$ )

Given two morphisms  $f, g : X \rightarrow Y$  in  $\mathbf{Pos}_L$ , their *intersection* (meet) is a morphism  $f \wedge g : X \rightarrow Y$ , given by

$$\begin{aligned} (f \wedge g)^* : X &\rightarrow LY \\ x &\mapsto f^*(x) \cap g^*(x). \end{aligned} \quad (66)$$

**Definition 36.20** (Union of morphisms in  $\mathbf{Pos}_U$ )

Given two morphisms  $f, g : X \rightarrow Y$  in  $\mathbf{Pos}_U$ , their *union* (join) is a morphism  $f \vee g : X \rightarrow Y$ , given by

$$\begin{aligned} (f \vee g)^* : X &\rightarrow UY \\ x &\mapsto f^*(x) \cup g^*(x). \end{aligned} \quad (67)$$

**Definition 36.21** (Union of morphisms in  $\mathbf{Pos}_L$ )

Given two morphisms  $f, g : X \rightarrow Y$  in  $\mathbf{Pos}_L$ , their *union* (join) is a morphism  $f \vee g : X \rightarrow Y$ , given by

$$\begin{aligned} (f \vee g)^* : X &\rightarrow LY \\ x &\mapsto f^*(x) \cup g^*(x). \end{aligned} \quad (68)$$

**Lemma 36.22.** Given any  $X, Y \in \mathbf{Ob}_{\mathbf{Pos}_U}$ ,  $\mathbf{Hom}_{\mathbf{Pos}_U}(X; Y)$  is a bounded lattice with union  $\vee$  of morphisms in  $\mathbf{Pos}_U$  as join, intersection  $\wedge$  of morphisms in  $\mathbf{Pos}_U$  as meet, least upper bound  $\top_{\mathbf{Hom}_{\mathbf{Pos}_U}(X; Y)} : X \rightarrow Y$  given by

$$\begin{aligned} \top_{\mathbf{Hom}_{\mathbf{Pos}_U}(X; Y)}^* : X &\rightarrow UY \\ x &\mapsto \emptyset, \end{aligned} \quad (69)$$

and greatest lower bound  $\perp_{\mathbf{Hom}_{\mathbf{Pos}_U}(X; Y)} : X \rightarrow Y$  given by

$$\begin{aligned} \perp_{\mathbf{Hom}_{\mathbf{Pos}_U}(X; Y)}^* : X &\rightarrow UY \\ x &\mapsto Y. \end{aligned} \quad (70)$$



*Proof.* First, we need to prove that  $\text{Hom}_{\text{Pos}_U}(X; Y)$  forms a poset. To prove this, we check the following, using the order defined in Def. 36.17

▷ *Reflexivity:* Given  $f \in \text{Hom}_{\text{Pos}_U}(X; Y)$ , we can write

$$f^\star(x) \supseteq f^\star(x), \quad \forall x \in X, \quad (71)$$

which implies  $f \leq_{\text{Pos}_U} f$ .

▷ *Antisymmetry:* Consider

$$f, g \in \text{Hom}_{\text{Pos}_U}(X; Y) \quad (72)$$

with  $f \leq_{\text{Pos}_U} g$  and  $g \leq_{\text{Pos}_U} f$ . We know

$$(f \leq_{\text{Pos}_U} g) \Rightarrow f^\star(x) \supseteq g^\star(x), \quad \forall x \in X, \quad (73)$$

but also

$$(g \leq_{\text{Pos}_U} f) \Rightarrow g^\star(x) \supseteq f^\star(x), \quad \forall x \in X, \quad (74)$$

implying  $f = g$ .

▷ *Transitivity:* Consider

$$f, g, h \in \text{Hom}_{\text{Pos}_U}(X; Y) \quad (75)$$

with  $f \leq_{\text{Pos}_U} g$  and  $g \leq_{\text{Pos}_U} h$ . We have, for all  $x \in X$ ,

$$\begin{aligned} (f^\star(x) \supseteq g^\star(x)) \wedge (g^\star(x) \supseteq h^\star(x)) &\Rightarrow f^\star(x) \supseteq h^\star(x) \\ &\Rightarrow f \leq_{\text{Pos}_U} h. \end{aligned} \quad (76)$$

Consider now  $f, g \in \text{Hom}_{\text{Pos}_U}(X; Y)$ . Their least upper bound (join) is  $f \wedge g$ , since it is the least morphism such that  $f \leq_{\text{Pos}_U} (f \wedge g)$  and  $g \leq_{\text{Pos}_U} (f \wedge g)$ . Their greatest lower bound (meet) is  $f \vee g$ , since it is the greatest morphism such that  $(f \vee g) \leq_{\text{Pos}_U} f$  and  $(f \vee g) \leq_{\text{Pos}_U} g$ . Furthermore, for any  $f \in \text{Hom}_{\text{Pos}_U}(X; Y)$ , one will have, for all  $x \in X$

$$f^\star(x) \supseteq \emptyset = \top_{\text{Hom}_{\text{Pos}_U}(X; Y)}^\star(x), \quad (77)$$

implying that for all  $f \in \text{Hom}_{\text{Pos}_U}(X; Y)$  we have  $f \leq_{\text{Pos}_U} \top_{\text{Hom}_{\text{Pos}_U}(X; Y)}$ . Finally, for any  $f \in \text{Hom}_{\text{Pos}_U}(X; Y)$ , one will have, for all  $x \in X$

$$\perp_{\text{Hom}_{\text{Pos}_U}(X; Y)}^\star(x) = Y \supseteq f^\star(x) \quad (78)$$

implying that for all  $f \in \text{Hom}_{\text{Pos}_U}(X; Y)$  we have  $\perp_{\text{Hom}_{\text{Pos}_U}(X; Y)} \leq_{\text{Pos}_U} f$ .  $\square$

**Lemma 36.23.** Given any  $X, Y \in \text{Ob}_{\text{Pos}_U}$ ,  $\text{Hom}_{\text{Pos}_L}(X; Y)$  is a bounded lattice with intersection  $\wedge$  of morphisms in  $\text{Pos}_L$  as meet, union  $\vee$  of morphisms in  $\text{Pos}_L$  as join, least upper bound  $\top_{\text{Hom}_{\text{Pos}_L}(X; Y)} : X \rightarrow Y$  given by

$$\begin{aligned} \top_{\text{Hom}_{\text{Pos}_L}(X; Y)}^\star &: X \rightarrow LY \\ x &\mapsto Y, \end{aligned} \quad (79)$$

and greatest lower bound  $\perp_{\mathbf{Hom}_{\mathbf{Pos}_U}(X;Y)} : X \rightarrow Y$  given by

$$\begin{aligned} \perp_{\mathbf{Hom}_{\mathbf{Pos}_L}(X;Y)}^{\star} : X &\rightarrow LY \\ x &\mapsto \emptyset. \end{aligned} \tag{80}$$

*Proof.* The proof is analogous to the one of Lemma 36.22. Note that meets/joins and top/bottom are switched in meaning, because of the difference in order between  $UX$  and  $LX$ .  $\square$

### 36.3. DP queries are functors from problem statements to solutions

**Lemma 36.24.** There is a functor

$$\text{FixFunMinRes} : \mathbf{DP} \rightarrow \mathbf{Pos}_U \quad (81)$$

that maps:

1. An object (poset) in  $\mathbf{DP}$  to the same object (poset) in  $\mathbf{Pos}_U$ .
2. A morphism  $\mathbf{e} \in \text{Hom}_{\mathbf{DP}}(\mathbf{F}; \mathbf{R})$  to the morphism  $H_{\mathbf{e}} \in \text{Hom}_{\mathbf{Pos}_U}(\mathbf{F}; \mathbf{R})$ , where:

$$\begin{aligned} H_{\mathbf{e}}^* : \mathbf{F} &\rightarrow \text{Pos } U\mathbf{R} \\ f &\mapsto \{r \in \mathbf{R} \mid \mathbf{e}(f^*, r)\}. \end{aligned} \quad (82)$$

*Proof.* We prove the two conditions.

**Preservation of identities:** We have

$$\begin{aligned} \text{FixFunMinRes}(\text{id}_X^{\mathbf{DP}})^*(x) &= \{y \in Y \mid \text{id}_X^{\mathbf{DP}}(x^*, y)\} \\ &= \{y \in Y \mid x \leq y\} \\ &= \uparrow\{x\} \\ &= \text{id}_X^{\text{Pos}_U}(x). \end{aligned} \quad (83)$$

**Preservation of composition:** On one hand, we have

$$\begin{aligned} \text{FixFunMinRes}(\mathbf{d} \circ_{\mathbf{DP}} \mathbf{e})^*(x) &= \{z \in Z \mid (\mathbf{d} \circ_{\mathbf{DP}} \mathbf{e})(x^*, z)\} \\ &= \{z \in Z \mid \bigvee_{y \in Y} \mathbf{d}(x^*, y) \wedge \mathbf{e}(y^*, z)\}. \end{aligned} \quad (84)$$

On the other hand:

$$\begin{aligned} &(\text{FixFunMinRes}(\mathbf{d}) \circ_{\text{Pos}_U} \text{FixFunMinRes}(\mathbf{e}))^*(x) \\ &= \bigcup_{y \in \text{FixFunMinRes}(\mathbf{d})^*(x)} \text{FixFunMinRes}(\mathbf{e})^*(y) \\ &= \bigcup_{y \in \{y \in Y \mid \mathbf{d}(x^*, y)\}} \{z \in Z \mid \mathbf{e}(y^*, z)\} \\ &= \{z \in Z \mid (y \in Y) \wedge \mathbf{d}(x^*, y) \wedge \mathbf{e}(y^*, z)\} \\ &= \{z \in Z \mid \bigvee_{y \in Y} \mathbf{d}(x^*, y) \wedge \mathbf{e}(y^*, z)\}. \end{aligned} \quad (85)$$

Clearly, (84) and (85) coincide.  $\square$

**Lemma 36.25.** There is a functor

$$\text{FixResMaxFun} : \mathbf{DP} \rightarrow \mathbf{Pos}_L \quad (86)$$

which maps:

1. An object (poset) of  $\mathbf{DP}$  to the same object (poset) in  $\mathbf{Pos}_L$ .
2. A morphism  $\mathbf{e} \in \text{Hom}_{\mathbf{DP}}(\mathbf{F}; \mathbf{R})$  to the morphism  $K_{\mathbf{e}} \in \text{Hom}_{\mathbf{Pos}_L}(\mathbf{R}; \mathbf{F})$ , where:

$$\begin{aligned} K_{\mathbf{e}}^* : \mathbf{R} &\rightarrow \text{Pos } L\mathbf{F} \\ r &\mapsto \{f \in \mathbf{F} \mid \mathbf{e}(f^*, r)\}. \end{aligned} \quad (87)$$

*Proof.* The proof is analogous to the one of Lemma 36.24.  $\square$

**Lemma 36.26.** There is a functor  $\text{FixFunMinResBack} : \mathbf{Pos}_U \rightarrow \mathbf{DP}$  which maps:

1. An object (poset) in  $\mathbf{Pos}_U$  to the same object (poset) in  $\mathbf{DP}$ .
2. A morphism  $g \in \text{Hom}_{\mathbf{Pos}_U}(\mathbf{F}; \mathbf{R})$  to the morphism  $\mathbf{d}_g \in \text{Hom}_{\mathbf{DP}}(\mathbf{F}; \mathbf{R})$ , where:

$$\begin{aligned} \mathbf{d}_g : \mathbf{F}^{\text{op}} \times \mathbf{R} &\rightarrow_{\mathbf{Pos}} \mathbf{Bool} \\ \langle f^*, r \rangle &\mapsto r \in g^*(f). \end{aligned} \quad (88)$$

*Proof.* We prove the two conditions.

**Preservation of identities:** We have

$$\begin{aligned} \text{FixFunMinResBack}(\text{id}_X^{\text{Pos}_U})(x^*, y) &= y \in \text{id}_Y^{\text{Pos}_U}{}^* \\ &= y \in \uparrow\{x\} \\ &= \text{id}_X^{\mathbf{DP}}(x^*, y). \end{aligned} \quad (89)$$

**Preservation of composition:** On one hand, we have

$$\begin{aligned} \text{FixFunMinResBack}(f \circ_{\mathbf{Pos}_U} g)(x^*, z) &= z \in (f \circ_{\mathbf{Pos}_U} g)^*(x) \\ &= z \in \bigcup_{y \in f^*(x)} g^*(y). \end{aligned} \quad (90)$$

On the other hand:

$$\begin{aligned} (\text{FixFunMinResBack}(f) \circ_{\mathbf{DP}} \text{FixFunMinResBack}(g))(x^*, z) &= \bigvee_{y \in Y} (y \in f^*(x)) \wedge (z \in g^*(y)) \\ &= z \in \bigcup_{y \in f^*(x)} g^*(y). \end{aligned} \quad (91)$$

Clearly, (90) and (91) coincide.  $\square$

**Lemma 36.27.** There is a functor  $\text{FixResMaxFunBack} : \mathbf{Pos}_L \rightarrow \mathbf{DP}$  which maps:

1. An object (poset) in  $\mathbf{Pos}_L$  to the same object (poset) in  $\mathbf{DP}$ .
2. A morphism  $g \in \text{Hom}_{\mathbf{Pos}_L}(\mathbf{F}; \mathbf{R})$  to the morphism  $\mathbf{d}_g \in \text{Hom}_{\mathbf{DP}}(\mathbf{F}; \mathbf{R})$ , where:

$$\begin{aligned} \mathbf{d}_g : \mathbf{F}^{\text{op}} \times \mathbf{R} &\rightarrow_{\mathbf{Pos}} \mathbf{Bool} \\ \langle f^*, r \rangle &\mapsto f \in g^*(r). \end{aligned} \quad (92)$$

*Proof.* The proof is analogous to the one of Lemma 36.26.  $\square$

**Lemma 36.28.** The pair of functors  $\text{FixFunMinRes}$  and  $\text{FixFunMinResBack}$  together with the natural isomorphisms

$$\text{FixFunMinRes} \circ \text{FixFunMinResBack} \cong \text{id}_{\mathbf{DP}}, \quad (93)$$

and

$$\text{FixFunMinResBack} \circ \text{FixFunMinRes} \cong \text{id}_{\mathbf{Pos}_U}, \quad (94)$$

form an equivalence for  $\mathbf{DP}$  and  $\mathbf{Pos}_U$ .

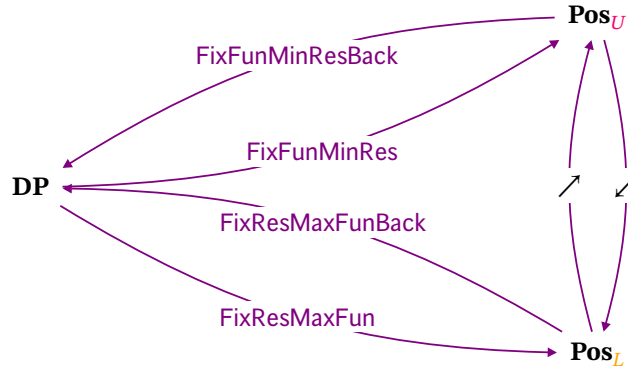


Figure 2.: From  $\mathbf{DP}$  to  $\mathbf{Pos}_U$  and  $\mathbf{Pos}_L$ , and back.

*Proof.* First, consider any morphism in  $\mathbf{Hom}_{\mathbf{DP}}(X; Y)$ . We have

$$\begin{aligned}
 & (\text{FixFunMinRes} \circ \text{FixFunMinResBack})(\mathbf{d})(x^*, y) \\
 &= y \in \text{FixFunMinRes}(\mathbf{d})^*(x) \\
 &= y \in \{y' \in Y \mid \mathbf{d}(x^*, y')\} \\
 &= \mathbf{d}(x^*, y) \\
 &= \text{id}_{\mathbf{DP}}(\mathbf{d})(x^*, y).
 \end{aligned} \tag{95}$$

Now consider any morphism  $H_{\mathbf{d}} \in \mathbf{Hom}_{\mathbf{Pos}_U}(X; Y)$ . We have

$$\begin{aligned}
 & (\text{FixFunMinResBack} \circ \text{FixFunMinRes})(H_{\mathbf{d}})(x) \\
 &= \{y \in Y \mid y \in H_{\mathbf{d}}^*(x)\} \\
 &= \{y \in Y \mid \mathbf{d}(x^*, y)\} \\
 &= \text{id}_{\mathbf{Pos}_U}(H_{\mathbf{d}})(x).
 \end{aligned} \tag{96}$$

□

**Lemma 36.29.**  $\text{FixFunMinRes}$  preserves the bounded lattice structure.

*Proof.* Given  $X, Y \in \mathbf{Ob}_{\mathbf{DP}}$  and  $\mathbf{d}, \mathbf{e} \in \mathbf{Hom}_{\mathbf{DP}}(X; Y)$ , we want to check the following properties.

**Order reversing:** We want to check

$$\frac{\mathbf{d} \leq_{\mathbf{DP}} \mathbf{e}}{\text{FixFunMinRes}(\mathbf{d}) \geq_{\mathbf{Pos}_U} \text{FixFunMinRes}(\mathbf{e})}. \tag{97}$$

We have:

$$\begin{aligned}
 \text{FixFunMinRes}(\mathbf{d})^*(x) &= \{y \in Y \mid \mathbf{d}(x^*, y)\} \\
 &\subseteq \{y \in Y \mid \mathbf{e}(x^*, y)\} \\
 &= \text{FixFunMinRes}(\mathbf{e})^*(x),
 \end{aligned} \tag{98}$$

implying  $\text{FixFunMinRes}(\mathbf{d}) \geq_{\mathbf{Pos}_U} \text{FixFunMinRes}(\mathbf{e})$ .

**Meet and join preservation:** We want to check

$$\text{FixFunMinRes}(\mathbf{d} \wedge \mathbf{e}) = \text{FixFunMinRes}(\mathbf{d}) \wedge_{\mathbf{Pos}_U} \text{FixFunMinRes}(\mathbf{e}), \tag{99}$$

and

$$\text{FixFunMinRes}(\mathbf{d} \vee \mathbf{e}) = \text{FixFunMinRes}(\mathbf{d}) \vee_{\text{Pos}_U} \text{FixFunMinRes}(\mathbf{e}). \quad (100)$$

We have

$$\begin{aligned} & \text{FixFunMinRes}(\mathbf{d} \wedge \mathbf{e})^*(x) \\ &= \{y \in Y \mid (\mathbf{d} \wedge \mathbf{e})(x^*, y)\} \\ &= \{y \in Y \mid (\mathbf{d}(x^*, y) \wedge_{\text{DP}} \mathbf{e}(x^*, y))\} \\ &= \{y \in Y \mid \mathbf{d}(x^*, y)\} \cap \{y \in Y \mid \mathbf{e}(x^*, y)\} \\ &= \text{FixFunMinRes}(\mathbf{d})^*(x) \wedge_{\text{Pos}_U} \text{FixFunMinRes}(\mathbf{e})^*(x). \end{aligned} \quad (101)$$

Similarly:

$$\begin{aligned} & \text{FixFunMinRes}(\mathbf{d} \vee \mathbf{DPe})^*(x) \\ &= \{y \in Y \mid (\mathbf{d} \vee \mathbf{e})(x^*, y)\} \\ &= \{y \in Y \mid (\mathbf{d}(x^*, y) \vee_{\text{DP}} \mathbf{e}(x^*, y))\} \\ &= \{y \in Y \mid \mathbf{d}(x^*, y)\} \cup \{y \in Y \mid \mathbf{e}(x^*, y)\} \\ &= \text{FixFunMinRes}(\mathbf{d})^*(x) \vee_{\text{Pos}_U} \text{FixFunMinRes}(\mathbf{e})^*(x). \end{aligned} \quad (102)$$

**Top and bottom preservation:** We want to check

$$\text{FixFunMinRes}(\perp_{\text{Hom}_{\text{DP}}(X;Y)}) = \perp_{\text{Hom}_{\text{Pos}_U}(X;Y)}, \quad (103)$$

and

$$\text{FixFunMinRes}(\top_{\text{Hom}_{\text{DP}}(X;Y)}) = \top_{\text{Hom}_{\text{Pos}_U}(X;Y)}. \quad (104)$$

We have

$$\begin{aligned} \text{FixFunMinRes}(\perp_{\text{Hom}_{\text{DP}}(X;Y)})^*(x) &= \emptyset \\ &= \perp_{\text{Hom}_{\text{Pos}_U}(X;Y)}^*(x) \end{aligned} \quad (105)$$

Similarly

$$\begin{aligned} \text{FixFunMinRes}(\top_{\text{Hom}_{\text{DP}}(X;Y)})^*(x) &= Y \\ &= \top_{\text{Hom}_{\text{Pos}_U}(X;Y)}^*(x). \end{aligned} \quad (106)$$

□

**Lemma 36.30.**  $\text{FixFunMinRes}$  preserves traces. In other words:

*Proof.* We want to show that

$$\text{FixFunMinRes}(\text{Tr}_{X,Y}^Z(\mathbf{d})) = \text{Tr}_{X,Y}^Z(\text{FixFunMinRes}(\mathbf{d})), \quad (107)$$

for all  $\mathbf{d} \in \text{Hom}_{\text{DP}}(X \times Z; Y \times Z)$ , and  $X, Y, Z \in \text{Ob}_{\text{DP}}$ . On one hand, we have

$$\begin{aligned} \text{FixFunMinRes}(\text{Tr}_{X,Y}^Z(\mathbf{d}))^*(x) &= \{y \in Y \mid \text{Tr}_{X,Y}^Z(\mathbf{d})(x^*, y)\} \\ &= \{y \in Y \mid \bigvee_{z \in Z} \mathbf{d}(\langle x, z \rangle^*, \langle y, z \rangle)\} \end{aligned} \quad (108)$$

On the other hand, we have

$$\begin{aligned}
 & \text{Tr}_{X,Y}^Z(\text{FixFunMinRes}(\mathbf{d}))^*(x) \\
 &= \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in \text{FixFunMinRes}(\mathbf{d})^*(x, z)\} \\
 &= \{y \in Y \mid \bigvee_{z \in Z} \langle y, z \rangle \in \{\langle y, z \rangle \in Y \times Z \mid \mathbf{d}(\langle x, z \rangle^*, \langle y, z \rangle)\}\} \\
 &= \{y \in Y \mid \bigvee_{z \in Z} \mathbf{d}(\langle x, z \rangle^*, \langle y, z \rangle)\}.
 \end{aligned} \tag{109}$$

□



## 37. Solving finite co-design problems

In this chapter we discuss the solution of *finite* design problem. A finite co-design problem is one in which the upper set of the solutions can be described as the upper closure of a finite antichain. Therefore, the solution can be represented with finite memory.

37.1 Domain theory and fixed points	548
37.2 Finite co-design problems . . . .	552
37.3 Handling loops . . . . .	553
37.4 Example: Optimizing over the natural numbers . . . . .	556
37.5 Extended Numerical Examples .	559
37.6 Complexity of the solution . . . .	568
37.7 Decomposition of CDPs . . . . .	570



### 37.1. Domain theory and fixed points

In this section we recall some fundamentals of domain theory. It is used in computer science for defining denotational semantics (see *e.g.*, [20]). It is used in embedded systems for defining the semantics of models of computation (see, *e.g.*, [15]). What we need from domain theory is the least necessary to define *least fixed points* and to use Kleene's theorem.

Domain theory builds on order theory by defining “directed” and “complete” partial orders. These attributes play the same role as compactness in analysis: they will be used to make sure that certain sequences can converge to a fixed point.

#### Directed and complete partial orders

##### Definition 37.1 (Directed set)

In a poset  $P = \langle P, \leq_P \rangle$ , we say that a set  $S \subseteq P$  is *directed* if each pair of elements in  $S$  has an upper bound: for all  $x, y \in S$ , there exists  $z \in S$  such that  $x \leq z$  and  $y \leq z$ .

##### Definition 37.2 (Completeness)

A poset is a *directed complete partial order* (DCPO) if each of its directed subsets has a supremum (least of upper bounds). It is a *complete partial order* (CPO) if it also has a bottom.

**Example 37.3** (Completion of  $\mathbb{R}_{\geq 0}$  to  $\overline{\mathbb{R}_{\geq 0}}$ ). The poset  $\langle \mathbb{R}, \leq \rangle$  is not a CPO, because it lacks a bottom.

The non-negative reals  $\mathbb{R}_{\geq 0} = \{x \in \mathbb{R} : x \geq 0\}$  have a bottom  $\perp = 0$ , however, they are not a DCPO because some of their directed subsets do not have an upper bound. For example, take  $\mathbb{R}_{\geq 0}$ , which is a subset of  $\mathbb{R}_{\geq 0}$ . Then  $\mathbb{R}_{\geq 0}$  is directed, because for each  $a, b \in \mathbb{R}_{\geq 0}$ , there exists  $c = \max\{a, b\} \in \mathbb{R}_{\geq 0}$  for which  $a \leq c$  and  $b \leq c$ .

One way to make  $\langle \mathbb{R}_{\geq 0}, \leq \rangle$  a CPO is by adding an artificial top element  $\top$  that we think as “a point at infinitely”. We can define then the completion

$$\overline{\mathbb{R}_{\geq 0}} := \mathbb{R}_{\geq 0} \cup \{\top\}, \quad (1)$$

and extending the partial order  $\leq$  so that  $a \leq \top$  for all  $a \in \mathbb{R}_{\geq 0}$ .

**Example 37.4.** Any lattice is a DCPO.

**Example 37.5.** For any poset  $P$ ,  $\top P$  is a CPO, because it is a bounded lattice.

#### Scott continuity

Scott continuity is a property of maps on DCPOs that is slightly stronger than monotonicity.

##### Definition 37.6 (Scott continuity)

A map  $f : P \rightarrow_{\text{Pos}} Q$  between DCPOs is *Scott continuous* iff for each directed subset  $S \subseteq P$ , the image  $f(S)$  is directed, and

$$f(\text{Sup } S) = \text{Sup } f(S). \quad (2)$$

**Lemma 37.7.** Scott continuity implies monotonicity.

*Proof.* Consider a map  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{Q}$  that is Scott continuous. Take two elements  $x, y \in \mathbf{P}$  such that  $x \leq y$ . The set  $\mathbf{S} = \{x, y\}$  is directed. From (2), we know that

$$f(\text{Sup } \mathbf{S}) = f(y) = \text{Sup } \{f(x), f(y)\}, \quad (3)$$

which implies that  $f(x) \leq f(y)$ . Therefore,  $f$  is monotone.  $\square$

**Remark 37.8.** Scott continuity is not the same as the notion of continuity as used in analysis you might be familiar with. A map from the CPO  $\langle \overline{\mathbb{R}}_{\geq 0}, \leq \rangle$  to itself is Scott continuous iff it is nondecreasing and left-continuous. For example, the ceiling function  $x \mapsto \lceil x \rceil$  is not continuous in the usual sense, but it is Scott continuous (Fig. 1).

However, the name “continuity” for this property is aptly chosen. In analysis, a function is continuous if it preserves limits, in the sense that

$$\lim_{n \rightarrow \infty} f(a_n) = f(\lim_{n \rightarrow \infty} a_n), \quad (4)$$

which is, in spirit, the same as (2).

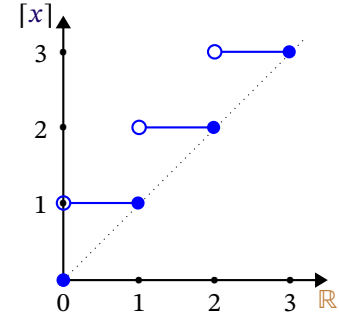


Figure 1.: The ceiling function is Scott continuous.

## Least fixed points

### Definition 37.9 (Fixed points)

A *fixed point* of  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{P}$  is a point  $x$  such that  $f(x) = x$ .

### Definition 37.10 (Least fixed points)

A *least fixed point* of  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{P}$  is the minimum (if it exists) of the set of fixed points of  $f$ :

$$\text{lfp}(f) := \min_{\leq} \{x \in \mathbf{P} : f(x) = x\}. \quad (5)$$

In general, a function need not have a fixed point. It also might have multiple fixed points; and also in that case there might not be a *least* fixed point.

However, the conditions for a least fixed point to exist are quite weak. Monotonicity of the map  $f$  plus completeness is sufficient to ensure existence.

**Lemma 37.11.** If  $\mathbf{P}$  is a CPO and  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{P}$  is monotone, then  $\text{lfp}(f)$  exists and is unique.

This is given as CPO Fixpoint Theorem II, 8.22 in [4].

With the additional assumption of Scott continuity, Kleene’s algorithm is a systematic procedure to find the least fixed point.

**Lemma 37.12** (Kleene’s fixed-point theorem). Assume  $\mathbf{P}$  is a CPO, and  $f : \mathbf{P} \rightarrow_{\mathbf{Pos}} \mathbf{P}$  is Scott continuous. Then the least fixed point of  $f$  is the supremum of the Kleene ascent chain

$$\perp \leq f(\perp) \leq f(f(\perp)) \leq \dots \leq f^{(n)}(\perp) \leq \dots \quad (6)$$

This is given as CPO fixpoint theorem I, 8.15 in [4].

## Example: party invite

Consider again the party scenario of Example 5.27. Consider the case where a subset  $\mathbf{S} \subseteq \mathbf{A}$  of people decide to throw a party. They then proceed to call all their

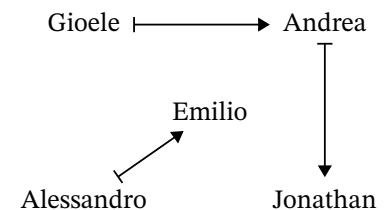


Figure 2.: Party invite relation.

friends, who accept, and, if they were not invited already, enthusiastically call *their* friends to extend the invite. We want to find out what is the final group of people that will show up at the party. We call this map  $\phi : \text{Pow } \mathbf{A} \rightarrow \text{Pow } \mathbf{A}$ , so that if  $\mathbf{S}$  is the initial group,  $\phi(\mathbf{S})$  is the complete set of invites.

Note that this is related to the transitive closure operation, but we are only interested in the transitive closure from a certain initial set  $\mathbf{S}$ .

For example, consider the case in which the relation is as in Fig. 2. In this case, we would have

$$\phi(\emptyset) = \emptyset, \quad (7)$$

which means that, if nobody starts a party, no party takes place. Jonathan does not invite anybody, so we would have

$$\phi(\{\text{Jonathan}\}) = \{\text{Jonathan}\} \quad (8)$$

If Gioele and Alessandro start the party, everybody will get invited:

$$\phi(\{\text{Alessandro, Gioele}\}) = \text{everybody}. \quad (9)$$

We can show that

1. The function  $\phi$  can be computed as a fixed point.
2. The recursive invite strategy corresponds to Kleene's iteration.

We summarize the properties that we want the function  $\phi$  to have. Given an initial subset  $\mathbf{S}$ , we would like to find the set of people  $\mathbf{T} = \phi(\mathbf{S})$  such that:

1.  $\mathbf{T}$  contains the initial set  $\mathbf{S}$ :

$$\mathbf{S} \subseteq \mathbf{T} \quad (10)$$

2.  $\mathbf{T}$  is closed with respect to a certain invite relation  $R : \mathbf{A} \rightarrow \mathbf{A}$ . If  $x R y$ , then  $x$  invites  $y$  to the party. Define the function

$$\begin{aligned} m : \text{Pow } \mathbf{A} &\rightarrow \text{Pow } \mathbf{A}, \\ \mathbf{T} &\mapsto \mathbf{T} \cup \bigcup_{x \in \mathbf{T}} \{y \in \mathbf{A} : x R y\}. \end{aligned} \quad (11)$$

This represents one iteration of the invite process: given a set  $\mathbf{T}$ , we add to  $\mathbf{T}$  all invitees of each of the elements of  $\mathbf{T}$ .

We are looking for a set  $\mathbf{T}$  such that it is a fixed point of the invite function:

$$\mathbf{T} = m(\mathbf{T}). \quad (12)$$

3.  $\phi(\mathbf{S})$  is the smallest among all such sets that satisfy the two conditions above. Let  $\mathbf{P}$  be the upper principal set of  $\mathbf{S}$ : given (10), we know that we want sets that contain at least  $\mathbf{S}$ :

$$\mathbf{P} = \uparrow \mathbf{S} = \{\mathbf{T} \in \text{Pow } \mathbf{A} : \mathbf{S} \subseteq \mathbf{T}\}. \quad (13)$$

The poset  $\mathbf{P}$  is a sublattice of  $\text{Pow } \mathbf{A}$ . Note also that the bottom of  $\mathbf{P}$  is  $\mathbf{S}$ .

In summary, we are looking for the smallest point of  $\mathbf{P}$  that is closed to  $m$ :

$$\phi(\mathbf{S}) = \min_{\subseteq} \{\mathbf{T} \in \mathbf{P} : \mathbf{T} = m(\mathbf{T})\} \quad (14)$$

Comparing this with (5), we see that  $\phi(\mathbf{S})$  is the least fixed point of  $m$ :

$$\phi(\mathbf{S}) = \text{lfp}(m). \quad (15)$$

Take Kleene's iteration in (6):

$$\perp \leq f(\perp) \leq f(f(\perp)) \leq \dots \leq f^{(n)}(\perp) \leq \dots \quad (16)$$

Because the bottom of  $\mathbf{P} = \uparrow \mathbf{S}$  is  $\mathbf{S}$ , we can rewrite it as:

$$\mathbf{S} \subseteq m(\mathbf{S}) \subseteq m(m(\mathbf{S})) \subseteq m(m(m(\mathbf{S}))) \dots \quad (17)$$

Each element of the sequence corresponds to one iteration of the invite algorithm.

## 37.2. Finite co-design problems

If we want a computable algorithm for solving co-design queries, it is necessary that the solution can be finitely representable. One way to do this is to zero-in on those design problems that are guaranteed, by construction, to have a finite solution. This is what we do in this section. In the next chapters, we will see how we can construct bounded finite solutions to non-finitely-representable DPs.

In the **FixFunMinRes** queries, the solution lives in an upper set of resources. We now look at upper sets that can be represented as the upper closure of a finite antichains.

**Definition 37.13** (Finitely-supported upper sets)

Given a poset  $\mathbf{P}$ , we call an upper set  $\mathbf{S} \in \mathbf{UP}$  *finitely supported* if it can be written as the upper closure of a finite antichain:

$$\mathbf{S} = (\uparrow \alpha), \text{ for } \alpha \in \text{Anti } \mathbf{P}, \text{ card}(\alpha) < \infty. \quad (18)$$

We call  $\overline{\mathbf{U}}_{\mathbf{f}} \mathbf{P}$  the set of finitely-supported upper sets of a poset  $\mathbf{P}$ .

We call  $\text{Anti}_{\mathbf{f}} \mathbf{P}$  the set of finite antichains.

**Definition 37.14** (Finite design problems)

We call a design problem finite if, in its representation  $H : \mathbf{F} \rightarrow \mathbf{UR}$ ,  $h(f) \in \overline{\mathbf{U}}_{\mathbf{f}} \mathbf{R}$  for all  $f \in \mathbf{F}$ .

We show that finite co-design problems form a subcategory of **DP** that is also monoidal and locally posetal. (Note that we are leaving out “traced” for now.) To show this, we just need to check that all the ways to compose finite DPs result in finite DPs. The formulas that we derive work also describe an algorithm to compute the solution to the queries.

**Definition 37.15** (Category of finite design problems  $\mathbf{Pos}_{\overline{\mathbf{U}}_{\mathbf{f}}}$ )

The category of *finite design problems*  $\mathbf{Pos}_{\overline{\mathbf{U}}_{\mathbf{f}}}$  consists of the following constituents:

1. *Objects*: The objects are posets.
2. *Morphisms*: The morphisms are *finite* design problems (Def. 37.14).
3. *Identity morphism*: The identity morphism  $\text{id}_{\mathbf{P}} : \mathbf{P} \leftrightarrow \mathbf{P}$  is as in **DP**.
4. *Composition operation*: Given morphisms  $\mathbf{d} : \mathbf{P} \leftrightarrow \mathbf{Q}$  and  $\mathbf{e} : \mathbf{Q} \leftrightarrow \mathbf{R}$ , their composition  $\mathbf{d} \circ \mathbf{e} : \mathbf{P} \leftrightarrow \mathbf{R}$  is as in **DP**.

### 37.3. Handling loops

We are close to having a complete solution. The only part that is missing is dealing with loops (trace).

First, we will work with a particular form of loops, called “the Conway form”, shown in Fig. 3. This corresponds to working with design problems of the type

$$\mathbf{d} : \mathbf{A} \times \mathbf{B} \mapsto \mathbf{B}, \quad (19)$$

The new feedback operator has signature

$$\text{loop} : (\mathbf{A} \times \mathbf{B} \mapsto \mathbf{B}) \rightarrow (\mathbf{A} \mapsto \mathbf{B}) \quad (20)$$

We can do this without loss of generality, because we can re-write the trace with this new operator. We leave Fig. 4 as a graphical proof that this is possible. Hagesawa [11] discusses the equivalence in detail.

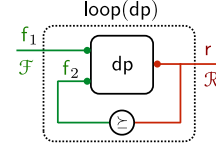


Figure 3.

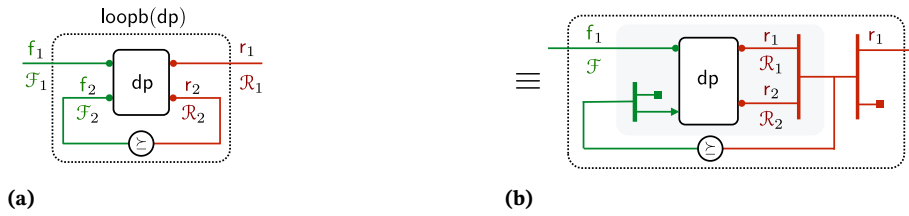


Figure 4.: We can rewrite the trace in Conway’s form.

The following theorem establishes a closed form for  $h_{\text{loop}(\mathbf{d})}$  as a least fixed point. Here on we consider  $\text{Anti}_f \mathbf{R}$  as a poset with the order given by

$$\frac{\alpha_1 \leq_{\text{Anti}_f \mathbf{R}} \alpha_2}{\uparrow \alpha_1 \leq_{\uparrow \mathbf{R}} \uparrow \alpha_2}. \quad (21)$$

**Theorem 37.16.** For any DP  $\mathbf{d}$  of the right shape, we can compute  $h_{\text{loop}(\mathbf{d})}$  as follows:

$$h_{\text{loop}(\mathbf{d})} : f_1 \mapsto \uparrow \text{lfp}(\Phi_{f_1}), \quad (22)$$

that is, as the *least fixed point* of a map  $\Phi_{f_1}$  defined as

$$\begin{aligned} \Phi_{f_1} : \text{Anti}_f \mathbf{R} &\rightarrow \text{Anti}_f \mathbf{R}, \\ \alpha &\mapsto \text{Min}_{\leq_{\mathbf{R}}} \bigcup_{r \in \alpha} h_{\mathbf{d}}(f_1, r) \cap \uparrow r. \end{aligned} \quad (23)$$

*Proof.* The diagram in Fig. 3 implies that the map  $h_{\text{loop}(\mathbf{d})}$  can be described as:

$$h_{\text{loop}(\mathbf{d})} : \mathbf{F}_1 \rightarrow \text{Anti}_f \mathbf{R}, \quad (24)$$

$$f_1 \mapsto \begin{cases} \text{using} & r, f_2 \in \mathbf{R}, \\ \text{Min}_{\leq_{\mathbf{R}}} & r, \\ \text{s.t.} & r \in h_{\mathbf{d}}(f_1, f_2), \\ & r \leq_{\mathbf{R}} f_2. \end{cases} \quad (25)$$

Denote by  $h_{f_1}$  the map  $h_{\mathbf{d}}$  with the first element fixed:

$$h_{f_1} : f_2 \mapsto h_{\mathbf{d}}(f_1, f_2). \quad (26)$$

Rewrite  $r \in h_d(f_1, f_2)$  in (24) as

$$r \in h_{f_1}(f_2). \quad (27)$$

Let  $r$  be a feasible solution, but not necessarily minimal. Lemma 37.17 implies that the constraint (27) can be rewritten as

$$\{r\} = h_{f_1}(f_2) \cap \uparrow r. \quad (28)$$

Because  $f_2 \geq r$ , and  $h_{f_1}$  is Scott continuous, it follows that  $h_{f_1}(f_2) \geq_{\text{Anti}_f \mathbf{R}} h_{f_1}(r)$ . Therefore, by Lemma 37.18, we have

$$\{r\} \geq_{\text{Anti}_f \mathbf{R}} h_{f_1}(r) \cap \uparrow r. \quad (29)$$

This is a recursive condition that all feasible  $r$  must satisfy.

Let  $\alpha \in \text{Anti}_f \mathbf{R}$  be an antichain of feasible resources, and let  $r$  be a generic element of  $\mathbf{R}$ . Tautologically, rewrite  $\alpha$  as the minimal elements of the union of the singletons containing its elements:

$$\alpha = \text{Min}_{\leq \mathbf{R}} \bigcup_{r \in \alpha} \{r\}. \quad (30)$$

Substituting (29) in (30) we obtain (cf Lemma 37.19)

$$\alpha \geq_{\text{Anti}_f \mathbf{R}} \text{Min}_{\leq \mathbf{R}} \bigcup_{r \in \alpha} h_{f_1}(r) \cap \uparrow r. \quad (31)$$

Converse: It is also true that if an antichain  $\alpha$  satisfies (31) then all  $r \in \alpha$  are feasible. The constraint (31) means that for any  $r_0 \in \alpha$  on the left side, we can find a  $r_1$  on the right side so that  $r_0 \geq_{\mathbf{R}} r_1$ . The point  $r_1$  needs to belong to one of the sets of which we take the union; say that it comes from  $r_2 \in \alpha$ , so that  $r_1 \in h_{f_1}(r_2) \cap \uparrow r_2$ . Summarizing:

$$\forall r_0 \in \alpha : \exists r_1 : (r_0 \geq_{\mathbf{R}} r_1) \wedge (\exists r_2 \in \alpha : r_1 \in h_{f_1}(r_2) \cap \uparrow r_2). \quad (32)$$

Because  $r_1 \in h_{f_1}(r_2) \cap \uparrow r_2$ , we can conclude that  $r_1 \in \uparrow r_2$ , and therefore  $r_1 \geq_{\mathbf{R}} r_2$ , which together with  $r_0 \geq_{\mathbf{R}} r_1$ , implies  $r_0 \geq_{\mathbf{R}} r_2$ . We have concluded that there exist two points  $r_0, r_2$  in the antichain  $\alpha$  such that  $r_0 \geq_{\mathbf{R}} r_2$ ; therefore, they are the same point:  $r_0 = r_2$ . Because  $r_0 \geq_{\mathbf{R}} r_1 \geq_{\mathbf{R}} r_2$ , we also conclude that  $r_1$  is the same point as well. We can rewrite (32) by using  $r_0$  in place of  $r_1$  and  $r_2$  to obtain  $\forall r_0 \in \alpha : r_0 \in h_{f_1}(r_0)$ , which means that  $r_0$  is a feasible resource.

We have concluded that all antichains of feasible resources  $\alpha$  satisfy (31), and conversely, if an antichain  $\alpha$  satisfies (31), then it is an antichain of feasible resources.

Equation (31) is a recursive constraint for  $\alpha$ , of the kind

$$\Phi_{f_1}(\alpha) \leq_{\text{Anti}_f \mathbf{R}} \alpha, \quad (33)$$

with the map  $\Phi_{f_1}$  defined by

$$\begin{aligned} \Phi_{f_1} : \text{Anti}_f \mathbf{R} &\rightarrow \text{Anti}_f \mathbf{R}, \\ \alpha &\mapsto \text{Min}_{\leq \mathbf{R}} \bigcup_{r \in \alpha} h_{f_1}(r) \cap \uparrow r. \end{aligned} \quad (34)$$

If we want the *minimal* resources, we are looking for the *least* antichain:

$$\min_{\leq_{\text{Anti}_f \mathbf{R}}} \{ \alpha \in \text{Anti}_f \mathbf{R} : \Phi_{f_1}(\alpha) \leq_{\text{Anti}_f \mathbf{R}} \alpha \}, \quad (35)$$

which is equal to the *least fixed point* of  $\Phi_{f_1}$ . Therefore, the map  $h_{\text{loop}(\mathbf{d})}$  can be written as

$$h_{\text{loop}(\mathbf{d})} : f_1 \mapsto \text{lfp}(\Phi_{f_1}). \quad (36)$$

Lemma 37.20 shows that  $\text{lfp}(\Phi_{f_1})$  is Scott continuous in  $f_1$ .  $\square$

**Lemma 37.17.** Let  $\mathbf{S}$  be an antichain in  $\mathbf{P}$ . Then

$$\frac{x \in \mathbf{S}}{\{x\} = \mathbf{S} \cap \uparrow x}. \quad (37)$$

**Lemma 37.18.** For  $\mathbf{S}, \mathbf{T} \in \text{Anti}_f \mathbf{P}$ , and  $\mathbf{A} \subseteq \mathbf{B}$ ,  $\mathbf{S} \leq_{\text{Anti}_f \mathbf{R}} \mathbf{T}$  implies  $\mathbf{S} \cap \mathbf{A} \leq_{\text{Anti}_f \mathbf{R}} \mathbf{T} \cap \mathbf{A}$ .

**Lemma 37.19.** For  $\mathbf{S}, \mathbf{T}, \mathbf{U}, \mathbf{V} \in \text{Anti}_f \mathbf{P}$ ,  $\mathbf{S} \leq_{\text{Anti}_f \mathbf{R}} \mathbf{U}$  and  $\mathbf{T} \leq_{\text{Anti}_f \mathbf{R}} \mathbf{V}$  implies  $\mathbf{S} \cup \mathbf{T} \leq_{\text{Anti}_f \mathbf{R}} \mathbf{U} \cup \mathbf{V}$ .

**Lemma 37.20.** Let  $f : \mathbf{P} \times \mathbf{Q} \rightarrow_{\text{Pos}} \mathbf{Q}$  be Scott continuous. For each  $x \in \mathbf{P}$ , define the map

$$f_x : y \mapsto f(x, y) \quad (38)$$

Then the map

$$f^\dagger : x \mapsto \text{lfp}(f_x) \quad (39)$$

is Scott continuous.

*Proof.* Davey and Priestly [4] leave this as Exercise 8.26. A proof is found in Gierz *et al.* [7, Exercise II-2.29].  $\square$



### 37.4. Example: Optimizing over the natural numbers

This is a simple example that can show two interesting properties of CDPIs:

1. the ability to work with discrete posets; and
2. the ability to treat multi-objective optimization problems.

Consider the family of optimization problems indexed by  $c \in \mathbb{N}$ :

$$\left\{ \begin{array}{ll} \text{Min}_{\leq_{\mathbb{N} \times \mathbb{N}}} & \langle x, y \rangle, \\ \text{s.t.} & x + y \geq \lceil \sqrt{x} \rceil + \lceil \sqrt{y} \rceil + c. \end{array} \right. \quad (40)$$

One can show that this optimization problem is a CDP by producing a co-design diagram with an equivalent semantics, such as the one in Fig. 5.

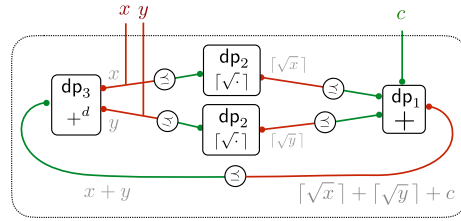


Figure 5.: Co-design diagram equivalent to (40)

The diagram contains three primitive DPIs:  $\mathbf{d}_1$ ,  $\mathbf{d}_2$  (used twice), and  $\mathbf{d}_3$ . Their  $h$  maps are:

$$\begin{aligned} h_1 : \overline{\mathbb{N}} \times \overline{\mathbb{N}} \times \overline{\mathbb{N}} &\rightarrow \text{Anti}_f \overline{\mathbb{N}}, \\ \langle f_1, f_2, f_3 \rangle &\mapsto \{f_1 + f_2 + f_3\}, \\ h_2 : \overline{\mathbb{N}} &\rightarrow \text{Anti}_f \overline{\mathbb{N}}, \\ f &\mapsto \{\lceil \sqrt{f} \rceil\}, \\ h_3 : \overline{\mathbb{N}} &\rightarrow \text{Anti}_f (\overline{\mathbb{N}} \times \overline{\mathbb{N}}), \\ f &\mapsto \{\langle a, b \rangle \in \overline{\mathbb{N}} \times \overline{\mathbb{N}} : a + b = f\}. \end{aligned}$$

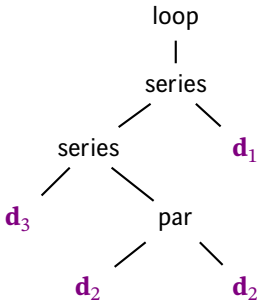


Figure 6.

The tree decomposition (Fig. 6) corresponds to the expression

$$\mathbf{d} = \text{loop}(\text{series}(\text{par}(\mathbf{d}_2, \mathbf{d}_2), \text{series}(\mathbf{d}_1, \mathbf{d}_3))). \quad (41)$$

From (41) we obtain an expression for  $h$ :

$$h = ((h_2 \otimes h_2) \odot h_1 \odot h_3)^\dagger. \quad (42)$$

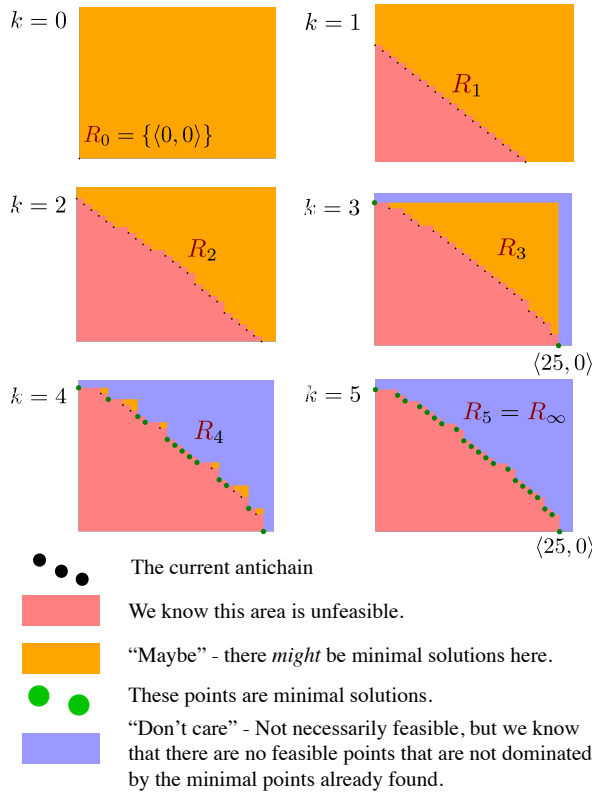
This problem is small enough that we can write down an explicit expression for  $h$ . By substituting in (42) the definitions for  $\otimes$ ,  $\dagger$ ,  $\odot$ , we obtain that evaluating  $h(c)$  means finding the least fixed point of a map  $\Psi_c$ :

$$h : c \mapsto \text{lfp}(\Psi_c). \quad (43)$$

The map  $\Psi_c : \text{Anti}_f (\overline{\mathbb{N}} \times \overline{\mathbb{N}}) \rightarrow \text{Anti}_f (\overline{\mathbb{N}} \times \overline{\mathbb{N}})$  can be obtained from Theorem 37.16 as follows:

$$\Psi_c : \alpha \mapsto \text{Min} \bigcup_{\langle x, y \rangle \in \alpha} \uparrow \langle x, y \rangle n \quad (44)$$

$$n \{ \langle a, b \rangle \in \mathbb{N}^2 : (a + b \geq \lceil \sqrt{x} \rceil + \lceil \sqrt{y} \rceil + c) \}. \quad (45)$$



**Figure 7.:** Kleene ascent to solve the problem (40) for  $c = 20$ . The sequence converges in five steps to  $\alpha_5 = \alpha_\infty$ .

Kleene’s algorithm is the iteration  $\alpha_{k+1} = \Psi_c(\alpha_k)$  starting from

$$\alpha_0 = \perp_{\text{Antif}(\overline{\mathbb{N}} \times \overline{\mathbb{N}})} = \{\langle 0, 0 \rangle\}. \quad (46)$$

For  $c = 0$ , the sequence converges immediately:

$$\alpha_0 = \{\langle \mathbf{0}, \mathbf{0} \rangle\} = h(\mathbf{0}). \quad (47)$$

For  $c = 1$ , the sequence converges at the sixth step; however, some solutions (in bold) converge sooner:

$$\alpha_0 = \{\langle 0, 0 \rangle\}, \quad (48)$$

$$\alpha_1 = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle\}, \quad (49)$$

$$\alpha_2 = \{\langle 0, 2 \rangle, \langle 1, 1 \rangle, \langle 2, 0 \rangle\}, \quad (50)$$

$$\alpha_3 = \{\langle \mathbf{0}, \mathbf{3} \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle \mathbf{3}, \mathbf{0} \rangle\}, \quad (51)$$

$$\alpha_4 = \{\langle \mathbf{0}, \mathbf{3} \rangle, \langle 2, 2 \rangle, \langle \mathbf{3}, \mathbf{0} \rangle\}, \quad (52)$$

$$\alpha_5 = \{\langle \mathbf{0}, \mathbf{3} \rangle, \langle \mathbf{3}, \mathbf{0} \rangle\} = h(\mathbf{1}). \quad (53)$$

For  $c = 2$ , the sequence converges at the fifth step; however, some solutions (in bold) converge sooner:

$$\alpha_0 = \{\langle 0, 0 \rangle\}, \quad (54)$$

$$\alpha_1 = \{\langle 0, 2 \rangle, \langle 1, 1 \rangle, \langle 2, 0 \rangle\}, \quad (55)$$

$$\alpha_2 = \{\langle \mathbf{0}, \mathbf{4} \rangle, \langle 1, 3 \rangle, \langle 2, 2 \rangle, \langle 3, 1 \rangle, \langle \mathbf{4}, \mathbf{0} \rangle\}, \quad (56)$$

$$\alpha_3 = \{\langle \mathbf{0}, \mathbf{4} \rangle, \langle 3, 2 \rangle, \langle 2, 3 \rangle, \langle \mathbf{4}, \mathbf{0} \rangle\} \quad (57)$$

$$\alpha_4 = \{\langle \mathbf{0}, \mathbf{4} \rangle, \langle \mathbf{3}, \mathbf{3} \rangle, \langle \mathbf{4}, \mathbf{0} \rangle\} = h(\mathbf{2}). \quad (58)$$

The next values in the sequence are:

$$h(3) = \{\langle 0, 6 \rangle, \langle 3, 4 \rangle, \langle 4, 3 \rangle, \langle 6, 0 \rangle\}, \quad (59)$$

$$h(4) = \{\langle 0, 7 \rangle, \langle 3, 6 \rangle, \langle 4, 4 \rangle, \langle 6, 3 \rangle, \langle 7, 0 \rangle\}. \quad (60)$$

Figure 7 shows the sequence for  $c = 20$ .

### Guarantees of Kleene ascent

Solving an CDP with cycles reduces to computing a Kleene ascent sequence  $\alpha_k$ . At each instant  $k$  we have some additional guarantees.

For any finite  $k$ , the resources “below”  $\alpha_k$  (the set  $\mathbf{R} \setminus \uparrow \alpha_k$ ) are infeasible. (In Fig. 7, those are colored in red.)

If the iteration converges to a non-empty antichain  $\alpha_\infty$ , the antichain  $\alpha_\infty$  divides  $\mathbf{R}$  in two. Below the antichain, all resources are infeasible. However, above the antichain (purple area), it is not necessarily true that all points are feasible, because there might be holes in the feasible set. Note that this method does not compute the entire feasible set, but rather only the *minimal elements* of the feasible set, which might be much easier to compute.

Finally, if the sequence converges to the empty set, it means that there are no solutions. The sequence  $\alpha_k$  can be considered a certificate of infeasibility.

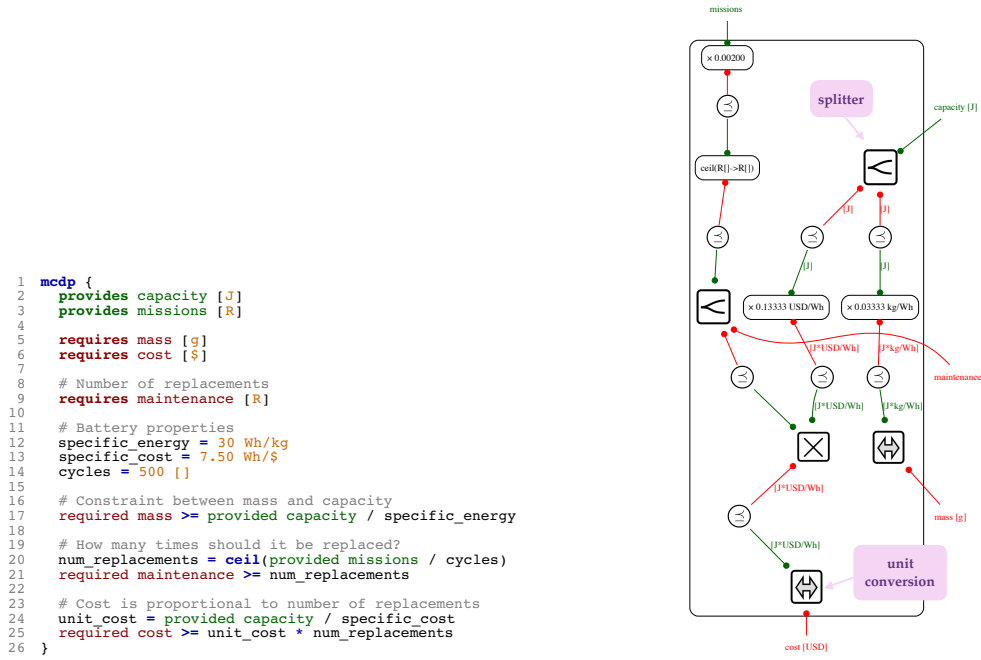
## 37.5. Extended Numerical Examples

This example considers the choice of different battery technologies for a robot. The goals of this example are: 1) to show how design problems can be composed; 2) to show how to define hard constraints and precedence between resources to be minimized; 3) to show how even relatively simple models can give very complex trade-offs surfaces; and 4) to introduce MCDPL, a formal language for the description of co-design problems.

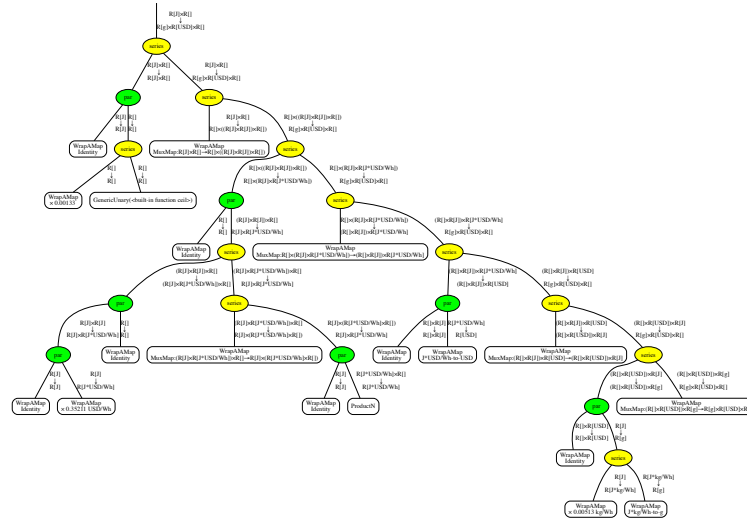
### Language and interpreter/solver

MCDPL is a modeling language to describe CDPs and their compositions. It is inspired by CVX and “disciplined convex programming” [9]. MCDPL is even more disciplined than CVX; for example, multiplying by a negative number is a *syntax* error. The figures are generated by PyMCDP, an interpreter and solver for CDPs, which implements the techniques described in these sections. An in-depth description of MCDPL is available in the next volume of this series.

### Model of a battery



(a) MCDPL code equivalent to equations Eqs. (64) to (66). (b) Co-design diagram generated by PyMCDP from code in panel (b).



(c) Tree representation using par/series of diagram in panel (c).

**Figure 8.:** Panel (c) shows the co-design diagram generated from the code in (b). Panel (d) shows a tree representation (series, parallel) for the diagram. The edges show the types of functionality and resources. The leaves are labeled with the Python class used internally by the interpreter PyMCDP.

The choice of a battery can be modeled as a DPI (Fig. 9) with functionalities **capacity [J]** and **number of missions** and with resources **mass [kg]**, **cost [\$]** and **“maintenance”**, defined as the number of times that the battery needs to be replaced over the lifetime of the robot.

Each battery technology is described by the three parameters specific energy, specific cost, and lifetime (number of cycles):

$$\rho := \text{specific energy [Wh/kg]}, \quad (61)$$

$$\alpha := \text{specific cost [Wh/\$]}, \quad (62)$$

$$c := \text{battery lifetime [\# of cycles]}. \quad (63)$$

The relation between functionality and resources is described by three nonlinear monotone constraints:

$$\text{mass} \geq \text{capacity} / \rho, \quad (64)$$

$$\text{maintenance} \geq \lceil \text{missions} / c \rceil, \quad (65)$$

$$\text{cost} \geq \lceil \text{missions} / c \rceil (\text{capacity} / \alpha). \quad (66)$$

Figure 8a shows the MCDPL code that describes the model corresponding to Eqs. (64) to (66). The diagram in Fig. 8b is automatically generated from the code. Fig. 8c shows a tree representation of the diagram using the series/par operators.

## Competing battery technologies

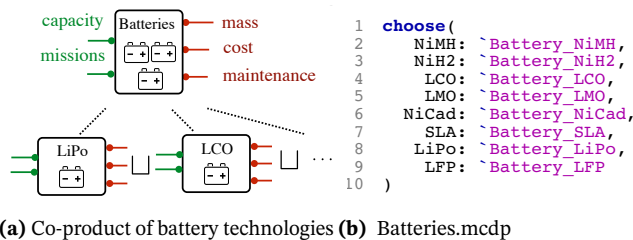
The parameters for the battery technologies used in this example are shown in Table 37.1.

technology	energy density [ Wh/kg]	specific cost [Wh/\\$]	operating life # cycles
NiMH	100	3.41	500
NiH2	45	10.50	20000
LCO	195	2.84	750
LMO	150	2.84	500
NiCad	30	7.50	500
SLA	30	7.00	500
LiPo	250	2.50	600
LFP	90	1.50	1500

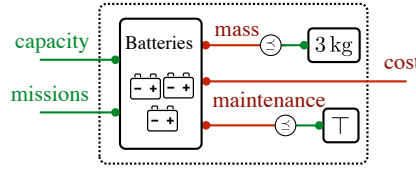
**Table 37.1.:** Specifications of common batteries technologies

Each row of the table is used to describe a model as in Fig. 8a by plugging in the specific values in lines 12–14.

Given the different models, we can define their coproduct (Fig. 10a) using the MCDPL code in Fig. 10b.



**Figure 10.:** The coproduct of design problems describes the choices among different technologies. The MCDPL keyword for the coproduct is “choose”.

(a) Co-design diagram that expresses hard constraints for **mass**.

```

1 mcdp {
2   provides capacity [J]
3   provides missions [R]
4
5   requires cost [$]
6
7   battery = instance `Batteries
8
9   provided capacity <= capacity provided by battery
10  provided missions <= missions provided by battery
11
12  mass required by battery <= 3 kg
13
14  ignore maintenance required by battery
15
16  required cost >= cost required by battery
17 }

```

(b) MCDPL code equivalent to diagram in (a).

**Figure 11.:** Composition of DPs can express hard constraints and precedence of objectives. In this case, there is a hard constraint on the **mass**. Because there is only one outgoing edge for **mass**, and the **cost** and **maintenance** are terminated by a dummy constraint ( $x \leq \top$ ), the semantics of the diagram is that the objective is to minimize the **mass** as primary objective.

## Introducing other variations or objectives

The design problem for the battery has two functionalities (**capacity** and **number of missions**) and three resources (**cost**, **mass**, and **maintenance**). Thus, it describes a family of multi-objective optimization problems, of the type “Given **capacity** and **missions**, minimize **cost**, **mass**, **maintenance**”. We can further extend the class of optimization problems by introducing other hard constraints and by choosing which resource to prioritize. This can be done by composition of design problems; that is, by creating a larger DP that contains the original DP as a subproblem, and contains some additional degenerate DPs that realize the desired semantics.

For example, suppose that we would like to find the optimal solution(s) such that: 1) The mass does not exceed 3 kg; 2) The mass is minimized as a primary objective, while cost/maintenance are secondary objectives.

This semantics can be described by the co-design diagram in Fig. 11a, which contains two new symbols. The DP labeled “3 kg” implements the semantics of hard constraints. It has one functionality ( $F = \overline{\mathbb{R}}_{\geq 0}^{\text{kg}}$ ) and zero resources ( $R = \mathbf{1}$ ). The poset  $\mathbf{1} = \{\langle \rangle\}$  has exactly two antichains:  $\emptyset$  and  $\{\langle \rangle\}$ . These represent “infeasible” and “feasible”, respectively. The DP is described by the map

$$h : \overline{\mathbb{R}}_{\geq 0}^{\text{kg}} \rightarrow \text{Anti } \mathbf{1}, \quad (67) \quad \text{---} \boxed{3 \text{ kg}}$$

$$f \mapsto \begin{cases} \{\langle \rangle\}, & \text{if } f \leq 3 \text{ kg}, \\ \emptyset, & \text{if } f > 3 \text{ kg}. \end{cases} \quad (68)$$

The block labeled “T” is similarly defined and always returns “feasible”, so it has the effect of ignoring **cost** and **maintenance** as objectives. The only resource edge is the one for **mass**, which is then the only objective.

The MCDPL code is shown in Fig. 11b. Note the intuitive interface: the user can directly write “mass required by battery  $\leq 3$  kg” and “ignore maintenance required by battery”, which is compiled to “maintenance required by battery  $\leq \top$ ”.

This relatively simple model for energetics already shows the complexity of CDPs. Figure 14 shows the optimal choice of the battery technology as a function of capacity and number of missions, for several slight variations of the problem that differ in constraints and objectives. For each battery technology, the figures show whether at each operating point the technology is the optimal choice, and

how many optimal choices there are. Some results are intuitive. For example, Fig. 14f shows that if the only objective is minimizing **mass**, then the optimal choice is simply the technology with the largest specific energy (LiPo). The decision boundaries become complex when considering nonlinear objectives. For example, Fig. 14d shows the case where the objective is to minimize the **cost**, which, defined by (66), is nonlinearly related to both **capacity** and **number of missions**. When considering multi-objective problems, such as minimizing jointly  $\langle \text{mass}, \text{cost} \rangle$  (Fig. 14h) or  $\langle \text{mass}, \text{cost}, \text{maintenance} \rangle$  (Fig. 14h), there are multiple non-dominated solutions.

## From component to system co-design

The rest of the section reuses the battery DP into a larger co-design problem that considers the co-design of actuation together with energetics for a drone (Fig. 12a). We will see that the decision boundaries change dramatically, which shows that the optimal choices for a component cannot be made in isolation from the system.

The functionality of the drone's subsystem considered (Fig. 12a) are parametrized by **endurance**, **number of missions**, **extra power** to be supplied, and **payload**. We model “actuation” as a design problem with functionality **lift [N]** and resources **cost**, **mass** and **power**, and we assume that power is a quadratic function of lift (Fig. 13). Any other monotone map could be used.



The co-design constraints that combine energetics and actuation are

$$\text{battery capacity} \geq \text{total power} \times \text{endurance}, \quad (69)$$

$$\text{total power} = \text{actuation power} + \text{extra power},$$

$$\text{weight} = \text{total mass} \times \text{gravity},$$

$$\text{actuation lift} \geq \text{weight},$$

$$\text{labor cost} = \text{cost per replacement} \times \text{battery maintenance},$$

$$\text{total cost} = \text{battery cost} + \text{actuation cost} + \text{labor cost},$$

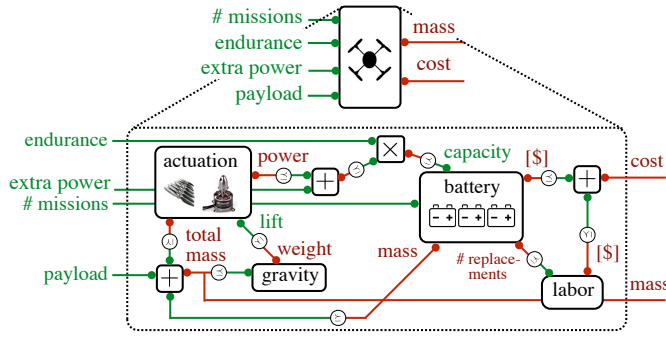
$$\text{total mass} = \text{battery mass} + \text{actuation mass} + \text{payload}. \quad (70)$$

The co-design graph contains recursive constraints: the power for actuation depends on the total weight, which depends on the mass of the battery, which depends on the capacity to be provided, which depends on the power for actuation. The MCDPL code for this model is shown in Fig. 12b; it refers to the previously defined models for “batteries” and “actuation”.

The co-design problem is now complex enough that we can appreciate the compositional properties of CDPs to perform a qualitative analysis. Looking at Fig. 12a, we know that there is a monotone relation between any pair of functionality and resources, such as **payload** and **cost**, or **endurance** and **mass**, even without knowing exactly what are the models for battery and actuation.

When fully expanded, the co-design graph (too large to display) contains 110 nodes and 110 edges. It is possible to remove all cycles by removing only one edge (e.g., the **energy**  $\leq$  **capacity** constraint), so the design complexity (Def. 37.25) is equal to  $\text{width}(\mathbb{R}_{\geq 0}) = 1$ . The tree representation is shown in Fig. 12c. Because the co-design diagram contains cycles, there is a loop operator at the root of the tree, which implies we need to solve a least fixed point problem. Because of the scale of the problem, it is not possible to show the map **h** explicitly, like we did in (41) for the previous example. The least fixed point sequence converges to 64 bits



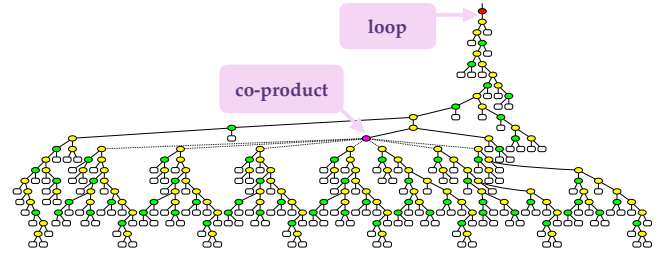


(a) Co-design diagram corresponding to Eqs. (69) to (70).

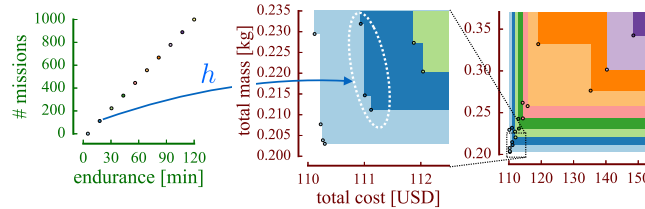
```

1 mcdp {
2   provides endurance [s]
3   provides extra_payload [kg]
4   provides extra_power [W]
5   provides num_missions [R]
6   provides velocity [m/s]
7
8   requires total_cost [$]
9   requires total_mass [g]
10
11   battery = instance `Batteries
12   actuation = instance `Actuation
13
14   total_power = extra_power +
15     power required by actuation
16
17   missions_provided_by_battery >= num_missions
18
19   energy = provided_endurance * total_power
20   capacity_provided_by_battery >= energy
21
22   total_mass = (
23     mass required by battery +
24     actuator_mass required by actuation +
25     extra_payload)
26
27   required_total_mass >= total_mass
28
29   gravity = 9.81 m/s^2
30   weight = total_mass * gravity
31
32   lift_provided_by_actuation >= weight
33   velocity_provided_by_actuation >= velocity
34
35   replacements = maintenance_required_by_battery
36   cost_per_replacement = 10 $
37   labor_cost = cost_per_replacement * replacements
38
39   required_total_cost >= (
40     cost required by actuation +
41     cost required by battery +
42     labor_cost)
43 }

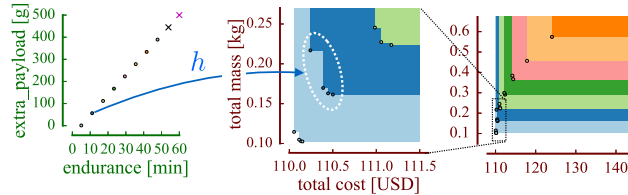
```



(b) MCDPL code for Eqs. (69) to (70). The “instance” statements refer to (c) Tree representation for the CDP. Yellow/green rounded ovals are previously defined models for batteries (Fig. 10b) and actuation (not shown). series/par junctions. There is one coproduct junction, signifying the choice between different battery technologies, and one loop junction, at the root of the tree.



(d) Relation between endurance and number of missions and cost and mass.



(e) Relation between endurance and payload and cost and mass.

**Figure 12.:** In panel (c), the payload is fixed to 100 g and extra power is set to 1 W. In panel (d), the number of missions is fixed to 400 and extra power is set to 1 W. The last two values, marked with “x”, are not feasible.

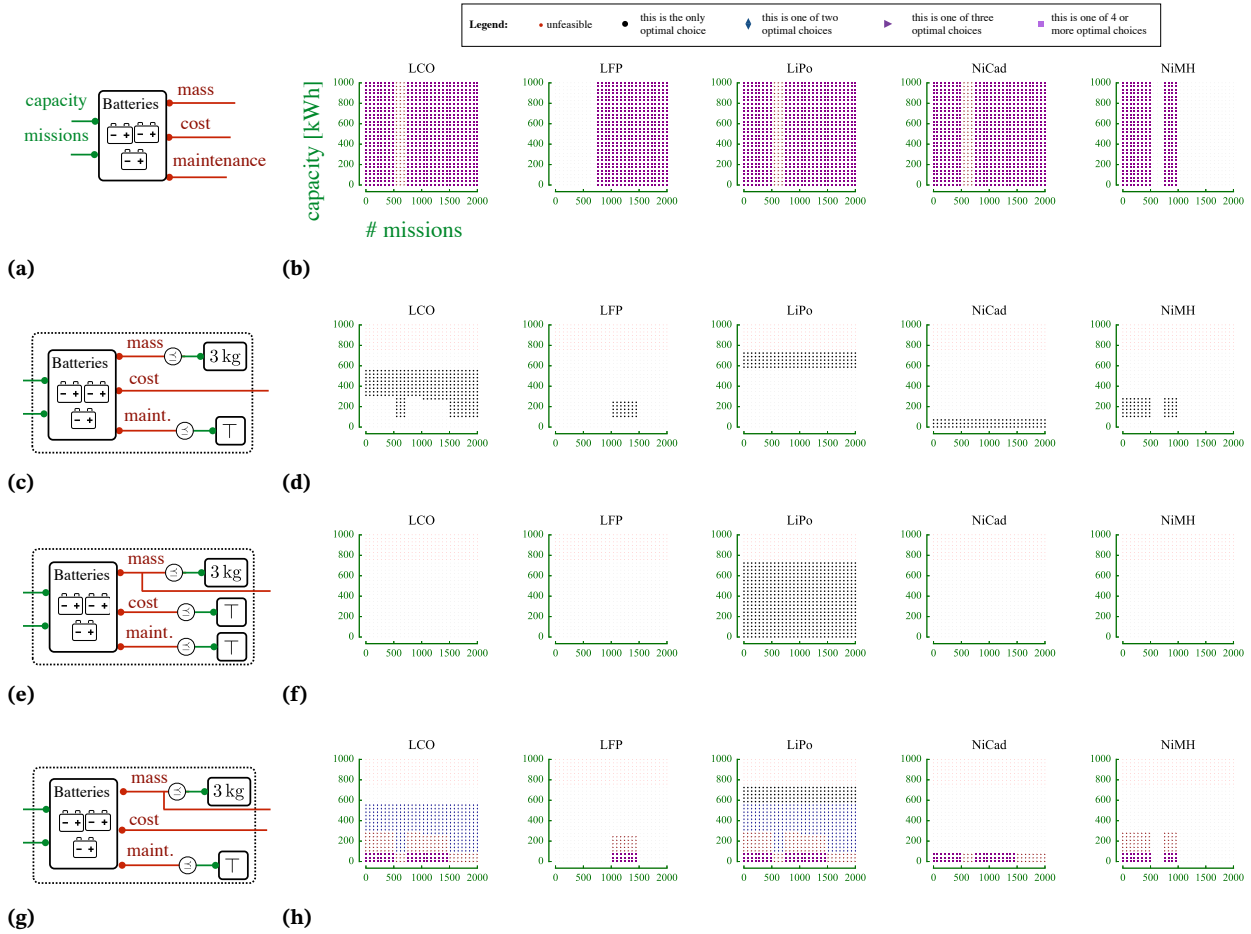
machine precision in 50–100 iterations.

To visualize the multidimensional relation

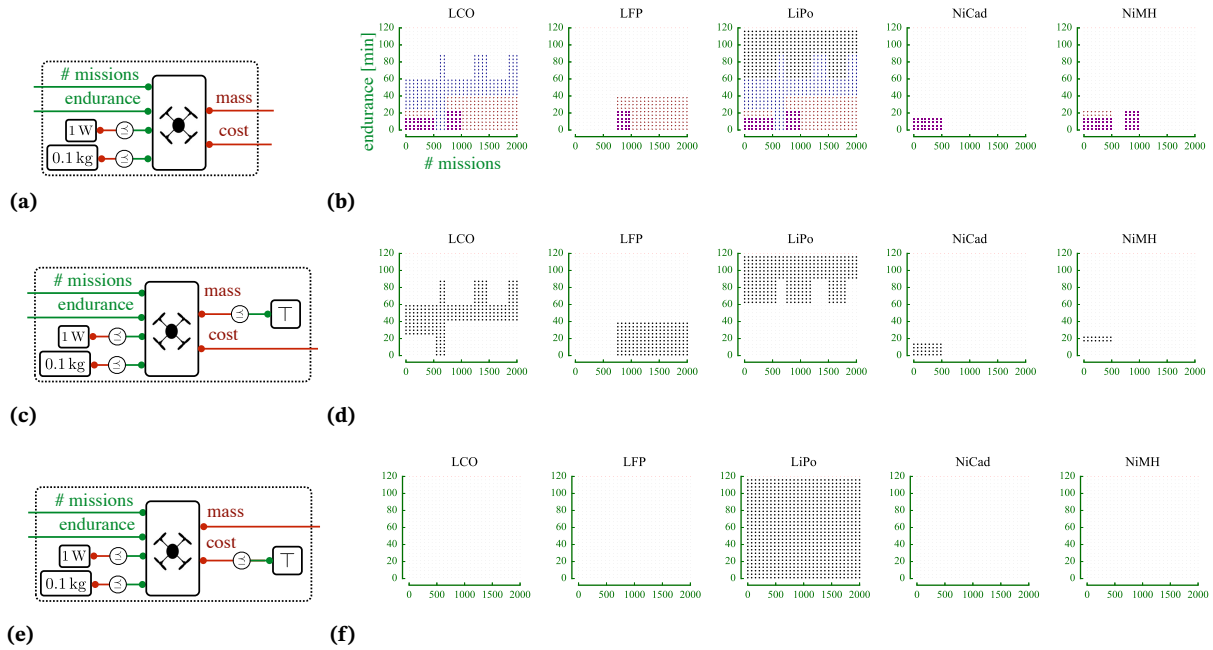
$$h: \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}^s \times \overline{\mathbb{R}}_{\geq 0}^W \times \overline{\mathbb{R}}_{\geq 0}^g \rightarrow \text{Anti}(\overline{\mathbb{R}}_{\geq 0}^{\text{kg}} \times \overline{\mathbb{R}}_{\geq 0}^{\text{USD}}), \quad (71)$$

we need to project across 2D slices. Fig. 12d shows the relation when the functionality varies in a chain in the space **endurance/missions**, and Fig. 12e shows the results for a chain in the space **endurance/payload**.

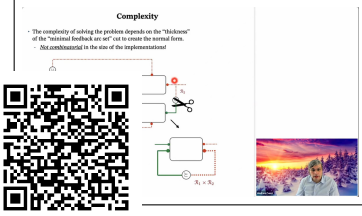
Finally, Fig. 15 shows the optimal choices of battery technologies in the **endurance/missions** space, when one wants to minimize **mass, cost**, or  $\langle \text{mass, cost} \rangle$ . The decision boundaries are completely different from those in Fig. 14. This shows that it is not possible to optimize a component separately from the rest of the system, if there are cycles in the co-design diagram.



**Figure 14.:** This figure shows the optimal decision boundaries for the different battery technologies for the design problem “batteries”, defined as the coproduct of all battery technologies (Fig. 10). Each row shows a different variation of the problem. The first row (panels *a–b*) shows the case where the objective function is the product of  $\langle \text{mass}, \text{cost}, \text{maintenance} \rangle$ . The shape of the symbols shows how many minimal solutions exists for a particular value of the functionality  $\langle \text{capacity}, \text{missions} \rangle$ . In this case, there are always three or more minimal solutions. The second row (panels *c–d*) shows the decision boundaries when minimizing only the scalar objective  $\text{cost}$ , with a hard constraint on  $\text{mass}$ . The hard constraints make some combinations of the functionality infeasible. Note how the decision boundaries are nonconvex, and how the formalism allows defining slight variations of the problem.



**Figure 15.:** This figure shows the decision boundaries for the different values of battery technologies for the integrated actuation-energetics model described in Fig. 12. Please see the caption of Fig. 14 for an explanation of the symbols. Notice how in most cases the decision boundaries are different from those in Fig. 14: this is an example in which one component cannot be optimized by itself without taking into account the rest of the system.

Watch: *Complexity* (7 minutes).

## 37.6. Complexity of the solution

### Complexity of fixed point iteration

Consider first the case of an DP that can be described as  $\mathbf{d} = \text{loop}(\mathbf{d}_0)$ , where  $\mathbf{d}_0$  is an DP that is described only using the series and par operators. Suppose that  $\mathbf{d}_0$  has resource space  $\mathbf{R}$ . Then evaluating  $h$  for  $\mathbf{d}$  is equivalent to computing a least fixed point iteration on the space of antichains  $\text{Anti } \mathbf{R}$ . This allows to give worst-case bounds on the number of iterations.

**Proposition 37.21.** Suppose that  $\mathbf{d} = \text{loop}(\mathbf{d}_0)$  and  $\mathbf{d}_0$  has resource space  $\mathbf{R}_0$  and evaluating  $h_0$  takes at most  $c$  computation. Then we can obtain the following bounds for the algorithm’s resources usage:

memory	$O(\text{width}(\mathbf{R}_0))$
number of steps	$O(\text{height}(\text{Anti } \mathbf{R}_0))$
total computation	$O(\text{width}(\mathbf{R}_0) \cdot \text{height}(\text{Anti } \mathbf{R}_0) \cdot c)$

*Proof.* The memory utilization is bounded by  $\text{width}(\mathbf{R}_0)$ , because the state is an antichain, and  $\text{width}(\mathbf{R}_0)$  is the size of the largest antichain. The iteration happens in the space  $\text{Anti } \mathbf{R}_0$ , and we are constructing an ascending chain, so it can take at most  $\text{height}(\text{Anti } \mathbf{R}_0)$  steps to converge. Finally, in the worst case the map  $h_0$  needs to be evaluated once for each element of the antichain for each step.  $\square$

These worst case bounds are strict.

**Example 37.22.** Consider solving  $\mathbf{d} = \text{loop}(\mathbf{d}_0)$  with  $\mathbf{d}_0$  defined by  $h_0 : \langle \langle \rangle, x \rangle \mapsto x + 1$  with  $x \in \overline{\mathbb{N}}$ . Then the least fixed point equation is equivalent to solving  $\min \{x : \Psi(x) \leq x\}$  with  $\Psi : x \mapsto x + 1$ . The iteration  $R_{k+1} = \Psi(R_k)$  converges to  $\top$  in  $\text{height}(\mathbb{N}) = \aleph_0$  steps.

**Remark 37.23.** Making more precise claims requires additional more restrictive assumptions on the spaces involved. For example, without adding a metric on  $\mathbf{R}$ , it is not possible to obtain properties such as linear or quadratic convergence.

**Remark 37.24** (Invariance to re-parameterization). All the results given in this paper are invariant to any order-preserving re-parameterization of all the variables involved.

### Relating complexity to the graph properties

Prop. 37.21 above assumes that the DP is already in the form  $\mathbf{d} = \text{loop}(\mathbf{d}_0)$ , and relates the complexity to the poset  $\mathbf{R}_0$ . Here we relate the results to the graph structure of an DP.

Take a DP  $\mathbf{d} = \langle \mathbb{F}, \mathbf{R}, \langle \mathcal{V}, \mathcal{E} \rangle \rangle$ . To put  $\mathbf{d}$  in the form  $\mathbf{d} = \text{loop}(\mathbf{d}_0)$  according to the procedure in Section 37.7, we need to find an arc feedback set (AFS) of the graph  $\langle \mathcal{V}, \mathcal{E} \rangle$ . Given an AFS  $F \subset \mathcal{E}$ , then the resource space  $\mathbf{R}$  for a  $\mathbf{d}_0$  such that  $\mathbf{d} = \text{loop}(\mathbf{d}_0)$  is the product of the resources spaces along the edges:  $\mathbf{R}_0 = \prod_{e \in F} \mathbf{R}_e$ .

Now that we have a relation between the AFS and the complexity of the iteration, it is natural to ask what is the optimal choice of AFS—which, so far, was left as an arbitrary choice. The AFS should be chosen as to minimize one of the performance measures in Prop. 37.21.

Of the three performance measures in Prop. 37.21, the most fundamental appears to be  $\text{width}(\mathbf{R}_0)$ , because that is also an upper bound on the number of distinct minimal solutions. Hence, we can call it “design complexity” of the DP.

**Definition 37.25** (Design complexity)

Given a graph  $\langle \mathcal{V}, \mathcal{E} \rangle$  and a labeling of each edge  $e \in \mathcal{E}$  with a poset  $\mathbf{R}_e$ , the *design complexity*  $\text{DC}(\langle \mathcal{V}, \mathcal{E} \rangle)$  is defined as

$$\text{DC}(\langle \mathcal{V}, \mathcal{E} \rangle) = \min_{F \text{ is an AFS}} \text{width}(\prod_{e \in F} \mathbf{R}_e). \quad (72)$$

In general, the width and height of posets are not additive with respect to products; therefore, this problem does not reduce to any of the known variants of the minimum arc feedback set problem, in which each edge has a weight and the goal is to minimize the sum of the weights.

### Considering relations with infinite cardinality

This analysis shows the limitations of the simple solution presented so far: it is easy to produce examples for which  $\text{width}(\mathbf{R}_0)$  is infinite, so that one needs to represent a continuum of solutions.

**Example 37.26.** Suppose that the platform to be designed must travel a distance  $d$  [m], and we need to choose the endurance  $T$  [s] and the velocity  $v$  [m/s]. The relation among the quantities is  $d \leq T v$ . This is a design problem described by the map

$$\begin{aligned} h : \overline{\mathbb{R}}_{\geq 0} &\rightarrow \text{Anti } \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0}, \\ d &\mapsto \{ \langle T, v \rangle \in \overline{\mathbb{R}}_{\geq 0} \times \overline{\mathbb{R}}_{\geq 0} : d = T v \}. \end{aligned}$$

For each value of  $d$ , there is a continuum of solutions.

One approach to solving this problem would be to discretize the functionality  $\mathbf{F}$  and the resources  $\mathbf{R}$  by sampling and/or coarsening. However, sampling and coarsening makes it hard to maintain completeness and consistency.

One effective approach that we will develop later is to *approximate the design problem itself*, rather than the spaces  $\mathbf{F}, \mathbf{R}$ , which are left as possibly infinite. The basic idea is that an infinite antichain can be bounded from above and below by two antichains that have a finite number of points. This idea leads to an algorithm that, given a prescribed computation budget, can compute an inner and outer approximation to the solution antichain.

Watch: *Canonical form* (6 minutes).

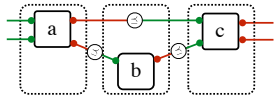


Figure 16.

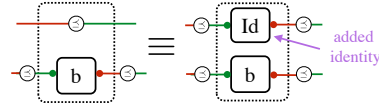


Figure 17.

## 37.7. Decomposition of CDPs

This section shows how to describe an arbitrary interconnection of design problems using only three composition operators. More precisely, for each CDPI with a set of atoms  $\mathcal{V}$ , there is an equivalent one that is built from series/par/loop applied to the set of atoms  $\mathcal{V}$  plus some extra “plumbing” (identities, multiplexers).

**Proposition 37.27.** Given a CDPI  $\langle \mathbf{F}, \mathbf{R}, \langle \mathcal{V}, \mathcal{E} \rangle \rangle$ , we can find an equivalent CDPI obtained by applying the operators par/series/loop to a set of atoms  $\mathcal{V}'$  that contains  $\mathcal{V}$  plus a set of trivial DPIs. Furthermore, one instance of loop is sufficient.

*Proof.* We show this constructively. We will temporarily remove all cycles from the graph, to be reattached later. To do this, find an *arc feedback set* (AFS)  $F \subseteq \mathcal{E}$ . An AFS is a set of edges that, when removed, remove all cycles from the graph (see [8]). For example, the CDPI represented in Fig. 18a has a minimal AFS that contains the edge  $c \rightarrow a$  (Fig. 18b).

Remove the AFS  $F$  from  $\mathcal{E}$  to obtain the reduced edge set  $\mathcal{E}' = \mathcal{E} \setminus F$ . The resulting graph  $\langle \mathcal{V}, \mathcal{E}' \rangle$  does not have cycles, and can be written as a series-parallel graph, by applying the operators par and series from a set of nodes  $\mathcal{V}'$ . The nodes  $\mathcal{V}'$  will contain  $\mathcal{V}$ , plus some extra “connectors” that are trivial DPIs.

Find a weak topological ordering of  $\mathcal{V}$ . Then the graph  $\langle \mathcal{V}, \mathcal{E}' \rangle$  can be written as the series of  $|\mathcal{V}|$  subgraphs, each containing one node of  $\mathcal{V}$ . In the example, the weak topological ordering is  $\langle a, b, c \rangle$  and there are three subgraphs (Fig. 16).

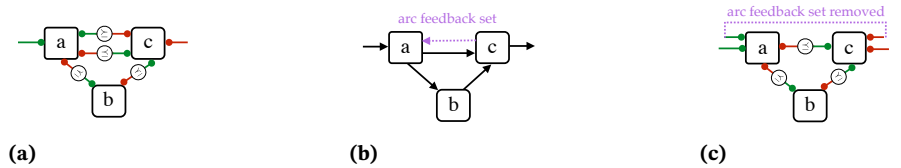
Each subgraph can be described as the parallel interconnection of a node  $v \in \mathcal{V}$  and some extra connectors. For example, the second subgraph in the graph can be written as the parallel interconnection of node  $b$  and the identity  $\text{Triv}(\text{id})$  (Fig. 17).

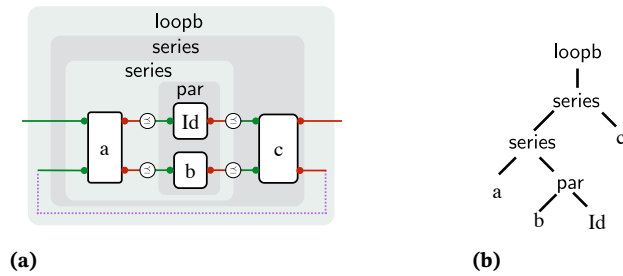
After this is done, we just need to “close the loop” around the edges in the AFS  $F$  to obtain a CDPI that is equivalent to the original one. Suppose the AFS  $F$  contains only one edge. Then one instance of the loopb operator is sufficient (Fig. 19a). In this example, the tree representation (Fig. 19b) is

$$\text{loopb}(\text{series}(\text{series}(a, \text{par}(\text{id}, b)), c)). \quad (73)$$

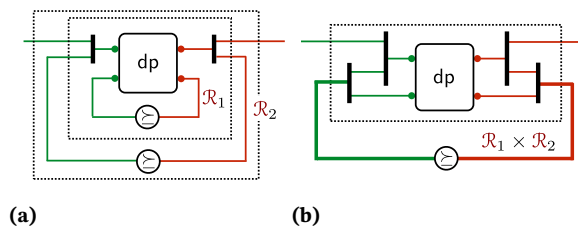
If the AFS contains multiple edges, then, instead of closing one loop at a time, we can always rewrite multiple nested loops as only one loop by taking the product of the edges. For example, a diagram like the one in Fig. 20a can be rewritten as Fig. 20b. This construction is analogous to the construction used for the analysis of process networks [15] (and any other construct involving a traced monoidal category). Therefore, it is possible to describe an arbitrary graph of design problems using only one instance of the loop operator.  $\square$

**Figure 18.:** An example co-design diagram with three nodes  $\mathcal{V} = \{a, b, c\}$ , in which a minimal arc feedback set is  $\{c \rightarrow a\}$ .





**Figure 19.:** Tree representation for the co-design diagram in Fig. 18a.



**Figure 20.:** If there are nested loops in a co-design diagram, they can be rewritten as one loop, by taking the product of the edges.







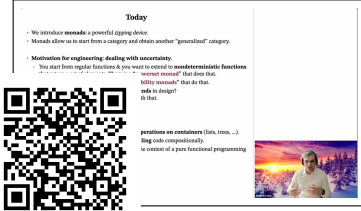
# 38. Monads

In this chapter, we introduce the concept of monad. We introduce monads using the computer science perspective, in which they are interpreted as “generalized computation”.

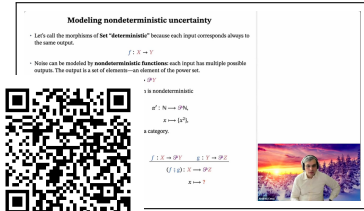
- 38.1 Generalized objects and operations574
- 38.2 Monads . . . . . 579
- 38.3 The Kleisli construction . . . . . 583
- 38.4 Algebras of a monad . . . . . 585
- 38.5 Monads from Adjunctions . . . . 589

When one thinks about Switzerland and the alps, one usually thinks about snow. Starting from 1,500 m height, most areas are covered in snow during winter (from December to March). Some areas (above 3,000 m), are *always* covered in snow.

Watch: *Introduction to monads* (6 minutes).



Watch: *Modeling nondeterministic uncertainty* (5 minutes).



## 38.1. Generalized objects and operations

The definition of monads is very powerful because it is very abstract and can fit many possible scenarios. Before getting to the formal definition, it is useful to build up some intuition using several examples.

Monads are a type of algebraic structure that is well-suited to represent generalized objects and operations. We begin by giving several examples.

### Modeling nondeterministic uncertainty

For the engineer, one intuitive scenario where generalization is necessary is handling uncertainty. We have seen the category **Set** of sets and functions between sets. A function

$$f: X \rightarrow Y \quad (1)$$

between sets is “deterministic”, in the sense that, given as input an element of  $X$ , it always produces the same output in  $Y$ .

Suppose now we want to deal with nondeterministic functions: functions that return, for each input, a *set* of possible values. We can model nondeterministic functions from  $X$  to  $Y$  as functions of the type

$$f: X \rightarrow \text{Pow } Y. \quad (2)$$

Note that this is a generalization, in the sense that any deterministic function is a special nondeterministic function. For example, the function

$$\begin{aligned} \alpha: \mathbb{N} &\rightarrow \mathbb{N}, \\ x &\mapsto x^2, \end{aligned} \quad (3)$$

can be rewritten as the function

$$\begin{aligned} \alpha': \mathbb{N} &\rightarrow \text{Pow } \mathbb{N}, \\ x &\mapsto \{x^2\}, \end{aligned} \quad (4)$$

which maps each element to a singleton set.

Once we have these generalized functions, we really want them to form a category, so that they can compose. To do this, we need extra information: additional structure. So far we know the rules of composition for functions:

$$\begin{array}{c} f: X \rightarrow Y \quad g: Y \rightarrow Z \\ \hline (f \circ g): X \rightarrow Z, \\ x \mapsto g(f(x)), \end{array} \quad (5)$$

but this does not help for the generalized functions. How can we compose non-deterministic function? What to fill in the space below?

$$\begin{array}{c} f: X \rightarrow \text{Pow } Y \quad g: Y \rightarrow \text{Pow } Z \\ \hline (f \circ g): X \rightarrow \text{Pow } Z, \\ x \mapsto ?. \end{array} \quad (6)$$

There is one natural way to define this operation. Did you notice the following?

**Lemma 38.1.** Functions from  $X$  to  $\text{Pow } Y$  are in one-to-one correspondence with relations from  $X$  to  $Y$ .

Therefore, we expect that nondeterministic functions can be composed like rela-

tions. If we have  $f : X \rightarrow \mathbf{Pow} Y$  and  $g : Y \rightarrow \mathbf{Pow} Z$  then we define the composite non-deterministic function to be

$$(f \circ g) : X \rightarrow \mathbf{Pow} Z, \\ x \mapsto \bigcup_{y \in f(x)} g(y). \quad (7)$$

Figure 1.

We should also check that this composition is associative; however, this comes automatically from the fact that we already know that **Rel** is a category.

To summarize:

- ▷ We wanted to extend **Set** from functions  $X \rightarrow Y$  to nondeterministic functions of type  $X \rightarrow \mathbf{Pow} Y$ .
- ▷ To do this, we needed three things:
  1. The particular choice of **Pow** as what maps a set to another set.
  2. A way to lift a function of type  $X \rightarrow Y$  to a function of type  $X \rightarrow \mathbf{Pow} Y$ , so that we can say that nondeterministic functions are a generalization of deterministic functions. This lifting operation is a family of functions of type

$$\mathbf{lift}_{X,Y} : (X \rightarrow Y) \rightarrow (X \rightarrow \mathbf{Pow} Y). \quad (8)$$

3. A way to define composition, through a map, traditionally called “fish”, of type

$$\mathbf{fish}_{X,Y,Z} : (X \rightarrow \mathbf{Pow} Y) \times (Y \rightarrow \mathbf{Pow} Z) \rightarrow (X \rightarrow \mathbf{Pow} Z) \quad (9)$$

- ▷ And we needed these pieces to satisfy the conditions:
  1. **fish** is associative. This ensures the generalized functions form a semicategory.
  2. The composition of the lifted functions are the lifting of the composition:

$$\frac{f : X \rightarrow \mathbf{Pow} Y \quad g : Y \rightarrow \mathbf{Pow} Z}{\mathbf{lift}_{X,Y}(f) \circ \mathbf{lift}_{Y,Z}(g) = \mathbf{lift}_{X,Z}(f \circ g)} . \quad (10)$$

This ensures that inside the generalized functions, the composition of regular functions continues to work as it should.

## Modeling interval uncertainty

We continue to build intuition considering another type of uncertainty.

We have seen the category **Pos** of posets and monotone functions between posets (Def. 14.8).

In this category it is easy to propagate uncertainty if the uncertain sets are represented by intervals.

Recall that for a poset **P**, we can define the poset of intervals **Arr P** (see Def. 10.10) and **Tw P** (see Def. 10.9).

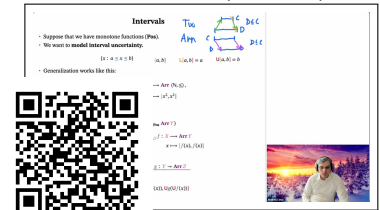
Analogously to the previous case, here we want to generalize from monotone functions to nondeterministic monotone functions, where the uncertainty is represented by intervals.

We recall the notation for intervals:

- ▷ The interval

$$\{x : a \leq x \leq b\} \quad (11)$$

Watch: *Intervals* (6 minutes).



is denoted

$$[a, b]. \quad (12)$$

▷  $\mathbf{L} \cdot$  and  $\mathbf{U} \cdot$  extract the lower and upper bound from the interval, so that we have

$$\mathbf{L}[a, b] = a \quad (13)$$

$$\mathbf{U}[a, b] = b \quad (14)$$

Because an interval is defined by two values, a function that returns an interval is a pair of functions, whose results are constrained to be ordered. For example, one such function is

$$\begin{aligned} f : \mathbb{R} &\rightarrow \mathbf{Arr} \langle \mathbb{R}, \leq \rangle, \\ x &\mapsto [x - 1, x + 1]. \end{aligned} \quad (15)$$

In the example above, we always have  $x - 1 \leq x + 1$ .

We now retrace the steps of the previous example.

First, we need to ensure that we can see regular monotone functions as special cases of interval functions. For example, the function

$$\begin{aligned} \alpha : \langle \mathbb{N}, \leq \rangle &\rightarrow \langle \mathbb{N}, \leq \rangle, \\ x &\mapsto x^2, \end{aligned} \quad (16)$$

can be rewritten as the function

$$\begin{aligned} \alpha' : \langle \mathbb{N}, \leq \rangle &\rightarrow \mathbf{Arr} \langle \mathbb{N}, \leq \rangle, \\ x &\mapsto [x^2, x^2], \end{aligned} \quad (17)$$

Generically, this is the definition of the **lift** function

$$\begin{aligned} \mathbf{lift}_{X,Y} : (X \rightarrow_{\mathbf{Pos}} Y) &\rightarrow (X \rightarrow_{\mathbf{Pos}} \mathbf{Arr} Y), \\ f &\mapsto \left\{ \begin{array}{l} \mathbf{lift}_{X,Y} f : X \rightarrow \mathbf{Arr} Y \\ x \mapsto [\mathbf{L}f(x), \mathbf{U}f(x)] \end{array} \right. . \end{aligned} \quad (18)$$

What is the “fish” function? Note that an interval-valued monotone map is also a special relation. Therefore, we want that composition continues to work in the same manner.

Therefore, we obtain for the **fish** operation:

$$\frac{f : X \rightarrow \mathbf{Arr} Y \quad g : Y \rightarrow \mathbf{Arr} Z}{(f \circ g) : X \rightarrow \mathbf{Arr} Z} \quad (19)$$

$$x \mapsto [\mathbf{L}g(\mathbf{L}f(x)), \mathbf{U}g(\mathbf{U}f(x))]$$

## Upper sets

We have seen that to solve DPs we encountered functions of type

$$f : \mathbf{F} \rightarrow_{\mathbf{Pos}} \mathbf{UR} \quad (20)$$

We can see this as another example of generalization from a function

$$f : X \rightarrow_{\mathbf{Pos}} Y \quad (21)$$

to functions

$$f : X \rightarrow_{\mathbf{Pos}} \mathbf{UY} \quad (22)$$

Once again we see these as particular types of relations.

The **lift** operation is

$$\begin{aligned} \text{lift}_{X,Y} : (X \rightarrow \text{Pos } Y) &\rightarrow (X \rightarrow \text{Pos } UY) \\ f &\mapsto \left\{ \begin{array}{l} \text{lift}_{X,Y} f : X \rightarrow UY \\ x \mapsto \uparrow f(x) \end{array} \right. \end{aligned} \quad (23)$$

The **fish** operation is

$$\begin{aligned} &\frac{f : X \rightarrow UY \quad g : Y \rightarrow UZ}{(f \circ g) : X \rightarrow UZ} \\ x &\mapsto \bigcup_{y \in f(x)} g(y) \end{aligned} \quad (24)$$

Note that the expression is the same as in (7) - only we are guaranteed to obtain upper sets.

## Keeping track of resource usage

As another example, we consider the case where we want to attach additional information to a category.

Suppose we want to consider not functions, but *procedures* which have resource consumption. A function is a mathematical entity - a procedure is a program that *implements* a function. Suppose we want to model execution time - and, that execution time might depend on the input of the procedure.

In this case, we would like to extend a function

$$f : X \rightarrow Y \quad (25)$$

into a procedure

$$f : X \rightarrow (Y \times \mathbb{R}_{\geq 0}) \quad (26)$$

which gives both result and the execution time.

We consider the ideal functions to be procedures that have zero execution time.

We can define the **lift** map as follows:

$$\begin{aligned} \text{lift}_{X,Y} : (X \rightarrow Y) &\rightarrow (X \rightarrow Y \times \mathbb{R}_{\geq 0}) \\ f &\mapsto \left\{ \begin{array}{l} \text{lift}_{X,Y} f : X \rightarrow Y \times \mathbb{R}_{\geq 0} \\ x \mapsto \langle x, 0 \rangle \end{array} \right. \end{aligned} \quad (27)$$

As for the fish function, we have

$$\begin{aligned} &\frac{f : X \rightarrow (Y \times \mathbb{R}_{\geq 0}) \quad g : Y \rightarrow (Z \times \mathbb{R}_{\geq 0})}{(f \circ g) : X \rightarrow (Z \times \mathbb{R}_{\geq 0})} \\ x &\mapsto \langle g_1(f_1(x)), g_2(f_1(x)) + f_2(x) \rangle \end{aligned} \quad (28)$$

Note that we can recover the naked function by taking the first component of the tuple.

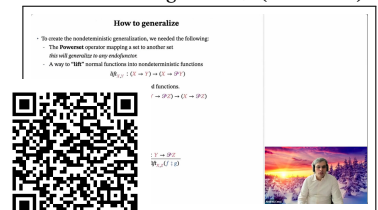
## Generalization with monoid

This can be further generalized from the real numbers to an arbitrary monoid structure

$$\mathbf{M} = \langle \mathbf{M}, \circ, \text{id}_{\mathbf{M}} \rangle. \quad (29)$$

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

Watch: *How to generalize* (3 minutes).



For the  $\text{lift}$  function we have

$$\begin{aligned} \text{lift}_{X,Y} : (X \rightarrow Y) &\rightarrow (X \rightarrow Y \times \mathbf{M}), \\ f &\mapsto \left\{ \begin{array}{l} \text{lift}_{X,Y} f : X \rightarrow Y \times \mathbf{M} \\ x \mapsto \langle x, \text{id}_{\mathbf{M}} \rangle \end{array} \right. . \end{aligned} \quad (30)$$

As for the fish function we have

$$\begin{array}{c} f : X \rightarrow (Y \times \mathbf{M}) \quad g : Y \rightarrow (Z \times \mathbf{M}) \\ \hline (f \mathbin{\circ} g) : X \rightarrow (Z \times \mathbf{M}) \\ x \mapsto \langle g_1(f_1(x)), g_2(f_1(x)) \mathbin{\circ}_{\mathbf{M}} f_2(x) \rangle . \end{array} \quad (31)$$

## 38.2. Monads

### Definition 38.2 (Monad)

Let  $\mathbf{C}$  be a category. A *monad* on  $\mathbf{C}$  is specified by:

#### Constituents

1. A functor  $M : \mathbf{C} \rightarrow \mathbf{C}$ ;
2. A natural transformation  $\mu : M \circ M \Rightarrow M$ , called the *composition* or *multiplication*;
3. A natural transformation  $\eta : \text{id}_{\mathbf{C}} \Rightarrow M$ , called the *unit*.

#### Conditions

1. *Associativity*: the following diagram must commute:

$$\begin{array}{ccc}
 M \circ M \circ M & \xrightarrow{M\mu} & M \circ M \\
 \downarrow \mu_M & & \downarrow \mu \\
 M \circ M & \xrightarrow{\mu} & M
 \end{array} \quad (32)$$

2. *Left and right unitality*: the following diagrams must commute:

$$\begin{array}{ccc}
 M & \xrightarrow{\eta_M} & M \circ M \\
 \searrow \text{id}_M & & \downarrow \mu \\
 & & M
 \end{array}
 \quad
 \begin{array}{ccc}
 M & \xrightarrow{M\eta} & M \circ M \\
 \searrow \text{id}_M & & \downarrow \mu \\
 & & M
 \end{array} \quad (33)$$

**Remark 38.3.** In terms of components, the unitality conditions state that for every object  $X \in \text{Ob}_{\mathbf{C}}$ , the following diagram commutes:

$$\begin{array}{ccc}
 M(X) & \xrightarrow{\eta_{MX}} & (M \circ M)(X) \\
 \searrow \text{id}_M & & \downarrow \mu_X \\
 & & M(X)
 \end{array}
 \quad
 \begin{array}{ccc}
 M(X) & \xrightarrow{M\eta_X} & (M \circ M)(X) \\
 \searrow \text{id}_M & & \downarrow \mu_X \\
 & & M(X)
 \end{array} \quad (34)$$

The associativity condition states that for every object  $X \in \text{Ob}_{\mathbf{C}}$ , the following diagram commutes:

$$\begin{array}{ccc}
 (M \circ M \circ M)(X) & \xrightarrow{M\mu_X} & (M \circ M)(X) \\
 \downarrow \mu_{M(X)} & & \downarrow \mu_X \\
 (M \circ M)(X) & \xrightarrow{\mu_X} & M(X)
 \end{array} \quad (35)$$

### Graded exercise K.1 (PowersetMonad)

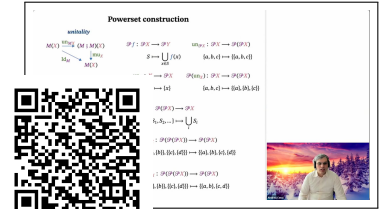
The aim of this exercise is to prove in full detail that the powerset functor  $\text{Pow} : \mathbf{Set} \rightarrow \mathbf{Set}$  is a monad, when equipped with the following *unit* and *multiplication*. We define  $\eta : \text{id}_{\mathbf{Set}} \Rightarrow \text{Pow}$  and  $\mu : \text{Pow} \circ \text{Pow} \Rightarrow \text{Pow}$  in terms of components: given an object  $A \in \mathbf{Set}$ , let

$$\begin{array}{ccc}
 \mu_A : (\text{Pow} \circ \text{Pow})(A) & \rightarrow & \text{Pow}(A) \\
 \text{D} & \mapsto & \bigcup_{S \in \text{D}} S
 \end{array} \quad (36)$$

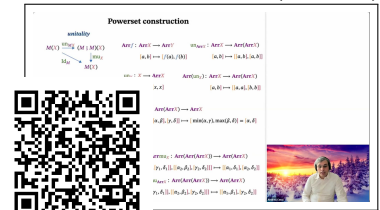
Watch: *Monad definition* (5 minutes).



Watch: *Powerset construction* (10 minutes).



Watch: *Interval construction* (5 minutes).





and

$$\begin{aligned} \text{un}_{\mathbf{A}} : \mathbf{A} &\rightarrow \text{Pow}(\mathbf{A}), \\ x &\mapsto \{x\}. \end{aligned} \quad (37)$$

To show that  $\langle \text{Pow}, \text{mu}, \text{un} \rangle$  is a monad,

1. prove that  $\text{un}$ , as defined in components in (37), is a natural transformation;
2. prove that  $\text{mu}$ , as defined in components in (36), is a natural transformation;
3. prove that  $\text{un}$  and  $\text{mu}$  satisfy the associativity condition and the left and right unitality conditions given in Def. 38.2. For this, work in components, as in Remark 38.3.

### Graded exercise K.2 ( $\text{FinProbMonad}$ )

Let  $\mathbf{Set}$  denote the category of sets, let  $\text{Id}_{\mathbf{Set}} : \mathbf{Set} \rightarrow \mathbf{Set}$  be the identity functor, and let  $\mathcal{P} : \mathbf{Set} \rightarrow \mathbf{Set}$  be the functor defined as follows. Given a set  $\mathbf{A}$ , the set  $\mathcal{P}(\mathbf{A})$  is the set of finitely supported probability measures on  $\mathbf{A}$ . We recall that these are functions  $p : \mathbf{A} \rightarrow [0, 1]$  with only finitely-many non-zero values and such that these values sum to 1:

$$\sum_{x \in \mathbf{A}} p(x) = 1. \quad (38)$$

On morphisms,  $\mathcal{P}$  is defined thus: for any function  $f : \mathbf{A} \rightarrow \mathbf{B}$ , the function  $\mathcal{P}(f) : \mathcal{P}(\mathbf{A}) \rightarrow \mathcal{P}(\mathbf{B})$  is

$$\begin{aligned} \mathcal{P}(f)(p) : \mathbf{B} &\rightarrow [0, 1], \\ y &\mapsto \sum_{x \in f^{-1}(\{y\})} p(x). \end{aligned} \quad (39)$$

Your tasks:

1. Given a set  $\mathbf{A}$ , let  $\delta_{\mathbf{A}} : \mathbf{A} \rightarrow \mathcal{P}\mathbf{A}$  be the function

$$\delta_{\mathbf{A}}(x)(y) = \begin{cases} 1 & \text{if } y = x \\ 0 & \text{else.} \end{cases} \quad x, y \in \mathbf{A}. \quad (40)$$

Check that the collection of functions  $\{\delta_{\mathbf{A}}\}_{\mathbf{A} \in \text{Ob}_{\mathbf{Set}}}$  defines a natural transformation  $\delta : \text{Id}_{\mathbf{Set}} \Rightarrow \mathcal{P}$ .

2. Given a set  $\mathbf{A}$ , let  $\mu_{\mathbf{A}} : \mathcal{P}\mathcal{P}\mathbf{A} \rightarrow \mathcal{P}\mathbf{A}$  be the function defined by

$$\mu_{\mathbf{A}}(f)(x) = \sum_{p \in \mathcal{P}\mathbf{A}} f(p)p(x) \quad f \in \mathcal{P}\mathcal{P}\mathbf{A}, x \in \mathbf{A}. \quad (41)$$

Check that the collection of functions  $\{\mu_{\mathbf{A}}\}_{\mathbf{A} \in \text{Ob}_{\mathbf{Set}}}$  defines a natural transformation  $\mu : \mathcal{P}\mathcal{P} \Rightarrow \mathcal{P}$ .

3. Show that  $\langle \mathcal{P}, \mu, \delta \rangle$  defines a monad on the category  $\mathbf{Set}$ .

**Example: interval construction** In the following, we analyze the interval construction and show that it gives rise to a monad. Given  $\mathbf{Pos}$ , we define the action of the  $\mathbf{Arr}$  functor on objects (posets) as

$$\begin{aligned} \mathbf{Arr} : \text{Ob}_{\mathbf{Pos}} &\rightarrow \text{Ob}_{\mathbf{Pos}} \\ \mathbf{P} &\mapsto \mathbf{Arr} \mathbf{P} \end{aligned} \quad (42)$$

and on morphisms as

$$\begin{aligned} \text{Arr } f &: \text{Arr } P \rightarrow \text{Arr } Q \\ [a, b] &\mapsto [f(a), f(b)]. \end{aligned} \quad (43)$$

Furthermore, we define the multiplication as

$$\begin{aligned} \text{mu}_X &: \text{Arr}(\text{Arr } X) \rightarrow \text{Arr } X \\ [[\alpha, \beta], [\gamma, \delta]] &\mapsto [\min(\alpha, \gamma), \max(\beta, \delta)] = [\alpha, \delta], \end{aligned} \quad (44)$$

and the unit as

$$\begin{aligned} \text{un}_X &: X \rightarrow \text{Arr } X \\ x &\mapsto [x, x]. \end{aligned} \quad (45)$$

First, we need to show that both the multiplication and the unit are indeed natural transformations. We start with multiplication. On the one hand, we have:

$$\begin{aligned} ((\text{Arr } \circ \text{Arr})(f) \circ \text{mu}_Y)([[a, b], [c, d]]) &= \text{mu}_Y([f(a), f(b)], [f(c), f(d)]) \\ &= [f(a), f(d)]. \end{aligned} \quad (46)$$

On the other hand we have:

$$\begin{aligned} (\text{mu}_X \circ \text{Arr})([[a, b], [c, d]]) &= (\text{Arr } f)([a, d]) \\ &= [f(a), f(d)]. \end{aligned} \quad (47)$$

The equivalence of (46) and (47) proves naturality. We now check that the unit defines a natural transformation. We have

$$\begin{aligned} (f \circ \text{un}_Y)(x) &= f(x) \circ \text{un}_Y \\ &= [f(x), f(x)] \end{aligned} \quad (48)$$

and

$$\begin{aligned} (\text{un}_X \circ \text{Arr } f)(x) &= (\text{Arr } f)([x, x]) \\ &= [f(x), f(x)] \end{aligned} \quad (49)$$

Again, equivalence of (48) and (49) proves naturality.

We now need to check associativity and left and right unitality. We start with associativity. To clearly see the property holding, we need to define intermediate quantities. We have

$$\begin{aligned} \text{Arr mu}_X &: \text{Arr}(\text{Arr}(\text{Arr}(X))) \rightarrow \text{Arr}(\text{Arr } X) \\ [[[\alpha_1, \beta_1], [\gamma_1, \delta_1]], [\alpha_2, \beta_2], [\gamma_2, \delta_2]] &\mapsto [[\alpha_1, \delta_1], [\alpha_2, \delta_2]] \end{aligned} \quad (50)$$

and

$$\begin{aligned} \text{mu}_{\text{Arr } X} &: \text{Arr}(\text{Arr}(\text{Arr } X)) \rightarrow \text{Arr}(\text{Arr } X), \\ [[[\alpha_1, \beta_1], [\gamma_1, \delta_1]], [\alpha_2, \beta_2], [\gamma_2, \delta_2]] &\mapsto [[\alpha_1, \beta_1], [\gamma_2, \delta_2]]. \end{aligned} \quad (51)$$

With this in mind, it is easy to see that the following diagram commutes.

For left and right unitality, we need to work out two additional quantities:

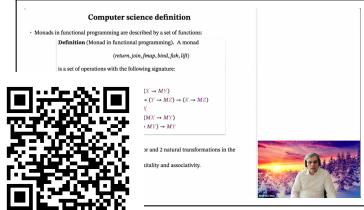
$$\begin{aligned} \text{un}_{\text{Arr } X} &: \text{Arr } X \rightarrow \text{Arr } \text{Arr } X \\ [a, b] &\mapsto [[a, b], [a, b]], \end{aligned} \quad (52)$$

and

$$\begin{aligned} \mathbf{Arr}(\mathbf{un}_X) : \mathbf{Arr} X &\rightarrow \mathbf{Arr}(\mathbf{Arr} X) \\ [a, b] &\mapsto [[a, a], [b, b]]. \end{aligned} \quad (53)$$

## Monads, computer science definition

Watch: *Computer science definition* (2 minutes).



For reference, we give the definition of a monad in functional programming, as a set of operations with particular types.

### Definition 38.4 (Monad in functional programming)

A monad

$$\langle \mathbf{return}, \mathbf{join}, \mathbf{fmap}, \mathbf{bind}, \mathbf{fish}, \mathbf{lift} \rangle \quad (54)$$

is a set of operations with the following signature:

$$\mathbf{return} : X \rightarrow MX \quad (55)$$

$$\mathbf{lift} : (X \rightarrow Y) \rightarrow (X \rightarrow MY) \quad (56)$$

$$\mathbf{fish} : (X \rightarrow MY) \rightarrow (Y \rightarrow MZ) \rightarrow (X \rightarrow MZ) \quad (57)$$

$$\mathbf{join} : MMX \rightarrow MX \quad (58)$$

$$\mathbf{fmap} : (X \rightarrow Y) \rightarrow (MX \rightarrow MY) \quad (59)$$

$$\mathbf{bind} : MX \rightarrow (X \rightarrow MY) \rightarrow MY \quad (60)$$

These maps satisfy the equivalent axioms of unitality and associativity:

- ▷ **return** is a left identity for **bind**;
- ▷ **return** is a right identity for **bind**;
- ▷ **bind** is associative.

### 38.3. The Kleisli construction

We return now to the discussion from the opening section of this chapter, in order to spell out further the relationship to monads.

#### Definition 38.5 (Kleisli morphisms)

Let  $\langle M, \text{mu}, \text{un} \rangle$  be a monad on a category  $\mathbf{C}$ , and let  $X, Y \in \text{Ob}_{\mathbf{C}}$ . A *Kleisli morphism*  $X \rightarrow Y$  is a morphism of  $\mathbf{C}$  of the form  $X \rightarrow MY$ .

#### Definition 38.6 (Kleisli composition)

Let  $\langle M, \text{mu}, \text{un} \rangle$  be a monad on a category  $\mathbf{C}$ , let  $X, Y, Z \in \text{Ob}_{\mathbf{C}}$ , and let  $f : X \rightarrow MY$  and  $g : Y \rightarrow MZ$  be morphisms in  $\mathbf{C}$  (so, they are Kleisli morphisms). Their *Kleisli composition* is the morphism in  $\mathbf{C}$  given by the composition

$$X \xrightarrow{f} M(Y) \xrightarrow{Mg} (M \circ M)(Z) \xrightarrow{\text{mu}_Z} M(Z). \quad (61)$$

#### Definition 38.7 (Kleisli category)

Let  $\langle M, \text{mu}, \text{un} \rangle$  be a monad on a category  $\mathbf{C}$ . The *Kleisli category*  $\mathbf{C}_M$  of the monad  $M$  is specified by:

1. *Objects*:  $\text{Ob}(\mathbf{C}_M) := \text{Ob}(\mathbf{C})$ ;
2. *Morphisms*:  $\text{Hom}_{\mathbf{C}_M}(X, Y) := \text{Hom}_{\mathbf{C}}(X, M(Y))$ ;
3. *Identities*:  $\text{id}_X := \text{un}_X$ ;
4. *Composition*: Kleisli composition.

#### Graded exercise K.3 (HwkRelKleisli)

Let  $\mathbf{Set}$  denote the category of sets and  $\text{Pow} : \mathbf{Set} \rightarrow \mathbf{Set}$  the powerset functor which assigns, to any given set  $A$ , the set of subsets of  $A$ . The endofunctor  $\text{Pow}$  may be equipped with the structure of a monad  $\langle \text{Pow}, \text{mu}, \text{un} \rangle$ , where the components of the natural transformations  $\text{mu}$  and  $\text{un}$  are given, respectively, by the functions

$$\begin{aligned} \text{mu}_A : \text{Pow Pow } A &\rightarrow \text{Pow } A \\ \mathbf{C} &\mapsto \bigcup_{B \in \mathbf{C}} B \end{aligned} \quad (62)$$

and

$$\begin{aligned} \text{un}_A : A &\rightarrow \text{Pow } A, \\ x &\mapsto \{x\}. \end{aligned} \quad (63)$$

Let  $\mathbf{Set}_{\text{Pow}}$  denote the Kleisli category of the monad  $\langle \text{Pow}, \text{mu}, \text{un} \rangle$ . The aim of this exercise is to show that the category  $\mathbf{Set}_{\text{Pow}}$  and the category  $\mathbf{Rel}$  of sets and relations are isomorphic as categories.

We define functors  $F : \mathbf{Set}_{\text{Pow}} \rightarrow \mathbf{Rel}$  and  $G : \mathbf{Rel} \rightarrow \mathbf{Set}_{\text{Pow}}$  as follows. On objects  $F$  is the identity function, and given a function  $f : A \rightarrow_{\mathbf{Set}} \text{Pow } B$  (in other words, a Kleisli morphism  $f : A \rightarrow_{\mathbf{Set}_{\text{Pow}}} B$ ) we let  $F(f)$  be the relation

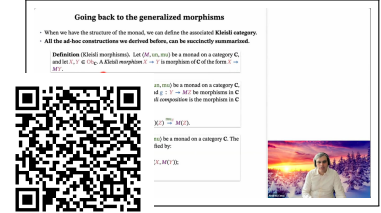
$$F(f) = \{\langle x, y \rangle \in A \times B \mid y \in f(x)\}. \quad (64)$$

The functor  $G$  is also defined to be the identity function on objects, and given a relation  $R : A \rightarrow B$ , we let  $G(R)$  be the Kleisli morphisms represented by the following function:

$$G(R) : A \rightarrow_{\mathbf{Set}} \text{Pow } B, x \mapsto \{y \in B \mid \langle x, y \rangle \in R\}. \quad (65)$$

Your tasks:

Watch: *Generalized morphisms* (3 minutes).



1. Prove that  $F : \mathbf{Set}_{\mathbf{Pow}} \rightarrow \mathbf{Rel}$  and  $G : \mathbf{Rel} \rightarrow \mathbf{Set}_{\mathbf{Pow}}$  are in fact functors.
2. Prove that  $F$  is an isomorphism of categories, with inverse  $G$ .



pressions to be those with three-layers of square brackets, fourth-order formal expressions have four layers of brackets, and so on. In the following, we'll probably only ever consider up to three layers.

We started our story just with the set  $\mathbf{A} = \{\text{🍎}, \text{🍌}, \text{🥕}\}$ . However, we can do the same construction – using “\*” and building formal expressions of any order – with any set. In fact, we can define a functor  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  which maps any set  $\mathbf{A}$  to the set  $F\mathbf{A}$  whose elements are all finite first-order formal expressions built from elements of  $\mathbf{A}$ . We also include this to mean the empty formal expression “[].” What might this functor do on the level of morphisms? For concreteness, let  $\mathbf{A} = \{\text{🍎}, \text{🍌}, \text{🥕}\}$  and  $\mathbf{B} = \{\text{🍷}, \text{🍺}\}$ . Given a function  $f : \mathbf{A} \rightarrow \mathbf{B}$ , we define

$$F(f) : F(\mathbf{A}) \rightarrow F(\mathbf{B}) \quad (72)$$

to act on (first-order) expressions like so

$$F(f)([\text{🍎}] * [\text{🍌}]) = [f(\text{🍎})] * [f(\text{🍌})]. \quad (73)$$

It turns out that this functor  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  can be made into a monad!

Let us explain how the unit and multiplication for this monad are defined. For any set  $\mathbf{A}$ , the component of the unit at  $\mathbf{A}$  is

$$\begin{aligned} \text{un}_{\mathbf{A}} : \mathbf{A} &\rightarrow F\mathbf{A}, \\ x &\mapsto [x]. \end{aligned} \quad (74)$$

The multiplication is a bit more of a mouthful. Its component at  $\mathbf{A}$ ,

$$\text{mu}_{\mathbf{A}} : (F \circ F)\mathbf{A} \rightarrow F\mathbf{A}, \quad (75)$$

is the function which takes a second-order formal expression

$$[[x_{11}] * [x_{12}] * \cdots * [x_{1k_1}]] * \cdots * [[x_{n1}] * [x_{n2}] * \cdots * [x_{nk_n}]] \quad (76)$$

and “collapses” it to the first-order expression

$$[x_{11}] * [x_{12}] * \cdots * [x_{1k_1}] * \cdots * [x_{n1}] * [x_{n2}] * \cdots * [x_{nk_n}]. \quad (77)$$

In other words, this operation simply “removes the outer brackets” from a second-order formal expression.

#### Graded exercise K.4 (ListMonad)

Let  $F : \mathbf{Set} \rightarrow \mathbf{Set}$  be the functor above that sends any set  $\mathbf{A}$  to the set of first-order formal expressions of the form

$$[x_1] * [x_2] * \cdots * [x_n] \quad x_i \in \mathbf{A}, n \in \mathbb{Z}_{\geq 0}, \quad (78)$$

and let  $\text{mu}_{\mathbf{A}}$  and  $\text{un}_{\mathbf{A}}$  be defined as above.

Show that:

1. the components  $\text{mu}_{\mathbf{A}}$  define a natural transformation  $\text{mu} : F \circ F \Rightarrow F$ ;
2. the components  $\text{un}_{\mathbf{A}}$  define a natural transformation  $\text{un} : \text{Id}_{\mathbf{Set}} \Rightarrow F$ ;
3.  $\text{mu}$  and  $\text{un}$  satisfy the conditions for  $\langle F, \text{mu}, \text{un} \rangle$  to be a monad.

Now let's finally talk about giving formal expressions a computational meaning: a way to *evaluate* them. The way we will do this is to define a notion of “evaluation map”  $a : F\mathbf{A} \rightarrow \mathbf{A}$  which we will interpret as a way of specifying how any formal expression – an element of  $F\mathbf{A}$  – should be evaluated (or: computed) to an element of  $\mathbf{A}$ . We will require such evaluation maps to additionally satisfy two

coherence conditions, and the resulting mathematical structure will be what is called an *algebra* of the monad  $F$ .

### Definition 38.8 (Algebra of a monad)

Let  $\langle M, \mu, \eta \rangle$  be a monad on a category  $\mathbf{C}$ . An algebra of  $M$  (also called an  $M$ -algebra) is specified by:

#### Constituents

1. an object  $X$  of  $\mathbf{C}$ ;
2. a morphism  $a : M(X) \rightarrow X$  of  $\mathbf{C}$ .

#### Conditions

1. *Composition*: the following diagram commutes:

$$\begin{array}{ccc} (M \circ M)(X) & \xrightarrow{Ma} & M(X) \\ \mu_X \downarrow & & \downarrow a \\ M(X) & \xrightarrow{a} & X \end{array} \quad (79)$$

2. *Unit*: the following diagram commutes:

$$\begin{array}{ccc} X & \xrightarrow{\eta_X} & M(X) \\ & \searrow \text{id} & \downarrow a \\ & & X \end{array} \quad (80)$$

### Definition 38.9 ( $M$ -algebra morphism)

Let  $\langle M, \eta, \mu \rangle$  be a monad on a category  $\mathbf{C}$ , and let  $\langle X_1, a_1 \rangle$  and  $\langle X_2, a_2 \rangle$  be algebras of  $M$ . A morphism  $\langle X_1, a_1 \rangle \rightarrow \langle X_2, a_2 \rangle$  of  $M$ -algebras is specified by:

#### Constituents

1. A morphism  $f : X_1 \rightarrow X_2$  in  $\mathbf{C}$ .

#### Conditions

1. The following diagram commutes:

$$\begin{array}{ccc} M(X_1) & \xrightarrow{Mf} & M(X_2) \\ a_1 \downarrow & & \downarrow a_2 \\ X_1 & \xrightarrow{f} & X_2 \end{array} \quad (81)$$

### Definition 38.10 (Category of $M$ -algebras)

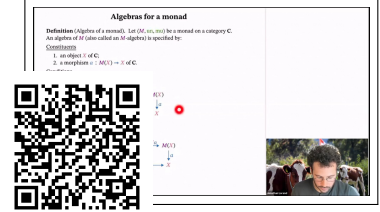
Let  $\langle M, \eta, \mu \rangle$  be a monad on a category  $\mathbf{C}$ . The *category of  $M$ -algebras*  $\mathbf{C}^M$  of the monad  $M$  is specified by:

1. *Objects*:  $M$ -algebras;
2. *Morphisms*:  $M$ -algebra morphisms;
3. *Identities*: given an  $M$ -algebra  $\langle X, a \rangle$ , its identity morphism is  $\text{id}_X$ ;
4. *Composition*: is induced by the composition of morphisms in  $\mathbf{C}$ .

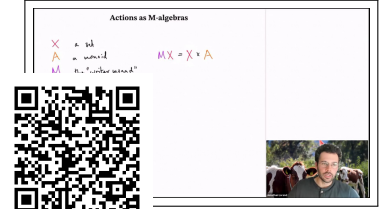
**Proposition 38.11.** There is an  $M$ -algebra structure on  $X$  if and only if there is a monoidal structure on  $X$ . The correspondence is as follows:

Something incorrect or unclear or missing? Report issues on GitHub by clicking here.

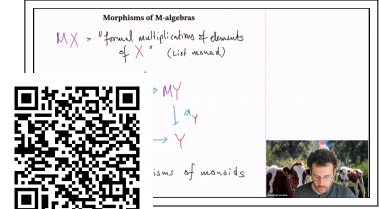
Watch: *Algebras for a monad* (2 minutes).



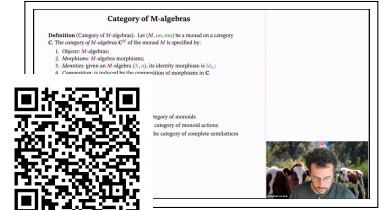
Watch: *Actions as  $M$ -algebras* (5 minutes).



Watch: *Morphisms of  $M$ -algebras* (3 minutes).



Watch: *Category of  $M$ -algebras* (2 minutes).





- ▷ The neutral element of  $X$  corresponds to  $a([])$ .
- ▷ The composition in  $X$  corresponds to the action of  $a$ :

$$[x_1] * [x_2] \xrightarrow{a} x_1 \circ x_2. \quad (82)$$

- ▷ The associativity and unitality of monoid composition are encoded by the  $M$ -algebra axioms.

$$X = \{0, 1, 2, 3, \dots\} \quad (83)$$

$$\begin{aligned} a : MX &\rightarrow X \\ [] &\mapsto 0 \\ [x] &\mapsto x \\ [x] * [y] &\mapsto x + y \end{aligned} \quad (84)$$

**Graded exercise K.5** (HwkFreeAlgebras)

Let  $\langle M, \text{mu}, \text{un} \rangle$  be a monad on a category  $\mathbf{C}$ , and let  $X$  be an object of  $\mathbf{C}$ . Your task is to prove that the object  $MX$ , together with the morphism

$$\text{mu}_X : MMX \rightarrow MX \quad (85)$$

defines an algebra for the monad  $M$ . *Hint:* use the axioms/conditions in the definition of a monad.

## 38.5. Monads from Adjunctions



## Solutions to selected exercises





When one thinks about Switzerland and the alps, one usually thinks about snow. Starting from 1,500 m height, most areas are covered in snow during winter (from December to March). Some areas (above 3,000 m), are *always* covered in snow.



# Nomenclature

symbol	meaning	defined in
<b>Generic symbols</b>		
<i>Booleans</i>		
$\top$	True	
$\perp$	False	
$\wedge$	Boolean and	
$\vee$	Boolean or	
$\Rightarrow$	Implies	
$\Leftrightarrow$	Equivalence	
<i>Categories</i>		
<i>Adjunctions</i>		
$L$	Left adjunt functor	$\rightarrow$ Def. 33.18 on p.488
$R$	Right adjunt functor	$\rightarrow$ Def. 33.18 on p.488
$L \dashv R$	$L$ and $R$ are adjoint functors.	$\rightarrow$ Def. 33.18 on p.488
$\tau$	Adjunction isomorphism	$\rightarrow$ Def. 33.18 on p.488
$un$	Unit	$\rightarrow$ Def. 33.20 on p.489
$co$	Co-unit	$\rightarrow$ Def. 33.20 on p.489
<i>Arrows</i>		
$\rightarrow$	Set arrow	
$\rightarrow$	Morphism arrow	
$\longrightarrow$	Morphism arrow	
$\leftarrow$	Morphism arrow	
$\rightarrowtail$	Functors arrow	
$\Rightarrow$	Natural transformation arrow	
$\Rrightarrow$		
$\dashrightarrow$	profunctor arrow	
<i>Companion/conjoints</i>		
<i>Composition</i>		
$f \circ g$	Composition of morphisms	
$\circ$	Composition for functors	
$\circ$	Composition for natural transformations	
$F \circ G$	Composition of functors	
$Ob_{\mathbf{A}}$	Objects of the category $\mathbf{A}$ .	$\rightarrow$ Def. 13.9 on p.216
$Mor_{\mathbf{A}}$	Collection of all morphisms of the category $\mathbf{A}$ .	$\rightarrow$ Def. 13.9 on p.216
$id_X$	Identity morphism for the object $X$	$\rightarrow$ Def. 13.9 on p.216
$Hom_{\mathbf{A}}(X; Y)$	Hom-set between $X$ and $Y$ .	$\rightarrow$ Def. 13.9 on p.216
<i>Constructors</i>		
<b>Free</b>	free construction	$\rightarrow$ Def. 13.18 on p.222
<b>Arr</b> $\mathbf{A}$	Arrow construction on category $\mathbf{A}$ .	
<b>Tw</b> $\mathbf{A}$	Twisted arrow construction on category $\mathbf{A}$ .	$\rightarrow$ Def. 17.10 on p.290
<i>Generic names</i>		
$f, g, h, i$	Generic morphisms	$\rightarrow$ Def. 13.9 on p.216
$F, G, H, K$	Generic functors	$\rightarrow$ Def. 19.2 on p.308
$id_{\mathbf{A}}$	Identity functor for category $\mathbf{A}$	



symbol	meaning	defined in
$\alpha, \beta, \gamma, \delta$	Generic natural transformations	→Def. 32.1 on p.465
$\text{id}_F$	Identity natural transformation for functor $F$	
<i>Monoidal categories</i>		
$\otimes$	Stacking category objects semigroup operation	
$\boxtimes$	Stacking category morphisms semigroup operation	
$\otimes_A$	Monoidal product for category $A$ .	→Def. 34.1 on p.496
$1$	Identity object for monoidal operation	→Def. 34.1 on p.496
$l_u$	Left unitor	→Def. 34.1 on p.496
$r_u$	Right unitor	→Def. 34.1 on p.496
$as$	Associator	→Def. 34.1 on p.496
$br$	Braiding	→Def. 34.1 on p.496
$\mu$	Isomorphism for strong monoidal functor	→Def. 34.12 on p.503
$iso$	Natural isomorphism for strong monoidal functor	→Def. 34.12 on p.503
$ev$	evaluation map for dualizable objects	
$coev$	coevaluation map for dualizable objects	
<i>Well-known categories</i>		
<b>DP</b>	Category of design problems	→Def. 26.13 on p.391
<b>Feas</b>	Synonym of <b>DP</b>	→Def. 26.13 on p.391
<b>UDP</b>		
<b>DPI</b>	Semi-category <b>DPI</b>	→Def. 25.1 on p.358
<b>Bool</b>	Posets/category of Booleans	
<b>Cat</b>	Category of small categories	→Def. 22.3 on p.329
<b>FinVect</b>	Category of finite-dimensional vector spaces	
<b>Vect<sub>R</sub></b>	Category of real vector spaces	→Def. 30.10 on p.434
<b>Rel</b>	Category of sets and relations	→Def. 14.1 on p.233
<b>FinSet</b>	Category of finite sets and functions	→Example 20.2 on p.318
<b>Set</b>	Category of sets and functions	→Def. 13.11 on p.217
<b>InjSet</b>	Category of sets and injective functions	→Example 20.5 on p.320
<b>Prof</b>	Category of profunctors	
<b>Pos</b>	Category of posets and monotone maps	→Def. 14.8 on p.236
<b>Lat</b>	Category of lattices and lattice homomorphisms	→Def. 27.18 on p.404
<b>BoundedLat</b>	Category of lattices and lattice homomorphisms	→Def. 27.19 on p.404
<b>Grph</b>	Category of directed graphs	→Def. 14.14 on p.238
<b>Eff</b>	The effects category	→Def. 30.10 on p.434
<b>EndSet</b>	Category of endofunctions	
<b>Set<sub>*</sub></b>	Category of pointed sets	
<b>EquivRel</b>	Category of equivalence relations	
<b>Euc<sub>*</sub></b>	Category of pointed Euclidean spaces	
<b>Grph</b>	Category of graphs	→Def. 12.2 on p.204
<b>LTI</b>		→Def. 16.26 on p.280
<b>Beh</b>		
<b>LBeh</b>		
$\langle \text{Rel} \rangle$	category of tuple-sets and relations	→Def. 30.14 on p.437
$\langle \text{Set} \rangle$		→Def. 16.3 on p.264
$\langle \text{Pos} \rangle$	category of tuple-posets and monotone maps	→Def. 30.12 on p.436
$\langle \text{DP} \rangle$	Category of lists of DPs	→Def. 30.16 on p.437
<i>Operations</i>		
$\times$	Product in a category	
$+$	Product in a category	
$+$	Disjoint union of posets	
$+$	Co-product in a category	

symbol	meaning	defined in
$\times$	Product in a category	
$+$	Product in a category	
<i>Traced monoidal categories</i>		
Tr	Trace operator	→Def. 34.20 on p.508
Conw	Conway operator	
<i>DP</i>		
<i>Computational representation</i>		
$k$		
$K$		
$h$		
$H$		
<i>Formalization</i>		
$f$	A generic functionality in $\mathbf{F}$ .	→Def. 25.1 on p.358
$r$	A generic cost in $\mathbf{R}$ .	
$i$	A generic implementation in $\mathbf{I}$ .	→Def. 25.1 on p.358
$\mathbf{F}$	Functionality space	
$\mathbf{R}$	Requirements space	→Def. 25.1 on p.358
$\mathbf{I}$	Implementation space	→Def. 25.1 on p.358
$\text{prov} : \mathbf{I} \rightarrow \mathbf{F}$	functionality of an implementation	→Def. 25.1 on p.358
$\text{req} : \mathbf{I} \rightarrow \mathbf{R}$	requirements of an implementation	→Def. 25.1 on p.358
<i>Queries in DP</i>		
Feasibility		→Section 25.3 on p.368
FeasibleImp		→Section 25.3 on p.367
FixFunMinRes		→Section 25.3 on p.367
FixFunFeasRes		
FixResMaxFun		
FixResFeasFun		→Section 25.3 on p.367
<i>DP</i>		
$\mathbf{d}, \mathbf{e}, \mathbf{g}$	Generic design problems as profunctors	
vdc	Van Der Corput sequence	
<i>Groups</i>		
id	identity for group	→Def. 6.36 on p.117
$\mathbf{G}, \mathbf{H}$	Generic group names	→Def. 6.36 on p.117
$\mathbf{G}, \mathbf{H}$	Underlying set of groups.	→Def. 6.17 on p.112
inv	Group inverse	→Def. 6.36 on p.117
$x, y, z$	Generic group elements	
$\%$	Group composition operation	→Def. 6.36 on p.117
<i>Linear Algebra</i>		
$\mathbf{0}$	Zero matrix	
$\mathbf{1}$	Identity matrix	
$\det A$	matrix determinant (as functor)	
$\mathcal{P}^+$	Positive-definite matrices	
<i>Monoids</i>		
id	identity for monoid	→Def. 6.17 on p.112
$\mathbf{M}, \mathbf{N}, \dots$	Generic monoid names	→Def. 6.17 on p.112
$\mathbf{M}, \mathbf{N}, \dots$	Underlying sets for the monoids.	→Def. 6.17 on p.112
$x, y, z$	Generic monoid elements	→Def. 6.17 on p.112
$m, n$	Generic monoid elements	→Def. 6.17 on p.112
$\%$	Semigroup/monoid/group operation	→Def. 6.17 on p.112

symbol	meaning	defined in
<i>Posets</i>		
<i>Attributes</i>		
$\text{width}(\mathbf{P})$	Width of the poset $\mathbf{P}$ .	→Def. 9.19 on p.171
$\text{height}(\mathbf{P})$	Height of the poset $\mathbf{P}$ .	→Def. 9.20 on p.171
<i>Poset constructors</i>		
$\times$	Product of posets	
$\mathbf{Arr} \mathbf{P}$	Arrow construction on poset $\mathbf{P}$ .	→Def. 10.10 on p.186
$\mathbf{Tw} \mathbf{P}$	Twisted arrow construction on poset $\mathbf{P}$ .	→Def. 10.9 on p.183
$\mathbb{E}$	expectedvalue	
$\mathbf{1}$	singleton poset	
$\perp$	global dualizing object	
$\mathbf{P}$	a polycategory	
$\mathbf{Q}$	a polycategory	
$\mathbf{\Gamma}$	a polycategory	
$\mathbf{\Gamma}$	a polycategory	
$\Delta$	a polycategory	
$\Delta$	a polycategory	
$\Lambda$	a polycategory	
$\Sigma$	a polycategory	
<i>Constructors</i>		
$\mathbf{Anti} \mathbf{P}$	The set of antichains of a poset $\mathbf{P}$ .	→Def. 9.14 on p.168
$\mathbf{Anti}_f \mathbf{P}$	The set of finite antichains of a poset $\mathbf{P}$ .	
$\mathbf{LP}$	$= \langle \mathbf{USets} \mathbf{P}, \subseteq \rangle$ Poset of upward-closed lowerset of $\mathbf{P}$ ordered by $\subseteq$ .	→Def. 24.11 on p.351
$\overline{\mathbf{UP}}$	$= \langle \overline{\mathbf{UpSets} \mathbf{P}}, \supseteq \rangle$ Poset of downward-closed uppersets of $\mathbf{P}$ ordered by $\supseteq$ .	
$\overline{\mathbf{U}}_f \mathbf{P}$	$\subseteq \overline{\mathbf{UP}}$ Poset of finitely-supported upper sets of a poset $\mathbf{P}$ .	→Def. 37.13 on p.552
<i>Domain theory</i>		
$\text{lfp}$	Least fixed point	
$\text{CPO}$	Complete partial order	→Def. 37.2 on p.548
$\text{DCPO}$	Directed-complete partial order	→Def. 37.2 on p.548
<i>Generic poset names</i>		
$\mathbf{P}, \mathbf{Q}, \mathbf{R}$	Generic posets	→Def. 9.3 on p.159
$\mathbf{P}, \mathbf{Q}, \mathbf{R}$	Underlying set for the posets	→Def. 9.3 on p.159
$\leq_{\mathbf{P}}$	Order on poset $\mathbf{P}$	
$\mathbf{Pow} \mathbf{A}$	Power poset of $\mathbf{A}$ .	→Def. 9.12 on p.166
<i>Monoidal posets</i>		
$\otimes$	Monoidal poset operation	
<i>Operations on elements</i>		
$x \vee y$	Join of two elements $x, y$	→Def. 27.8 on p.400
$x \wedge y$	Meet of two elements $x, y$	→Def. 27.8 on p.400
<i>Operations on sets</i>		
$\text{Min}_{\leq_{\mathbf{P}}} \mathbf{S}$	Minimal elements of the subset $\mathbf{S}$ .	→Def. 24.1 on p.346
$\text{Max}_{\leq_{\mathbf{P}}} \mathbf{S}$	Maximal elements of the subset $\mathbf{S}$ .	→Def. 24.2 on p.346
$\text{Inf}_{\leq_{\mathbf{P}}} \mathbf{S}$	Infimum element of the subset $\mathbf{S}$ .	→Def. 24.7 on p.348
$\text{Sup}_{\leq_{\mathbf{P}}} \mathbf{S}$	Supremum element of the subset $\mathbf{S}$ .	→Def. 24.3 on p.348
$\uparrow \mathbf{S}$	Upper closure of $\mathbf{S}$ .	→Def. 24.12 on p.353
$\downarrow \mathbf{S}$	Lower closure of $\mathbf{S}$ .	→Def. 24.16 on p.353

symbol	meaning	defined in
<i>Symbols</i>		
$\top_P$	Top of poset $P$	→Def. 24.9 on p.349
$\perp_P$	Bottom of poset $P$	→Def. 24.9 on p.349
<i>Relations</i>		
-		
<i>Rings</i>		
$1$	identity for ring	→Def. 6.60 on p.123
$0$	identity for ring	→Def. 6.60 on p.123
$R$	Generic ring names	
$R, S$	Underlying set of rings.	
$S$	Underlying set of $S$ .	
$K$	Generic ring names	
$R, S$	Underlying set of rings.	
$L$	Underlying set of $L$ .	
<i>Semigroups</i>		
<i>Semigroups</i>		
$S, T, U$	Generic semigroup names.	
$S, T, U$	Generic names for underlying carrier set of a semigroup.	
$\log$	Matrix logarithm	
$\exp$	Matrix exponential	
$x, y, z$	Generic semigroup elements	
$F, G$	Generic semigroup morphisms.	
<i>Sets</i>		
<i>Constructors</i>		
$\text{Pow } A$	Powerset of $A$	
$\text{Pow } A$	Power set of $A$ .	→Def. 4.4 on p.44
<i>Generic sets and elements</i>		
$ x $	norm of vector $x$	
$\text{cod } f$	Codomain of function $f$	→Section 4.10 on p.62
$\text{dom } f$	Domain of function $f$	→Section 4.10 on p.62
$\cap$	Set intersection	
$\cup$	Set union	
$x \in A$	The element $x$ belongs to the set $A$ .	
$x \notin A$	The element $x$ does not belong to the set $A$ .	
$A \subseteq B$	The set $A$ is a subset of $B$ .	
$\subset$	subset	
$A \supseteq B$	The set $A$ is a superset of $B$ .	
<i>Well-known sets.</i>		
$\emptyset$	Empty set	
$\mathbb{C}$	Complex numbers	
$\mathbb{R}$	Real numbers	
$\mathbb{N}$	Natural numbers: 0, 1, 2, ...	
$\mathbb{Z}$	Integers: 0, 1, -1, 2, -2, ...	
$\mathbb{Q}$	Rational numbers	
$\mathbb{R}_{>0}$	Positive real numbers	
$\mathbb{R}_{\geq 0}$	Non-negative real numbers	
$\overline{\mathbb{R}_{\geq 0}}$	Completion of non-negative real numbers.	
$\mathbf{1}$	Singleton set, containing the element $\bullet$ .	

symbol	meaning	defined in
$\mathbb{R}^{[m]}$	A copy of $\mathbb{R}$ with units of meters	
<i>Operations</i>		
$\mathbf{A} \times \mathbf{B}$	Cartesian product of two sets.	→Def. 4.5 on p.46
$f \times g$	Product of two functions	
$f + g$	Direct sum of two functions	
$\mathbf{A} + \mathbf{B}$	Disjoint union of two sets.	
$\langle 1, a \rangle, \langle 1, b \rangle$	Decorated elements of disjoint union	
$\text{in}_1, \text{in}_2$	Injections into $\mathbf{A} + \mathbf{B}$ .	
<i>Well-known functions</i>		
$\text{id}_{\mathbf{A}}$	Identity map on $\mathbf{A}$	
$\text{ceil}$	ceiling function	→Example 11.11 on p.197
$\text{floor}$	floor function	→Example 11.11 on p.197
$\text{rntnte}$	Round to nearest, ties to even	→Example 11.11 on p.197
<i>Tuples</i>		
$\langle \rangle$	zero-size tuple	
$[]$	zero-size list	
<b>Specific use in Volume 1</b>		
<i>Part C - Algebra</i>		
<i>Chapter 7 - Morphisms</i>		
<i>ASCII example</i>		
$\text{emchar}$	includes spaces	
<i>Morse code</i>		
•	Morse dot	
—	Morse dash	
$s_1$	Silence between dots and dashes	
$s_3$	Silence between letters	
$s_7$	Silence between words	
■	Beep of $\ell$	
■ ■ ■	Beep of $3\ell$	
■	Silence of $\ell$	
■ ■ ■	Silence of $3\ell$	
■ ■ ■ ■ ■ ■ ■	Silence of $7\ell$	
<i>Chapter 8 - Actions</i>		
<b>Moo</b>	Category of Moore machines	→Def. 16.6 on p.267
<b>Mor</b>	Category of More machines	→Def. 16.17 on p.277
$P_{\text{effort}}$	effort	
$P_{\text{track}}$	tracking	
<i>Matrix groups</i>		
$O(n, \mathbb{R})$	Orthogonal group	→Def. 6.51 on p.119
$SO(n, \mathbb{R})$	Special orthogonal group	→Def. 6.53 on p.119
$GL(n, \mathbb{R})$	General linear group	→Def. 6.50 on p.119
$SL(n, \mathbb{R})$	Special linear group	→Def. 6.52 on p.119
$E(n, \mathbb{R})$	Euclidean group	→Def. 8.8 on p.147
$SE(n, \mathbb{R}^n)$	Special euclidean group	→Def. 8.9 on p.147
<i>Part E - Categories</i>		
-		
<b>Curr</b>	Currency category	→Def. 15.3 on p.247
<b>Temp</b>	Temperature category	→Exercise 30 on p.248

symbol	meaning	defined in
<b>Berg</b>	The category of Swiss mountains	→Def. 15.2 on p.244
<b>BergAma</b>		→Section 20.4 on p.321
<b>BergLazy</b>		→Section 20.4 on p.321
<i>Chapter 16 - Semicategory actions</i>		
<i>Procedures</i>		
<b>ProcTime</b>	Procedures with execution time	→Def. 15.6 on p.253
size	Size of datatype	→Def. 15.6 on p.253
<b>ProcSize</b>	Procedures with sized sets	→Def. 15.8 on p.254
$\sigma$		→Def. 15.8 on p.254
<b>ProcSizeTime</b>	Procedures with size-dependent durations	→Def. 15.9 on p.254
dur		→Def. 15.9 on p.254
<i>Part F - Functors</i>		
<i>Chapter 19 - Translation</i>		
<b>Plans</b>		
<i>Chapter 20 - Specialization</i>		
<i>drawings</i>		
<b>Draw</b>	Category of drawings	→Def. 20.4 on p.319
<i>Part G - Co-Design</i>		
<i>Construction</i>		
diag <sub>p</sub>	Diagonal function	→Def. 29.1 on p.416
codiag <sub>p</sub>	Co-diagonal function	→Def. 29.2 on p.416
<i>Part K - Compositional computation</i>		
<i>Chapter 38 - Monads</i>		
$M, N$	Generic monads.	→Def. 38.2 on p.579
un	Monad unit	→Def. 38.2 on p.579
mu	Monad identity	→Def. 38.2 on p.579
fish		→Def. 38.4 on p.582
lift		→Def. 38.4 on p.582
join		→Def. 38.4 on p.582
bind		→Def. 38.4 on p.582
fmap		→Def. 38.4 on p.582
return		→Def. 38.4 on p.582
$U$	upper-set endofunctor	
$U$	upper-set monad	
$L$	lower-set endofunctor	
$L$	lower-set monad	
<b>To categorize</b>		
:=	“defined as”	
<b>MCDP symbols</b>		





## Example exams

This chapter contains some examples of the style of the ETH class exam. Both sample exams are thought to last 90 minutes and to be open book (all notes allowed).

<b>1 Exam 1</b>	<b>604</b>
-----------------	------------



## 1. Exam 1

### Example 1: Uncertain Machines

Consider the category of Moore machines acting on signals as described in the course. Assuming the input/output sets are also ordered sets (posets), construct the Kleisli category corresponding to the interval monad  $\mathcal{U}$ . That is, the signals are closed intervals of values. Call this category **UMoore**.

1. (30%) Is **UMoore** a monoidal category?
2. (20%) Is **UMoore** a traced monoidal category?
3. (50%) Would the answers be the same if asked about the More category?

### Example 2: Machines with resources consumption

Consider the category of Moore machines. We want to be more precise about resource consumption and want to define an extension of Moore machines in which each machine also has associated a certain time  $T_1 \geq 0$  to run the dynamic function **dyn** and a certain time  $T_2 \geq 0$  for running the readout function **ro**. We call these resource-Moore machines (**RMoore**).

1. (30%) Formalize the **RMoore** category giving formulas for identities, compositions, and proof of associativity.
2. (70%) Suppose that you want to design the software for a robot. You are given the wiring diagram of the architecture, in which you have to plug in specific **RMoore** machines to implement the algorithmic functionality. Assume that the wiring diagram does not contain any loop - only series and parallel composition, and that there is only one input (robot observations) and one output (robot commands). For each hole in the diagram, you are given a set of 1 or more **RMoore** machines that can implement the functionality. Assume that the computer on which to run everything has  $N \geq 1$  processors. Think of each processor as a “lane” in which the operations of each machine are cars that must run sequentially.

Formalize the design problem in the category **DPI** that corresponds to choosing the best combination of machines and the best assignment to processors. Include as a resource the number of processors and as functionality the throughput of the system (how many commands are generated per second).

# Bibliography

- [1] Sergei L Bezrukov and Ian T Roberts. «On Antichains in Product Posets». In: ().
- [2] Yangjun Chen and Yibin Chen. «On the Decomposition of Posets». In: *2012 International Conference on Computer Science and Service System*. Aug. 2012, pp. 134–138. DOI: 10.1109/CSSS.2012.41.
- [3] P. Cuffe and A. Keane. «Visualizing the Electrical Structure of Power Systems». In: *IEEE Systems Journal* 11.3 (2017), pp. 1810–1821. DOI: 10.1109/JSYST.2015.2427994.
- [4] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002. ISBN: 9780521784511. DOI: 10.1017/cbo9780511809088.
- [5] Magnus Egerstedt. «Motion Description Languages for Multi-Modal Control in Robotics». In: *Control Problems in Robotics*. Springer Science + Business Media, 2003, pp. 75–89. DOI: 10.1007/3-540-36224-x\_5.
- [6] Brendan Fong and David I Spivak. *An invitation to applied category theory: seven sketches in compositionality*. Cambridge University Press, 2019.
- [7] G. Gierz et al. *Continuous Lattices and Domains*. Cambridge University Press, 2003. DOI: 10.1017/cbo9780511542725.
- [8] Petr A Golovach et al. «An Incremental Polynomial Time Algorithm to Enumerate All Minimal Edge Dominating Sets». In: *Algorithmica* 72.3 (June 2015), pp. 836–859.
- [9] Michael Grant and Stephen Boyd. «Graph implementations for nonsmooth convex programs». In: *Recent Advances in Learning and Control*. Ed. by V. Blondel, S. Boyd, and H. Kimura. Lecture Notes in Control and Information Sciences. [http://stanford.edu/~boyd/graph\\_dcp.html](http://stanford.edu/~boyd/graph_dcp.html). Springer-Verlag Limited, 2008, pp. 95–110.
- [10] G. Grisetti, C. Stachniss, and W. Burgard. «Improved Techniques for Grid Mapping With Rao Blackwellized Particle Filters». In: *IEEE Transactions on Robotics* 23.1 (Feb. 2007), pp. 34–46. ISSN: 1552-3098. DOI: 10.1109/TR0.2006.889486.
- [11] Masahito Hasegawa. «The Uniformity Principle on Traced Monoidal Categories». In: *Category Theory and Computer Science, CTCS 2002, Ottawa, Canada, August 15-17, 2002*. Ed. by Richard Blute and Peter Selinger. Vol. 69. Electronic Notes in Theoretical Computer Science. Elsevier, 2002, pp. 137–155. DOI: 10.1016/S1571-0661(04)80563-2. URL: [https://doi.org/10.1016/S1571-0661\(04\)80563-2](https://doi.org/10.1016/S1571-0661(04)80563-2).

- [12] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. Aug. 2008, p. 58. ISBN: 0-7381-5753-8 (paper), 0-7381-5752-X (electronic). DOI: <https://doi.org/10.1109/IEEESTD.2008.4610935>. URL: [http://en.wikipedia.org/wiki/IEEE\\_754-2008](http://en.wikipedia.org/wiki/IEEE_754-2008); <http://ieeexplore.ieee.org/servlet/opac?punumber=4610933>.
- [13] ISO. *ISO 80000-2: Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*. Dec. 2009, p. 40. URL: <https://www.iso.org/standard/31887.html>.
- [14] S. M. LaValle. *Sensing and Filtering: A Fresh Perspective Based on Preimages and Information Spaces*. Foundations and Trends in Robotics Series. Delft, The Netherlands: Now Publishers, 2012. DOI: 10.1561/230000000004.
- [15] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. 1st ed. Available for download on the authors' website <http://leeseshia.org/>. 2010. ISBN: 978-0-557-70857-4.
- [16] Nancy. Leveson. *Engineering a safer world systems thinking applied to safety*. eng. Engineering systems. Cambridge, Mass: MIT Press.
- [17] Alex Locher, Michal Perdoch, and Luc Van Gool. «Progressive Prioritized Multi-view Stereo». In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)* (2016).
- [18] A Lodi, S Martello, and M Monaci. «Two-dimensional packing problems: A survey». In: *European Journal of Operational Research* 141.2 (2002), pp. 241–252.
- [19] Alan MacCormack, Carliss Baldwin, and John Rusnak. «Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis». In: *Research Policy* 41.8 (Oct. 2012), pp. 1309–1324. ISSN: 00487333. DOI: 10.1016/j.respol.2012.04.011. URL: <http://dx.doi.org/10.1016/j.respol.2012.04.011>.
- [20] E.G. Manes and M.A. Arbib. *Algebraic approaches to program semantics*. Springer-Verlag, 1986. ISBN: 9780387963242. DOI: 10.1007/978-1-4612-4962-7.
- [21] Donald B. McIntyre. «The Role of Composition in Computer Programming». In: *SIGAPL APL Quote Quad* 25.4 (June 1995), pp. 116–133. ISSN: 0163-6006. DOI: 10.1145/206944.206985. URL: <http://doi.acm.org/10.1145/206944.206985>.
- [22] S.E. Morison. *Admiral of the Ocean Sea: A Life of Christopher Columbus*. Read Books, 2007. ISBN: 9781406750270. URL: <https://books.google.ch/books?id=T5x5xjsJtlwC>.
- [23] Luigi Nardi et al. «Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM». In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. May 2015.
- [24] Jason M. O’Kane and Steven M. LaValle. «On comparing the power of robots». In: *International Journal of Robotics Research* 27.1 (Jan. 2008), pp. 5–23.
- [25] Benjamin C. Pierce. *Types and Programming Languages*. 1st ed. MIT Press, Feb. 2002. ISBN: 0262162091. URL: [http://ropas.snu.ac.kr/~{kwang}/520/pierce\\_book.pdf](http://ropas.snu.ac.kr/~{kwang}/520/pierce_book.pdf).
- [26] E. Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2017. ISBN: 9780486820804. URL: <https://books.google.ch/books?id=6B9MDgAAQBAJ>.
- [27] S. Roman. *Lattices and Ordered Sets*. Springer, 2008. ISBN: 9780387789019. DOI: 10.1007/978-0-387-78901-9.
- [28] Herbert A. Simon. *The Sciences of the Artificial (3rd Ed.)*. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0262691914.
- [29] Stefano Soatto. «Steps Towards a Theory of Visual Information: Active Perception, Signal-to-Symbol Conversion and the Interplay Between Sensing and Control». In: *CoRR* abs/1110.2053 (2011). URL: <http://arxiv.org/abs/1110.2053>.
- [30] David I Spivak. «Categorical databases». In: *Presented at Kensho* (2019).
- [31] David I Spivak and Robert E Kent. «Ologs: a categorical framework for knowledge representation». In: *PloS one* 7.1 (2012), e24274.
- [32] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. 1st. New York, NY, USA: Marcel Dekker, Inc., 2000. ISBN: 0824793188. DOI: 10.1201/9781420048025.
- [33] Maria Svorenova et al. *Resource-Performance Trade-off Analysis for Mobile Robots*. 2016. eprint: [arXiv:1609.04888](https://arxiv.org/abs/1609.04888).
- [34] Philip Wadler. «Linear Types Can Change the World!» In: *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.
- [35] Olivier L. de Weck, Daniel Roos, and Christopher L. Magee. *Engineering Systems: Meeting Human Needs in a Complex Technological World*. The MIT Press, Oct. 2011. ISBN: 9780262298513. DOI: 10.7551/mitpress/8799.001.0001. URL: <https://doi.org/10.7551/mitpress/8799.001.0001>.
- [36] M. Zeeshan Zia et al. «Comparative Design Space Exploration of Dense and Semi-Dense SLAM». In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2016.

# Index

adjoint equivalence, **490**  
adjunction, **488**, 488, **489**, 489–492, 494  
algebra of a monad, **587**  
antichain in a poset, **168**, 168–172, 354, 365–367, 547, 552, 554, 555, 558, 562, 568, 569  
antisymmetric relation, **91**, 159  
antitone Galois connection, **486**  
antitone map, **194**, 194, 195, 353, 486  
arrow category, **289**  
arrow poset of intervals, **186**  
associative law, 106–108, 112, 137, 149, 215  
associative stacking semicategory, 427, **431**, 431, 432, 434–437, 454, 455, 495  
associativity, 105, 106, 262, 263, 265, 267, 373, 386, 391, 420–422, 426, 427, 436, 438, 575, 582  
asymmetric relation, **91**  
automorphism, 144  
  
bijective function, **64**, 64, **65**, 66, 68, 98, 118, 127, 132, 518  
**Bool** (poset/category of booleans), **161**  
**Bool** set of booleans, **42**

bottom element of a poset  $\perp$ , 410  
bounded lattice, **400**, 400, 401, 403, 404, 407, 410, 411, 539, 540, 544, 548  
bounded lattice homomorphism, **403**  
**BoundedLat** (category of bounded lattices and lattice homomorphisms), **404**  
braided monoidal category, 427, **501**, 501  
braiding **br**, **501**  
  
cardinality of a set **card**, **39**  
cartesian product, **46**, 46–48, 59, 67, 73, 79, 143, 176, 262, 263, 285, 426, 497, 532, 534  
**Cat** (category of small categories), **329**  
categorical coproduct, 48  
categorical product, 48, 329  
category, **216**, 444  
category of currencies, **247**  
category of mountain trips, **244**  
chain in a poset, **168**, 168–172, 177, 549, 565, 568  
co-design problem with implementation, **369**  
codiagonal function, **416**

commutative diagram, **219**, 289, 290  
 commutative group, **117**  
 commutative magma, **104**  
 commutative semigroup, **105**, 117  
 companion construction, 416, 418, 419  
 complete lattice, **410**  
 composition (of functors), **328**, 329  
 composition of functions, **65**  
 composition of relations, **81**  
 concatenation of tuples  $\circ$ , 264  
 conjoint construction, 418, 419  
 contravariant semigroup action, **146**, 146  
 coslice category, **292**

design complexity, **569**  
 design problem (with implementation), **358**, 375  
 design problem (without implementation), 339, 357, 361, 364,  
 365, 369, 370, 374, 375, **378**, 378–382, 384, 385,  
**391**, 391, 406–411, 418–420, 437, 502, 528, 552,  
 553, 559, 561–563, 566, 569, 570, 604  
 diagonal function, **416**  
 directed complete partial order, **548**  
 directed graph, **204**  
 directed multigraph, **204**  
 directed subset, **548**, 548  
 disjoint union (of categories)  $+$ , **287**, 287  
 disjoint union (of posets)  $+$ , **179**  
 disjoint union (of sets)  $+$ , 47, 48, 61, 67, 73, 179, 180  
 distributive lattice, **411**  
 downward-closed, **354**  
**DP** (category of design problems), **391**  
**DPI** (semicategory of design problems with implementation),  
**373**  
 $\langle \mathbf{DP} \rangle$  (category of tuple-posets and design problems), **437**  
**Draw** category of drawings, **319**

**Eff** (category of sets and functions with side effects), **434**  
 endofunctor, **309**, 309, 311  
 endomorphism, 144, 505, 507  
 endorelation, 88, **91**, 91–93, 95, 159  
 equivalence of categories, **489**  
 equivalence of LTI systems, **280**  
 equivalence relation, **96**, 96  
 everywhere-defined, **87**, 87, 89

finite design problem, **552**  
 finitely supported upper set, **552**  
**FinSet** (category of finite sets and functions), **318**  
 fixed point, **549**  
 forgetful functor, 391  
 free semicategory on a graph, **222**  
 free semigroup, 137  
 function, **64**  
 function isomorphism, **66**  
 functor, 305, 306, **308**, 308–312, 314, 315, 317, 327–329,  
 331–333, 418, 426, 444, 463, 465, 467, 473, 475,  
 476, 488, 489, 491–494, 496, 497, 527, 529, 531,  
 532, 543, 579, 580, 586  
 functorial stacking semicategory, **439**, 439–441, 443

graph homomorphism, **206**, 206, 207, 238, 364

greatest lower bound, **349**  
 group, 103, 104, **117**, 117–119, 122, 134–136, 138, 141, 144,  
 146, 147, 210, 234  
 group action, **144**, 144  
 group morphism, **133**, 133, 134, 144, 234  
 groups of symmetries, 117  
**Grp** (category of groups and morphisms), **234**  
**Grph** (category of graphs and graphs homomorphisms),  
**238**

Hasse diagram, **160**, 161  
 hom-sets, **213**, 219, 405, 407, 410

identity design problem, **387**, 418  
 identity function, **66**, 126, 134, 144, 216, 234, 299  
 identity functors, 309, **310**, 329, 475, 476, 497, 531, 532  
 identity matrix, **120**, 235  
 identity morphism, **216**, 216, 217, 219, 222, 232–234, 239, 244,  
 246, 247, 252, 286, 294, 299, 310, 312, 318, 320,  
 321, 329, 331, 332, 391, 404, 445, 476, 493, 499,  
 518, 530, 532, 552, 587  
 identity morphism of groups, **134**  
 identity morphism of monoids, **131**  
 identity morphism of semigroups, **126**  
 identity natural transformation, 469  
 infimum, **349**  
 injective function, **64**, 65, 98, 232, 299, 320  
 injective relation, **87**, 87, 89  
**InjSet** (category of sets and injective functions), **320**  
 intersection of sets  $\cap$ , **44**  
 interval, **183**  
 inverse, 518  
 irreflexive relation, **91**  
 isomorphism, **518**  
 isomorphism (of categories), 489  
 isomorphism (of groups), **134**  
 isomorphism (of monoids), **131**, 132  
 isomorphism (of semigroups), **126**, 127, 128  
 isomorphism (of sets), **66**  
 isomorphism in a category, **518**

join, **348**, 348, 400, 410, 539, 540, 544

Kleisli category, **583**, 604  
 Kleisli composition, **583**, 583  
 Kleisli morphism, **583**

**Lat** (category of lattices and lattice homomorphisms), **404**  
 lattice, 117, 349, **400**, 400–405, 410–412, 421, 528, 529, 548  
 lattice homomorphism, **403**, 403, 404, 421  
 least fixed point, **549**, 549, 550, 553, 556, 563, 568  
 least upper bound, **348**  
 left adjoint, 488, **489**  
 lower bounds, **348**  
 lower closure, **353**, 354, 356, 530  
 lower set, 345, **351**, 351, 353, 355, 382, 383, 534  
**LTI** (category of finite-dimensional linear time-invariant  
 systems), **280**

magma, **104**, 104–106  
**Mat<sub>ℝ</sub>** (category of real matrices), **235**

matrix groups, 117, 134  
 matrix multiplication, 235  
 meet, **349**, 349, 400, 410, 539, 540  
**Mon** (category of monoids and morphisms), **234**, 309  
 monad, 529, 573, 574, **579**, 583, 585  
 monoid, 103, 104, **112**, 112–118, 120, 122, 125, 131, 133–136, 138, 141, 144, 146, 209, 210, 216, 234, 251, 252, 309, 396, 426, 577  
 monoid action, **144**, 144  
 monoid morphism, **131**, 131–134, 144, 234  
 monoidal category, 7, 427, 495, **496**, 497, 501–503, 505, 506, 532  
 monoidal poset, **396**, 396, 398  
 monotone Galois connection, **486**  
 monotone map, **190**, 190, 194–197, 236, 310, 312, 354, 378–383, 396, 398, 418, 486, 521, 530, 531, 533, 534, 536, 563, 575, 576  
**Moo** (semicategory of Moore machines), **267**  
 Moore machine, 260–263, **266**, 266, 267, 269, 270, 272–277, 324, 426, 604  
**Mor** (semicategory of More machines), **277**  
 More machine, 277, 278  
  
 natural isomorphism, **470**, 488–490, 494, 496, 501–503  
 natural transformation, 7, 306, 427, 463, **465**, 465–467, 470, 474–476, 489, 494, 495, 497, 579–581, 586  
 neutral element of a monoid, 18, 113, 114, 116–118, 120, 121, 131, 133, 134, 136, 144, 216, 252  
  
 opposite category, **288**, 288  
 opposite poset, **181**, 288  
 order on monotone maps, **197**  
  
 partition, **96**, 172, 365  
 path in a graph, **204**  
 permutation, **520**  
**Pos** (category of posets and monotone maps), **236**  
 poset, 50, **159**, 159, 161, 163, 164, 167, 168, 171, 172, 175–181, 183–186, 190, 191, 194–197, 236, 239, 288–290, 310, 343, 346, 348, 353, 365, 367, 369, 373, 378, 380, 382–385, 391, 396, 398, 400, 402, 406, 418, 420, 421, 436, 437, 440, 486, 532–534, 548, 552, 553, 569, 575  
 positive definite matrices, **162**  
**⟨Pos⟩** (category of tuple-posets and monotone maps), **436**  
 power poset, **166**  
 powerset, **44**, 44, 45, 382, 467, 579  
 pre-ordered set, **158**, 158, 159  
 product (of posets)  $\times$ , **176**, 396  
 product of categories, **286**  
 product of functions  $\times$ , **67**  
 profunctor, 530  
  
 quasitransitive relation, **173**  
  
 real vector space, **124**, 235  
 reflexive relation, **91**, 92, 96, 158, 160, 164, 239  
**⟨Rel⟩** (category of tuple-sets and relations), **437**  
**Rel** (category of sets and relations), **233**  
 relation, **79**, 84  
 relation on a semigroup, **136**  
  
 right adjoint, 488, **489**  
 right dual object, **506**  
  
 Scott continuity, **548**, 548, 549, 554, 555  
 semicategory, v, 7, 209–212, **213**, 213–216, 219, 221, 253, 254, 262, 263, 267, 272, 273, 307, 357, 373, 374, 398, 425–427, 429, 431, 433–436, 439, 454, 455, 498, 575  
 semicategory action, **273**, 275, 276, **310**, 310  
 semifunctor, 7, **307**, 307, 308, 310, 391–393, 439  
 semigroup, 6, 104, **105**, 105–108, 110–113, 115, 116, 125–128, 131, 134–138, 141, 143, 144, 146, 209, 210, 216, 234, 272, 426, 431  
 semigroup action, **144**, 144, 148, 272  
 semigroup morphism, **126**, 126–128, 131, 133, 138, 144, 234, 272, 310  
 semiorder, **173**  
**⟨Set⟩** (category of tuple-sets and functions), **264**  
**Set** (category of sets and functions), **217**, 309  
 setoid, **49**, 49, 50, 58, 59, 70  
**SGrp** (category of semigroups and morphisms), **234**  
 single-valued, **87**, 87, 89  
 singleton set, **43**  
 skeletal category, 239  
 slice category, **292**  
 stacking semicategory, **429**, 431, 445  
 standard action of Moore machines, **273**  
 strict monoidal category, 7, 427, **448**, 450  
 strict monoidal semicategory, **448**  
 strictly monotone, **521**  
 strong monoidal functor, **503**  
 subcategory, 7, **318**, 318–321, 436, 437, 552  
 subgroup, **121**  
 sublattice, 550  
 submonoid, **114**, 114  
 subsemigroup, **107**  
 sum of functions  $+$ , **68**  
 supremum, **348**, 411, 548, 549  
 surjective function, **64**, 64, 65, 98, 232  
 surjective relation, **87**, 87, 89  
 symmetric monoidal category, 427, **501**, 502, 507, 508, 533, 536  
 symmetric monoidal poset, **396**  
 symmetric relation, **91**, 92, 96  
  
 terminal object, **506**  
 top element of a poset  $T$ , 410, 548  
 total relation, **92**, 159  
 totally ordered set, **159**  
 traced symmetric monoidal category, 495, 509, 528, 536, 570  
 transitive closure of a relation, **94**, 94, 95, 550  
 transitive relation, 92, **94**, 94, 96, 158, 160, 161, 164, 239  
 transpose of a relation, **89**  
 twisted arrow category, **290**  
 twisted poset of intervals, **183**  
  
 union of sets  $\cup$ , **44**  
 unitality property, 391  
 upper bounds, **348**  
 upper closure, **353**, 353, 530, 547, 552

upper set, 312, **351**, 351–354, 380, 381, 383, 476, 528, 531, 532,  
534, 537, 547, 552, 577

upward-closed, **355**

**$\mathbf{Vect}_{\mathbb{R}}$**  (category of real vector spaces), **235**

vector space, 279, 311, 475, 476, 488, 492, 499, 505–507

width of a poset, **171**