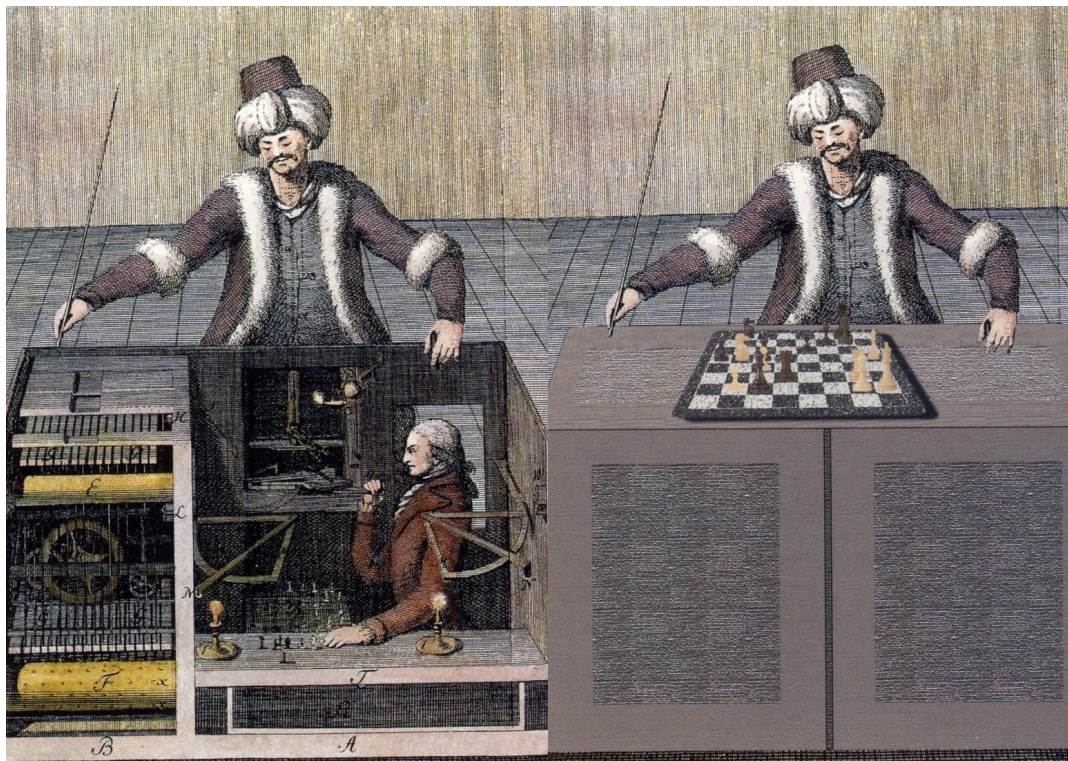


# Programs as Diagrams

## From Categorical Computability to Computable Categories

Dusko Pavlovic

20221212



## blurb

Computers changed the world. They know more about people than people about them. In this century, computation completely changed the practices of science, but computer science and the theory of computation remained largely unchanged.

This book depicts computations as string diagrams. The diagrammatic depictions can be viewed as programs in a single-instruction programming language called Run. The single instruction is called RUN and it is drawn as a rectangle with a cut-off corner, where programs are plugged in. The programs as diagrams are computer-drawn, human-generated, and tested on students.

20221212

# Contents

<b>Preface</b> . . . . .	<b>v</b>
<b>What?</b> . . . . .	<b>vii</b>
<b>1 Drawing types and functions</b> . . . . .	<b>3</b>
<b>2 Monoidal computers: computability as a structure</b> . . . . .	<b>25</b>
<b>3 Fixpoints</b> . . . . .	<b>49</b>
<b>4 What can be computed</b> . . . . .	<b>65</b>
<b>5 What cannot be computed</b> . . . . .	<b>89</b>
<b>6 Computing programs</b> . . . . .	<b>99</b>
<b>7 Stateful computing</b> . . . . .	<b>123</b>
<b>8 Program-closed categories: computability as a property</b> . . . . .	<b>125</b>
<b>9 Computability as continuity</b> . . . . .	<b>127</b>
<b>What??</b> . . . . .	<b>129</b>
<b>Appendices</b> . . . . .	<b>131</b>
<b>Bibliography</b> . . . . .	<b>157</b>
<b>Index</b> . . . . .	<b>170</b>

20221212

20221212

# Preface

Pythagoreans discovered that the world was built of numbers. Their rituals with numbers, allowing you to measure some sides of objects and to compute the size of other sides, were called *mathemata*. Systematized into *mathematics*, those rituals demonstrated that the world was regular and predictable.

The world spanned by computers and networks is definitely built of numbers. Mathematics, in the meantime, evolved beyond and away from the Pythagorean computing rituals. While programmers program computers in a great variety of languages, mathematicians study mathematics in a great variety of completely different languages. A great variety of languages is a good thing, provided that the languages do not isolate but diversify the views of the world and that there are multi-lingual communities to connect them. Many mathematicians speak programming languages; many programmers are conversant with mathematics. The language of categories seems convenient for both, and to some extent connects the communities.

In this book, we use a categorical language based on string diagrams that is so basic that it must be contained in every mathematical model of computation, and so compact that every programming language must project onto it. It evolved through many years of teaching mathematics of computation. The diversity of the mathematical and the programming languages makes such teaching and learning tasks into lengthy affairs, loaded with standard prerequisites. By peeling off what got standardized by accident and following the shortcuts that emerged, and then peeling off what became unnecessary after that, we arrived at a mathematical model of computation generated by a single diagrammatic element, which is also a single-instruction programming language. I know it sounds crazy, but read on.

The presentation is self-contained, assuming enough confidence and curiosity. The early chapters have been taught as undergraduate courses, the middle chapters as graduate courses. The final chapters contain the results that make a point of it all. Without a teacher, a reader familiar with categories is hoped to get easy access to the basic concepts of computability, a reader familiar with computability to basic category theory, and a reader familiar with both is hoped to make use of the many opportunities to improve the approach.

**The “workouts” and “stories” should probably be skipped the first time around.** They used to be called *Exercises* and *Historic Background* but that was misleading. The workouts concern the issues that the reader might like to consider on their own first. Some are presented as guided exercises, but some are much more than exercises, and some are just asides. Most are worked out in Appendix 5. The stories are sort of like philosophical gossips, precisely what the math is meant to defend us from, so you are welcome to skip them altogether.

20221212

# What?

---

1	What are we talking about? . . . . .	vii
2	What is computer science? . . . . .	vii
3	What is computer science about? . . . . .	viii
4	What is computation? . . . . .	viii
5	What is a computer? . . . . .	ix

---

## 1 What are we talking about?

We spell out the basic math behind computer science. Everyone knows what is a computer. Most people also know what is science, almost as confidently. Computers run programs. Programmers program programs. Why is there a computer science?

## 2 What is computer science?

Science inputs processes that exist in nature and outputs their descriptions. Engineering inputs descrip-

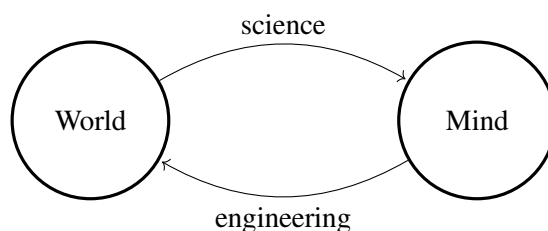


Figure 1: Civilisation as a conversation

tions of some desirable processes that do not exist in nature and outputs their realizations. Fig. 1 shows the resulting conversation staged by scientists and engineers. Computers and programs were originally built by engineers. After lots of computers were built and lots of programs programmed, they formed networks, and the Big Bang of the Internet created cyberspace. The cyberspace clumped into galaxies, life emerged in it, botnets and viruses, influencers and celebrities, genomic databases, online yoga studios. We live in cyberspace. Our children grow there. Like our cities, freeways, and space stations, it was built by engineers, but then life took over. Life in cyberspace is not programmable and it cannot be engineered anymore. Economy, security, epidemics are natural processes, whether they spread through physical space or cyberspace, or through a mixture. Cyberspace is the space of computations, spanned by the networks of people and computers. They act together, evolve together, and it is hard to tell who is who. They are the subject of the same science.

### 3 What is computer science about?

Biology is about life, and life comes about by evolution. Chemistry is about molecules, and they come about by binding atoms. And so on. Every science is about some basic principle (or two, in the case of modern physics) and when you come in to study it, on day one they tell you what it is about. You study biology and the first lesson goes: "Biology is about evolution". Not so when you study computer science. What is the basic principle of computation? What is  $X$  in the following equations?

$$\frac{\text{evolution}}{\text{biology}} = \frac{\text{chemical bonds}}{\text{chemistry}} = \dots = \frac{X}{\text{computer science}} \quad (1)$$

The answer  $X = \text{computability}$  is usually taught in a "theory" course in the final year of computer science. Many students avoid this course. Computer science and computer engineering are often taught together, which makes many courses avoidable. Those who do take a "theory" course learn that a function is computable if it can be implemented using a Turing machine, or the  $\lambda$ -calculus, or recursion and minimization schemas, or a cellular automaton, or a uniform boolean circuit, or one of many other theoretical models of computation. The Turing machine and the  $\lambda$ -calculus stories go back to the 1930s. The other models go back to the surrounding decades. Each of them is a story about a different aspect of computation, as seen by different people in different contexts. Each makes the same class of functions computable, but in each case they are encoded differently. The *Church-Turing Thesis* postulates that each of the defined notions of computation is the same, modulo the encodings. What is their common denominator?

### 4 What is computation?

The process of computation can also be viewed as a conversation. Fig. 2 shows the idea. Programs are

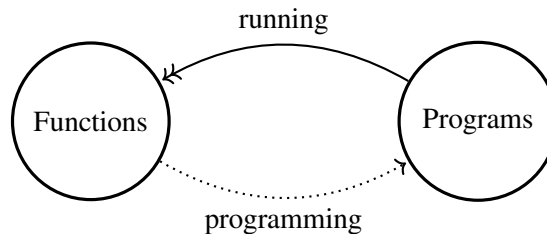


Figure 2: Computation as a conversation

the mind of a computer. Functions are its world. Programmers call anything that takes inputs and produces outputs a function. They analyze functions and synthesize programs. Computers run the programs and compute the functions. While running a given program is a routine operation, programming requires creativity and luck. That is why the two arrows in Fig. 2 are different. The difference impacts the sciences concerned with complexity, information, evolution, language [14]. The computational origin of this difference is described in Ch. 5. A version with the symmetry of running and abstracting restored is displayed in Fig. 5.2.



## 5 What is a computer?

We interact with many different computers, many different models of computation, many different programming languages. But they are avatars of the same character in the same story. We speak different languages in different societies, but they are manifestations of the same capability of speech and social interaction. What makes a language a language? What makes a sentence different from the bark of a dog? What makes a program different from a sentence? What makes a computer a computer?

Computers and programs are the two sides of the same coin. Programs are programs because there are computers to run them. A computer is a computer because there are programs to compute. While Fig. 2 displays the two sides of the coin, Fig. 3 distinguishes the head and the tail. The equation identifying them is the general schema of computation, overarching all different models of computation and underlying all different computers. This is their common denominator. This is what makes a computer

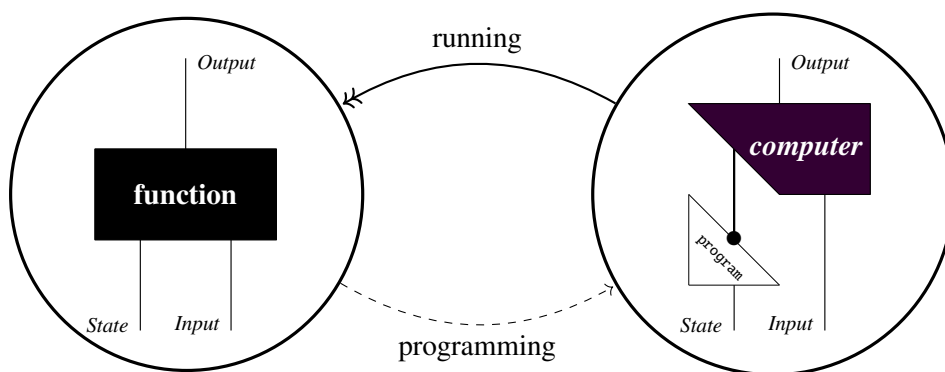


Figure 3: Any computer can be programmed to compute any computable function

a computer. It is also the main structural component of the monoidal computer, described in Ch. 2, formalized in Sec. 2.2.1. The “running” surjection is spelled out in Sec. 2.5.3. All capabilities and properties of real computers can be derived from it. An abstract computer can thus also be viewed as a single-instruction programming language, called Run, with run as its only instruction. The main computational constructs are derived in chapters 3–6. The conceptual insights and the technical stepping stones into further research are in chapters 7–9. The string diagrams, convenient for building programs from runs and for composing computations are introduced in Ch. 1.

20221212

# Contents

<b>1</b>	<b>Drawing types and functions</b>	<b>3</b>
<b>2</b>	<b>Monoidal computers: computability as a structure</b>	<b>25</b>
<b>3</b>	<b>Fixpoints</b>	<b>49</b>
<b>4</b>	<b>What can be computed</b>	<b>65</b>
<b>5</b>	<b>What cannot be computed</b>	<b>89</b>
<b>6</b>	<b>Computing programs</b>	<b>99</b>
<b>7</b>	<b>Stateful computing</b>	<b>123</b>
<b>8</b>	<b>Program-closed categories: computability as a property</b>	<b>125</b>
<b>9</b>	<b>Computability as continuity</b>	<b>127</b>
	<b>What??</b>	<b>129</b>
	<b>Appendices</b>	<b>131</b>

20221212

# 1 Drawing types and functions

---

1.1	Types as strings . . . . .	4
1.1.1	Types . . . . .	4
1.1.2	Strings instead of variables . . . . .	4
1.1.3	Tupling and switching . . . . .	5
1.2	Data services: copying and deleting . . . . .	6
1.3	Monoidal functions as boxes . . . . .	8
1.3.1	Composing functions . . . . .	8
1.3.2	The middle-two-interchange law . . . . .	9
1.3.3	Sliding boxes . . . . .	10
1.4	Cartesian functions as boxes with a dot . . . . .	10
1.5	Categories . . . . .	12
1.6	Workout . . . . .	14
1.7	Stories . . . . .	16
1.7.1	Type by any other name . . . . .	16
1.7.2	Categories as type universes . . . . .	17
1.7.3	Logics of types . . . . .	18
1.7.4	Categorical diagrams: chasing arrows and weaving strings . . . . .	20

---

## 1.1 Types as wires: Data passing without variables

Whatever else there might be in a computer, there are always lots of wires. In modern computers, most wires are printed on a chip. In higher animals, the wires grow as nerves. In string diagrams, the wires are drawn as strings.

### 1.1.1 Types

Sets are containers with contents. Types are just the containers. When you add or remove an element from a set, you get another set, but the type remains the same. The difference is illustrated in Fig. 1.1, sets on the left, types on the right. When the elements keep coming and going a lot, programmers use types to keep them apart. A set is completely determined by its elements, whereas a type is determined by a property of that characterize its elements, no matter how many of them there might be. E.g., there is only one empty set, but there are as many empty types as there are colors of unicorns: the type of blue unicorns is completely different from the type of green unicorns, although they are both empty. In Fig. 1.2 on the left, the types are drawn as strings. The same types are drawn on the right but



Figure 1.1: Sets are containers with contents. Types are just the containers.

distinguished by names and not by colors. The strings separate different data items. Their names (or colors) tell which ones are of the same type.

### 1.1.2 Strings instead of variables

When we write a program, we usually plan how to process some asyetunknown data, that will become known when the program is run. In algebra and in most programming languages, such *indeterminate* data values are manipulated using *variables*. In computers, they are moved around through wires, depicted as strings. They carry the data and thus play the role of variables. The type annotations constrain the data that can flow through each string. In Fig. 1.2 on the right, the strings are typed  $A, B, C$  and  $D$ , and the variables  $x : A, y : B, u : C$  and  $v : D$ , corresponding to each particular wire, are also displayed (just this time). Different variables of the same type are represented by different strings with the same type annotation.

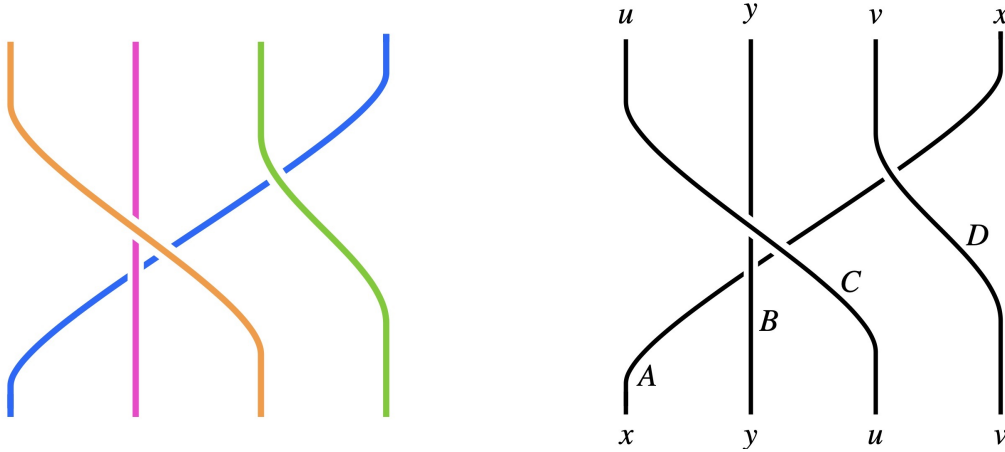


Figure 1.2: Types are wires conducting data.

### 1.1.3 Tupling and switching

**Product types.** An  $n$ -tuple of data is carried by an  $n$ -tuple of variables through an  $n$ -tuple of parallel wires. The wires of types  $A_1, A_2, \dots, A_n$  run in parallel correspond to the product type  $A_1 \times A_2 \times \dots \times A_n$ , as displayed in Fig. 1.3. An  $n$ -tuple product is the type of an  $n$ -tuple of values or variables  $\langle x_1, x_2, \dots, x_n \rangle$  :

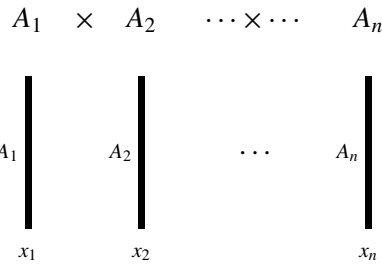


Figure 1.3: Parallel wires correspond to product types

$A_1 \times A_2 \times \dots \times A_n$ . Some of the  $A_i$ s may be the occurrences of the same types. If  $A_1 = A_2 = \dots = A_n$  are the same type  $A$ , then the product of  $n$  copies of this type is written  $A^n$ .

**The unit type** is the type  $I$  such that  $I \times X = X = X \times I$  holds for all types  $X$ . In string diagrams, this is captured by representing the type  $I$  by the *invisible* string. In that way, the string diagram representations for the type  $X$  and for the type expressions  $X \times I$ ,  $I \times X \times I \times I$  and  $I^n \times X$  are all the same<sup>1</sup>. The unit type  $I$  can be thought of as a product of 0 copies of any type  $X$ , since  $X^0 = I$ .

**Twisting the wires** like in Fig. 1.4, corresponds to changing the order of the variables in an algebraic expression. In an algebra of wires, the operation of twisting corresponds to the operation  $\zeta$ .

**Elements.** The functions in the form  $a : I \rightarrow A$  are called the *elements* of type  $A$ , and abbreviated to  $a : A$ . When  $A$  is a set and  $I$  is a single-element set, then the total, single-valued functions  $I \rightarrow A$  correspond to the elements of  $A$  in the usual sense. The single element of  $I$  corresponds to the identity function  $I \rightarrow I$ . Since  $I$  can be thought of as the type of empty tuples, satisfying  $X^0 = I$  for every

<sup>1</sup>Having invisible graphic elements will turn out to be quite convenient. Having invisible points in geometry and empty instructions in programs is not just convenient, but necessary [43].

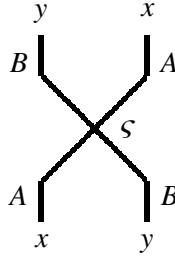


Figure 1.4: The wires are twisted to get the data where needed

type  $X$ , the single element of  $I$  can be thought of as the empty tuple  $()$ . While the identity is the only total, single-valued function on a set with one element, and  $() : I$  is unique in that sense, the functions  $f : A \rightarrow B$  that arise in computation may not be total or single-valued, as they may not always return an output, or they may return different outputs when fed same inputs at different times. Intuitively, this means that their outputs depend on a hidden state of the world. This is one of the reasons why types may not boil down to sets of their elements. Such issues cannot be ignored in computation. The next section presents the operations for handling the general functions that may not be total or single-valued. Sec. 1.4 tells how to recognize those that are.

## 1.2 Data services: copying and deleting using the cartesian structure

Mentioning a variable in several places in an algebraic expression or in a program allows reusing the same value several times. If a variable is not mentioned, then it is not used. The variables thus allow *copying* data where needed and *deleting* data where they are not needed. In categorical logic, the variable-free view of the data services of copying and deleting has been developed as the *cartesian structure*. In a string diagram, the data transmitted through a wire get copied when the wire branches, and they get deleted when the wire is cut. The data service operations, comprising the categorical structure, are thus:

- copying  $\Delta : A \rightarrow A \times A$  and
- deleting  $\uparrow : A \rightarrow I$ .

We assume that any type is equipped with such a pair of operations. They are drawn as black beads.

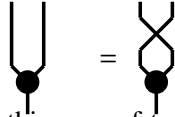
(1.1)

They are required to satisfy the following equations:

$$\begin{aligned}
 (\Delta \times \text{id}) \circ \Delta &= (\text{id} \times \Delta) \circ \Delta & (\uparrow \times \text{id}) \circ \Delta &= \text{id} = (\text{id} \times \uparrow) \circ \Delta
 \end{aligned}$$
(1.2)



$$\Delta = \zeta \circ \Delta$$



where  $\text{id}$  denotes the identity function, in this case of type  $A$ . Since all strings are of type  $A$ , the type annotations are omitted. Sec. 1.3 introduces functions as boxes with input and output strings, but  $\text{id}$  just outputs its inputs on the same string, and boils down to the string. The algebraic equations are written on top of the string diagram equations for clarity, in both cases with the type annotation

**Explanation.** The copying operation can be thought of as mapping  $x : A$  to  $\Delta_A(x) = \langle x, x \rangle$ . The *associativity* of  $\Delta$  on the left becomes  $\langle \langle x, x \rangle, x \rangle = \langle x, \langle x, x \rangle \rangle$ , which allows us to write  $\langle x, x, x \rangle$  without ambiguity, like in Fig. 1.5. Fig. 1.6 shows an example of moving data around by branching, cutting, and



Figure 1.5: The associativity of  $\Delta$  allows copying and tupling

twisting wires.

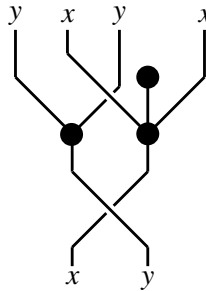


Figure 1.6: Using wires to rearrange  $\langle x, y \rangle$  into  $\langle y, x, y, x \rangle$

**Copying and deleting the empty tuple.** The data services on the unit type  $I$  boil down to the identity map. Intuitively, since  $()$  is the empty tuple already, the deletion must be  $\dagger_I = \text{id}_I$ . Since  $I \times I = I$ , copying also boils down to  $\Delta_I = \text{id}_I$ , with  $\langle (), () \rangle = ()$ .

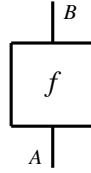
**Eliding redundancies.** There are data services  $A \times A \xleftarrow{\Delta_A} A \xrightarrow{\dagger_A} I$  of type  $A$ , data services  $B \times B \xleftarrow{\Delta_B} B \xrightarrow{\dagger_B} I$  of type  $B$ , and so on for all other types, all of them different, since functions of different types are different, but all of them denoted by  $\Delta$ s and  $\dagger$ s, distinguishing them as the data services. To avoid redundant notations, we omit the type subscripts and superscripts whenever they are clear from the context. E.g.,  $\Delta : A \rightarrow A \times A$  and  $\Delta : B \rightarrow B \times B$  refers to different functions whenever  $A$  and  $B$  are different.

### 1.3 Monoidal functions as boxes

**The 2-dimensional text of string diagrams.** A function  $f : A \rightarrow B$  maps inputs of type  $A$  to outputs of type  $B$ . A process  $g : X \times A \rightarrow B$  is a function that maps causes of type  $A$  to effects of type  $B$ , possibly also depending on states of type  $X$ .

The set-theoretic concept of function is more restrictive. It requires that the possible inputs and outputs are collected in sets and that there is a unique output for every input. Programmers deviated from these requirements early on, and have been using the more relaxed concept of function at least since the 1960s. Theoretical computer scientists deviated even earlier when they developed the  $\lambda$ -calculus as a formal theory of functions that can be fed to themselves as inputs [30]. This seemed like an inexorable property of computable functions, that take their programs as inputs. The task of finding set-theoretic models of such functions was tackled in the denotational semantics of computation [2, 55].

In many programming languages, in type theory, and in category theory, functions are specified as black boxes. The *blackness* does not mean that the boxes are painted black, but that no light shines in or out: the interior of the function box is hidden from the user. A functionality implemented in a black box is accessed through its input and output interfaces, which are drawn as strings. A function  $f : A \rightarrow B$  is drawn as a box  $f$  hanging from a string  $B$  above it, and with a string  $A$  hanging below it:



The inputs flow in from the bottom, and the outputs flow out at the top. This bottom-up processing is a drawing convention. At least four other drawing conventions are equally justified. The idea is that the processed data or resources flow through the strings in one direction, which is the direction of time, and that there are no flows in other directions, or between the strings. The two dimensions of string diagrams thus display where the data flow and where they do not flow. The flow through the strings bottom-up, and they do not flow between the strings.

The two dimensions also correspond to the two ways to compose functions.

#### 1.3.1 Composing functions

The functions  $f$  and  $g$  are composed *sequentially* when the outputs of  $f$  flow as the inputs to  $g$ , provided that output type of  $f$  is the input type of  $g$ . The functions  $f$  and  $t$  are composed *in parallel* when there are no flows between them. In string diagrams, the sequential compositions are thus drawn by hanging the functions  $f$  and  $g$  on one another vertically, whereas the parallel compositions are drawn by placing the functions  $f$  and  $t$  next to one another horizontally. The typing constraints are:

$$\frac{A \xrightarrow{f} B \quad B \xrightarrow{g} C}{A \xrightarrow{g \circ f} C} \qquad \frac{A \xrightarrow{f} B \quad U \xrightarrow{t} V}{A \times U \xrightarrow{f \times t} B \times V}$$

The string diagrams for the sequential composite  $g \circ f$  and for the parallel composite  $f \times t$  are displayed in Fig. 1.7. If a function  $h : C \rightarrow D$  is attached above to the composite  $g \circ f : A \rightarrow C$ , the three boxes hanging on four strings will look the same as if we attached  $f : A \rightarrow B$  below  $h \circ g : B \rightarrow D$ . String diagrams thus impose the associativity  $h \circ (g \circ f) = (h \circ g) \circ f$  as a geometric property of the notation. Similarly, if  $h : C \rightarrow D$  is adjoined to the right of the composite  $f \times t : A \times U \rightarrow B \times V$ , resulting triple composite will forget that  $f$  was adjoined on the left of  $t$  before  $h$  was adjoined on the right. The string diagram notation thus also imposes the associativity  $(f \times t) \times h = f \times (t \times h)$  as a geometric property.

The units for the composition operations are the identity function  $\text{id}_A$  and  $\text{id}_B$  for the sequential composition, and the identity  $\text{id}_I$  on unit type  $I$  for the parallel composition, in the sense of the equations

$$\text{id}_B \circ f \circ \text{id}_A = f = \text{id}_I \times f \times \text{id}_I$$

If the identity functions  $\text{id}_A$  and  $\text{id}_B$  are thought of as the invisible boxes on the strings  $A$  and  $B$ , and the unit type  $I$  is thought of as the invisible string, present in any diagram wherever convenient, then the unitary laws (??) are also imposed in the string-and-box diagrams tacitly, as the geometric properties of the invisible strings and boxes.

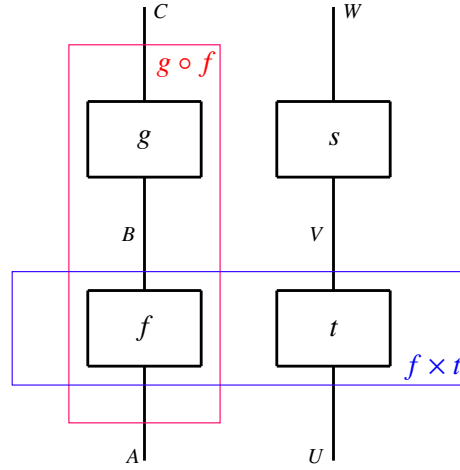


Figure 1.7:  $(g \circ f) \times (s \circ t) = (g \times s) \circ (f \times t)$

A string diagram is read as 2-dimensional text because the strings assure that it can either be read first vertically then horizontally, or first horizontally and then vertically, but not diagonally. The two ways of reading a string diagram impose the algebraic law that constrains the two composition operations:

### 1.3.2 The middle-two-interchange law

A string diagram, as 2-dimensional text, can be read from left to right, or from the bottom to the top. In Fig. 1.7, we can thus choose to first go vertically and compose  $f$  with  $g$  and  $s$  with  $t$  sequentially, and after that to go horizontally and compose the sequential composites  $g \circ f$  and  $s \circ t$  in parallel, to get  $(g \circ f) \times (s \circ t)$ . Alternatively, we can also choose to go horizontally first, and compose in parallel  $f$  with  $t$ , and  $g$  with  $s$ , to get  $f \times t$  and  $g \times s$ , and then we can go vertically, to produce the sequential composite  $(g \times s) \circ (f \times t)$ . These two ways of reading the string diagram in Fig. 1.7 thus correspond to

two different algebraic expressions, but they describe the same composite function:

$$(g \circ f) \times (s \circ t) = (g \times s) \circ (f \times t) \quad (1.3)$$

This is the *middle-two-interchange* law of the function composition algebra. This algebraic law is hard-wired in the geometry of string diagrams. More precisely, the assumption that a string diagram describes a unique function implies the middle-two-interchange law, which identifies the two ways to read a string diagram: first horizontally or first vertically.

### 1.3.3 Sliding boxes

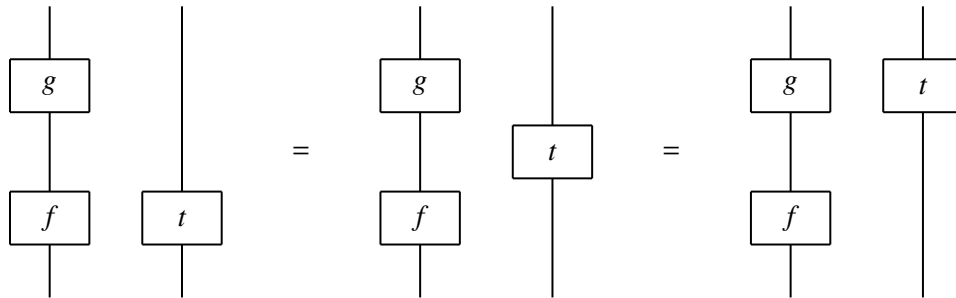
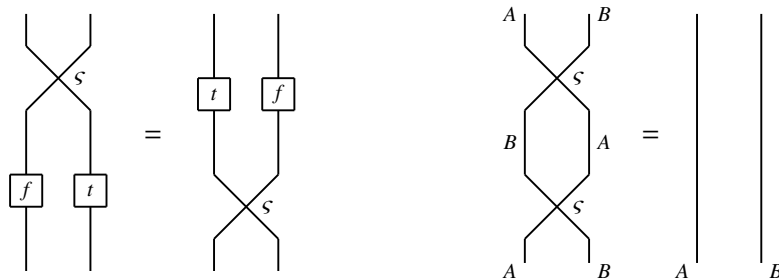


Figure 1.8:  $(g \times \text{id}) \circ (f \times t) = (g \times \text{id}) \circ (\text{id} \times t) \circ (f \times \text{id}) = (g \times t) \circ (f \times \text{id})$

The net effect of encodings of algebra by geometry is that some algebraic correspondences are encoded as geometric properties. E.g., the equations in the caption of Fig. 1.8 can be derived algebraically from the middle-two-interchange law. On the other hand, they can also be derived by sliding boxes along the strings, as displayed in Fig. 1.8. They are thus a geometric property of the string diagram displayed there, as three of its topological deformations. In general, any pair of function compositions, expressed algebraically in terms of  $\circ$  and  $\times$  and the same function boxes will denote the same composite function if and only if they correspond to two string diagrams that can be deformed into each other by some geometric transformations that may twist or extend strings, and slide boxes along them, but without tearing or disconnecting anything.

**Untwisting.** The boxes also slide through the twists, and the twists can be untwisted.



## 1.4 Cartesian functions as boxes with a dot

A function is said to be

- *total* if it produces at least one output for each input, and

- *single-valued* if it produces at most one output for each input.

Formally, a function  $s : A \longrightarrow B$  is total if it satisfies the left-hand equation in (1.4), and single-valued if it satisfies the right-hand equation.

$$\begin{array}{ccc}
 \begin{array}{c} \bullet \\ | \\ \boxed{s} \\ | \\ A \end{array} & = & \begin{array}{c} \bullet \\ | \\ A \end{array} \\
 \downarrow \text{ } \uparrow_B \circ s & & \downarrow \text{ } \uparrow_A
 \end{array}
 \qquad
 \begin{array}{ccc}
 \begin{array}{c} B \quad B \\ \diagdown \quad \diagup \\ \bullet \\ | \\ \boxed{s} \\ | \\ A \end{array} & = & \begin{array}{c} \begin{array}{c} B \\ \bullet \\ \boxed{s} \end{array} \quad \begin{array}{c} B \\ \bullet \\ \boxed{s} \end{array} \\ \diagdown \quad \diagup \\ \bullet \\ | \\ A \end{array} \\
 \Delta_B \circ s & & (s \times s) \circ \Delta_A
 \end{array}
 \tag{1.4}$$

The left-hand equation says that whenever an input  $a$  can be deleted by  $\uparrow_A(a) = ()$  then a total function  $s : A \longrightarrow B$  produces an output  $s(a)$  that can be deleted as  $\uparrow_B(s(a)) = ()$ . The right-hand equation says that cloning the output  $b = s(a)$  by  $\Delta_B(b) = \langle b, b \rangle$  produces the same output like cloning the input  $a$  by  $\Delta_A(a) = \langle a, a \rangle$  and then running two copies of  $s$  in parallel to get  $\langle s(a), s(a) \rangle$ . The two are the same if and only if  $b$  is the only possible value of  $s(a)$ .

A function is **cartesian** when it preserves the cartesian structure, i.e. satisfies (1.4). This means that it is total and single-valued. The black bead on the output port of  $s : A \longrightarrow B$  means that  $s$  is a cartesian function. The usual function notation  $f : A \longrightarrow B$  is left for *monoidal* functions, that do not necessarily satisfy (1.4).

**Elements are the cartesian functions from  $I$ .** For any type  $A$ , an *element*  $a : A$  corresponds to a map  $a : I \longrightarrow A$ . The empty tuple  $() : I$  corresponds to  $\text{id} : I \longrightarrow I$ , which is the only element of  $I$ .

**Every type  $A$  has a unique cartesian function into  $I$ .** The deletion  $\uparrow : A \longrightarrow I$  is obviously total. Showing that it is single-valued is a useful exercise.

**Cartesian functions can be paired.** The copying operation on any type  $X$  induces the pairing of cartesian functions from  $X$

$$\frac{a : X \longrightarrow A \quad b : X \longrightarrow B}{\langle a, b \rangle = \left( X \xrightarrow{\Delta} X \times X \xrightarrow{a \times b} A \times B \right)} \tag{1.5}$$

for all types  $A$  and  $B$ . The deletion operations on  $A$  and  $B$  induce the corresponding projections

$$\frac{h : X \longrightarrow A \times B}{\pi_A h = \left( X \xrightarrow{h} A \times B \xrightarrow{A \times \uparrow} A \right) \quad \pi_B h = \left( X \xrightarrow{h} A \times B \xrightarrow{\uparrow \times B} B \right)} \tag{1.6}$$

The diagrammatic views are displayed in Fig. 1.9. Together, the pairing and the projections provide a bijective correspondence between the pairs of cartesian functions to  $A$  and  $B$  and the cartesian functions to  $A \times B$ . The equations

$$a = \pi_A \langle a, b \rangle \qquad h = \langle \pi_A h, \pi_B h \rangle \qquad b = \pi_B \langle a, b \rangle \tag{1.7}$$

assure that going down (1.5) and then (1.6) returns  $a$  and  $b$ , and that beginning from  $h$  and going down

(1.6) and then (1.5) returns  $h$ . It is not necessary that  $a$ ,  $b$ , and  $h$  are cartesian for  $\langle a, b \rangle$ ,  $\pi_A h$ , and  $\pi_B h$  to be defined, but the equations in (1.7) may not be satisfied. The first equation holds if and only if  $b$  is total, the third if and only if  $a$  is total, and the second equation holds whenever  $h$  is single-valued, but the converse is not always valid. Checking these claims is an instructive exercise. See e.g., [94] for more details.

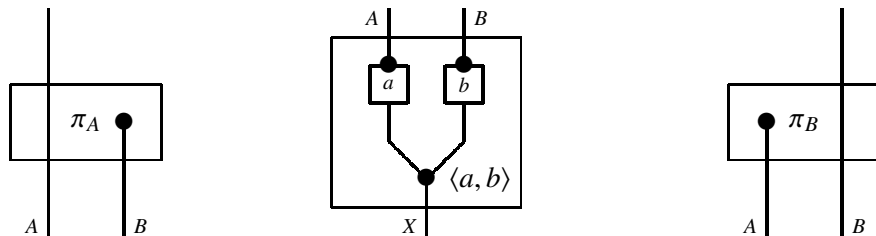


Figure 1.9: Pairing and projections

**Functions with side-effects are not cartesian.** In set-theoretic frameworks, it is usually required that a function  $f$  from  $A$  to  $B$  assigns to every single input  $a : A$  a single output  $f(a) : B$ , and all functions are assumed to be cartesian. When the function  $f$  needs to be computed, such a requirement is hard to satisfy. If a computation of  $f$  on an input  $a$  gets stuck in a loop, it may not produce any output. The left-hand equation in (1.4) then does not hold, because the input can be deleted, but there is no output to delete. A computation of  $f$  may also depend on hidden variables or covert parameters of the environment, and the same overt input may lead to one output today and a different output tomorrow, if the environment changes. Besides the outputs, computations thus often produce some *side-effects*, such as *non-termination* when there is no output, or *non-determinism* when there are multiple possible outputs. That is why **computable functions are generally not cartesian, but monoidal**. That is why the computer is monoidal. The next section spells out what that means. In Sec. 3.2 we shall see that it cannot be made cartesian as long as all of its functions are programmable in it.

**Terminology: partial, many-valued, monoidal functions.** A function that is not total is called *partial*. A function that is not single-valued is called *many-valued*. The functions that may be partial, many-valued, or cause other side-effects, are called *monoidal*. They form *monoidal categories*, described next. This terminology extends to elements, recalling that they are just functions from the unit type  $I$ .

## 1.5 Categories: Universes of types and functions

A *category*  $C$  is comprised of

- a *class of objects*

$$|C| = \{A, B, \dots, \mathbb{P}, \dots, X, Y, \dots\}$$

- for any pair  $X, Y \in |C|$  a *hom-set*

$$C(X, Y) = \left\{ \begin{array}{c} Y \\ \boxed{f} \\ X \end{array}, \begin{array}{c} Y \\ \boxed{g} \\ X \end{array}, \begin{array}{c} Y \\ \boxed{h} \\ X \end{array}, \dots \right\}$$

together with the *sequential composition* operation  $\circ$  from Sec. 1.3.1, presented by linking the boxes vertically along the strings of matching types. If needed, see Appendix 1.1 for more.

**Monoidal categories.** A category is *monoidal* when it also comes with the *parallel composition* operation  $\times$ , presented in Sec. 1.3.1 as the horizontal juxtaposition of strings and boxes next to each other. The monoidal structures are often written in the form  $(C, \times, I)$ , echoing the notation for monoids.

Since drawing the strings  $A$ ,  $B$ , and  $C$  in parallel does not show the difference between  $(A \times B) \times C$  and  $A \times (B \times C)$ , and the string  $A$  does not show how many invisible strings  $I$  are running in parallel with it, the *monoidal structure is in this book always assumed to be strictly associative and unitary*<sup>2</sup>, like monoids, and the types  $(A \times B) \times C$  and  $A \times (B \times C)$  are identical, as are  $A \times I$  and  $A$ . In general, though, monoidal categories are only associative and unitary only up to coherent isomorphisms. Even with string diagrams, the types  $A \times B$  and  $B \times A$  are not identical, but the twistings like in Fig. 1.4 represent the coherent isomorphisms between them. They are assumed to be available for all pairs of types, and the *monoidal structure is in this book always assumed to be symmetric*.

are assumed to be available for any pair of wires. By working with string diagrams, we also tacitly assume that the monoidal structure is *strictly* associative and unitary, since drawing the strings  $A$ ,  $B$ , and  $C$  in parallel does not show the difference between  $(A \times B) \times C$  and  $A \times (B \times C)$ , .

**Data services and cartesian functions form a cartesian category.** The data services in a monoidal category  $C$  distinguish a family of cartesian functions as those that satisfy (1.4). Keeping all of the objects of  $C$  but restricting to the cartesian functions yields the category  $C^\bullet$ , with

$$|C^\bullet| = |C|$$

$$C^\bullet(X, Y) = \left\{ \begin{array}{c} Y \\ \boxed{r} \\ X \end{array}, \begin{array}{c} Y \\ \boxed{s} \\ X \end{array}, \begin{array}{c} Y \\ \boxed{t} \\ X \end{array}, \dots \right\}$$

The pairing and the projections derived from the data services in Sec. 1.4 assure that the products inherited from  $C$  become the cartesian products in  $C^\bullet$ . The details are spelled out in the exercises below.

**Cartesian categories** are the categories where all functions are cartesian. The category  $C^\bullet$  defined above is cartesian by definition. It is the largest cartesian subcategory of the monoidal category  $C$ . The bijective correspondence

$$\frac{a = \pi_A h: X \longrightarrow A \quad b = \pi_B h: X \longrightarrow B}{h = \langle a, b \rangle : X \longrightarrow A \times B} \quad (1.8)$$

induced by (1.5–1.6), shows the universal property of the cartesian products with respect to the cartesian functions, which implies that they are unique up to isomorphism [106, III.4]. The correspondence in (1.8) ceases to be bijective when the functions are not cartesian. Nevertheless, the cartesian structure

<sup>2</sup>This causes no loss of generality since every monoidal category is equivalent to a strict one.

continues to play an important role in the monoidal computer, and at the entire gamut of monoidal frameworks [34].

**Notation.** When the category  $C$  is clear from the context, we write  $f : A \longrightarrow B$  instead of  $f \in C(A, B)$ , and  $s : A \multimap B$  instead of  $s \in C^\bullet(A, B)$ .

**Examples.** Some concrete categories that may be useful and interesting are in Appendix 1.

**Category theory.** The homomorphisms between categories are called *functors*. They preserve the identities and the sequential composition. Appendix 1 provides a very brief summary. The functors between categories with a structure are required preserve that structure: e.g., monoidal categories are connected by monoidal functors, which also preserve the parallel composition and the unit type. The upshot is that categories and the functors between them also form categories, and the structure preserved by the functors determines the structure of the functor categories. The homomorphisms between functors as the objects of functor categories are called the *natural transformations*. They make the family of functors between any fixed pair of categories into a category again, and this makes the category of categories into a 2-category. Sec. 1.7.2 tells a very short story how this came about. A very small sample of books about categories is: [4, 10, 18, 106, 136, 151].

## 1.6 Workout

- a. Given the type  $\mathbb{N}$  of natural numbers (non-negative integers), with the data service as in Sec. 1.2 and the addition and the multiplication operations presented as boxes

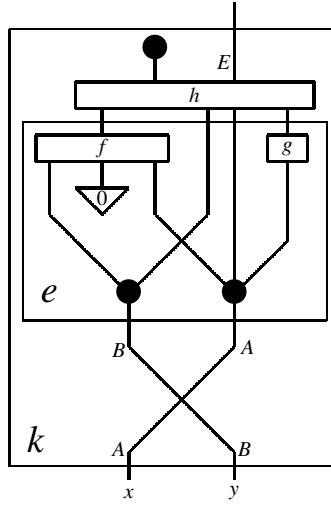


and draw the string diagrams for the following functions:

- i)  $\langle (+), (\cdot) \rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N} : \langle m, n \rangle \mapsto \langle m + n, m \cdot n \rangle$
- ii)  $f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} : \langle m, n \rangle \mapsto m(m + n)$
- iii)  $g : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} : \langle m, n \rangle \mapsto 2mn$
- iv)  $h : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} : \langle m, n \rangle \mapsto n^3$



- b. Translate the following string diagram into the algebraic notation.



(1.9)

Write the function expression in terms of the function names  $f, g, h, \pi_E$ , the constant 0, and the variables  $x, y$ .

- c. Prove that the projections and the pairing displayed in Fig. 1.9 satisfy equations (1.7) for all cartesian functions  $a, b, h$ . More precisely show that

- i) the first equation in (1.7) holds if and only if  $b$  is total,
- ii) the second if  $h$  is single-valued,
- iii) the third if and only if  $a$  is total.

- d. Assuming that the unit type satisfies  $I \times I = I$ , as explained in Sec. 1.1.3, show that the data service  $I \xrightarrow{\uparrow} I \xleftarrow{\Delta} I \times I$  is the identity, i.e.  $\uparrow = \text{id}_I = \Delta$ .

- e. Given the data services  $I \xrightarrow{\uparrow} A \xleftarrow{\Delta} A \times A$  and  $I \xrightarrow{\uparrow} B \xleftarrow{\Delta} B \times B$ , define a data service on  $A \times B$ .

- f. Prove that the copying  $\Delta : A \rightarrow A \times A$  and the deletion  $\uparrow : A \rightarrow I$  are cartesian functions for every type  $A$ .

- g. Show that the composites  $g \circ f : A \rightarrow C$  and  $f \times t : A \times U \rightarrow B \times V$  of cartesian functions  $f : A \rightarrow B$ ,  $g : B \rightarrow C$  and  $t : U \rightarrow V$  are also cartesian.

- h. Prove that the cartesian structure in a category is unique up to isomorphism. Towards this goal, suppose that

- $I$  and  $\tilde{I}$  are objects such that for every object  $X$  there is a unique morphism  $X \rightarrow I$  and a unique morphism  $X \rightarrow \tilde{I}$ ;
- $A \times B$  and  $A \tilde{\times} B$  are objects such that each of them supports the surjective pairing in the sense of the bijective correspondence in (1.8).

Prove that this implies that there are isomorphisms  $I \cong \tilde{I}$  and  $A \times B \cong A \tilde{\times} B$ .

i. Prove that a type  $B^A$  can be determined uniquely up to isomorphism by a family of bijections

$$C(X \times A, B) \stackrel{\varphi_X}{\cong} C(X, B^A)$$

indexed by  $X$  which is *natural* in the sense that for every  $s \in C(Y, X)$  the following diagram commutes:

$$\begin{array}{ccc} C(X \times A, B) & \xrightarrow{\varphi_X} & C(X, B^A) \\ \downarrow C(s \times A, B) & & \downarrow C(s, B^A) \\ C(Y \times A, B) & \xrightarrow{\varphi_Y} & C(Y, B^A) \end{array}$$

In other words, every  $g \in C(X \times A, B)$  satisfies

$$\varphi_Y \left( Y \times A \xrightarrow{s \times A} X \times A \xrightarrow{g} B \right) = \left( Y \xrightarrow{s} X \xrightarrow{\varphi_X(g)} B^A \right)$$

## 1.7 Stories

### 1.7.1 Type by any other name

The concept of type is usually attributed to Bertrand Russell. He introduced a type hierarchy as means of preventing set-theoretic paradoxes in [140]. Lord Russell discovered the paradox that carries his name, here presented in Appendix 3, by applying Cantor's diagonal argument to the element relation used in Frege's arithmetic. Russell's type hierarchy, like von Neumann's later distinction of sets and classes, was ostensibly based on Cantor's distinction of "consistent" and "inconsistent" sets [22, 40, 115]. Even Russell's use of the word "type" may have been inspired by Cantor, who studied what would now be called the *isomorphism classes* of ordered sets under the name "*order types*".

The central idea of Cantor's work, which made possible his breakthrough into the theory of infinities [40, 69], was to consider sets modulo bijections, to compare them by specifying injections [22], and to compare their orders by specifying order-preserving injections [23, III.9]. Order types are the equivalence classes of orders modulo the order-preserving bijections. E.g., identifying the orders  $\{a < b < c\}$ ,  $\{x < y < z\}$ , and all other 3-element linear orders, whatever their elements might be, was the order type of the ordinal 3, whereas identifying the sets  $\{a, b, c\}$ ,  $\{x, y, z\}$ , and all other 3-element sets, made them into the cardinal 3. Cantor's view of cardinals as the "*virtual patterns*" of equivalent sets [23, p. 283] is a precursor of the general concept of type. The notion that the equivalence class of all 3-element linear orders and the equivalence class of all 3-element sets are not sets but proper classes did not yet exist, since the notion of a proper class did not exist, but Cantor gleaned the diagonal argument over the element relation that became Russell's paradox, and isolated the "inconsistent" sets. Some 35 years and many paradoxes later, Zermelo, as the editor of Cantor's collected works, [23] systematically erred on the side of declaring as Cantor's error everything that he did not understand, as illustrated, for instance, by his informal argument that Cantor's informal argument could not be salvaged by Cantor's

well-ordering assumptions, but only by Zermelo’s own Axiom of Choice, “which Cantor uses unconsciously and instinctively everywhere, but does not formulate explicitly anywhere” [23, p. 451]<sup>3</sup> By the time of Cantor’s death in 1918, his foundational quest for “virtual patterns” modulo bijections got overshadowed by the research opportunities in the areas that it created, including transfinite arithmetic, point-set topology, measure theory. In the 1940s, the type formalism re-emerged in Church’s work as the framework of function abstraction [29]. We return to this in Sec. 4.6. By the 1960s, type declarations became a standard preamble in the program templates of high-level programming languages. N.G. de Bruijn merged the theory and the practice of typing in his Automath Project [113], which gave a lasting impetus to type-based programming [12, 56, 77, 108, 116].

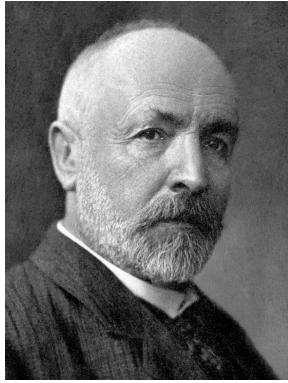


Figure 1.10: Georg Cantor



Figure 1.11: F. William Lawvere

### 1.7.2 Categories as type universes

While Cantor’s transfinite arithmetic was already studied in his lifetime, and celebrated shortly after his death<sup>4</sup>, his enabling idea, to identify sets along bijections and compare them along injections, was used as a tool but didn’t attract much attention as a concept — until it re-emerged several decades later, in an unrelated area, for unrelated reasons. The theory of algebraic invariants of topological spaces had in the meantime grown into a sweeping research movement, adding layer upon layer of structure to its analyses. To dam the flood of structure, it was often necessary to hide the irrelevant details of its constructions into black boxes and display just the relevant operations on the interfaces. The separation of the relevant structure from the irrelevant is usually enforced by the homomorphisms, which preserve one and filter out the other. Just like Cantor identified different sets along bijections, Samuel Eilenberg and Saunders MacLane identified different mathematical constructions of the same structure along the isomorphisms between the outcomes. They proposed that a family of isomorphisms between the pairs of outcomes of two constructions, indexed over the inputs fed into each construction, should be viewed as a *natural equivalence* of the two constructions, provided that the isomorphisms preserve not only the relevant structure, but also commute with the homomorphisms between the inputs. Many results, old and new, turned out to rely upon such identifications. By dropping the invertibility requirement from their natural equivalences and retaining the structure-preservation requirements, Eilenberg and MacLane also defined a more general concept of a *natural transformation* between mathematical constructions, [46, 94, Sec. 0.2], giving rise to a mathematical theory of mathematical constructions: the category the-

<sup>3</sup>This English translation is quoted from [67, p. 129]. For the broader context of the Cantor-Dedekind correspondence, see [115], prepared by Emmy Noether. Both Cantor’s method of well-ordering and the induced choice functions are used as programming tools in Ch. 7.

<sup>4</sup>“No-one shall expel us from Cantor’s Paradise”, exclaimed Hilbert in his 1925 article *On the Infinite* [69]

ory. The mathematical constructions themselves were formalized as *functors*, acting on categories as universes of structures. Some examples of categories can be found in Appendix 1, some functors are worked out in 2.6.3, and many textbooks and handbooks provide thorough expositions [10, 18, 106]. The central feature of category theory is that the relevant structures are captured as what the homomorphisms preserve, whereas the objects are used as black boxes, to hide the irrelevant structures. Beyond mathematics, similar or related treatments, with black-boxes and interfaces as first-class citizens, also evolved in software engineering as procedural, component-oriented, modular, object-oriented programming, as well as in other areas of engineering, science, philosophy, and art. As a tool for comprehending logical abstractions, and for damming the floods of structure, the type abstraction appears in so many avatars that it is needed even to comprehend itself.

### 1.7.3 Logics of types

By the 1960s, categories caught root in several burgeoning areas of mathematics [47, 62, 84, to name a few]. Sets, on the other hand, were settled as the accepted foundation of nearly all areas. So when Bill Lawvere proposed that categories could be used as a foundation of mathematics, abstracting away the elements from the sets, neither his thesis advisor Samuel Eilenberg, nor the other father of category theory, Saunders MacLane, nor pretty much anyone else, could comprehend what could possibly be meant or done with Lawvere’s “*sets without elements*”. Lawvere then narrowed his focus and wrote a thesis reducing to functors and categories just the universal algebra, not all of mathematics [96]. The reduction turned out to be so simple and effective that his broader foundational effort could not be denied a second chance. Within a couple of years, his analysis of “The category of categories as a foundation for mathematics” [97] appeared as the introductory article in a volume coedited by Eilenberg and MacLane.

Truth be told, Lawvere was not proposing sets without elements. Categories themselves are specified in terms of elements: the elements of hom-sets, the elements of the object class, and so on. Types also usually have some elements, even if they may change in various ways. A datatype in a software application may contain one set of elements today, another one tomorrow, while remaining the same datatype. Types and categories are not characterized by their elements but by their structures. Two different types, or two different categories, may have the same elements, but different structures. The category of finite relations and the category of finite-dimensional  $\mathbb{Z}_2$ -vector spaces have the same set of objects, say  $\mathbb{N} = \{0, 1, 2, \dots\}$ , and for every pair  $m, n \in \mathbb{N}$  the same morphisms  $u : m \rightarrow n$ , presented, say, as matrices of 0s and 1s. But the composition of matrices  $u = (u_{ji})_{n \times m}$  and  $v = (v_{kj})_{p \times n}$  in the category of relations has the  $ki$ -th entry  $\bigvee_{j < n} (v_{kj} \wedge u_{ji})$ , whereas in the category of  $\mathbb{Z}_2$ -vector spaces it is  $\sum_{j < n} (v_{kj} \cdot u_{ji})$ . So the two categories are quite different<sup>5</sup>. The set of injections from  $\{a, b\}$  to  $\{c\}$  and the set of surjections from  $\{c\}$  to  $\{a, b\}$  are identical as sets since they are both empty; but viewed as types, they become different, since checking whether a function is an injection from  $\{a, b\}$  to  $\{c\}$  is different from checking whether it is a surjection from  $\{c\}$  to  $\{a, b\}$ . The propositions  $(A \wedge \neg A)$  and  $(A \Leftrightarrow \neg A)$  are both false, but the proofs that they are false are different, and they correspond to different types.

#### 1.7.3.1 Propositions-as-types

The paradigm of propositions-as-types, straddling logic and type theory, goes back to Brouwer-Heyting-Kolmogorov interpretation of *proofs-as-constructions* in the 1920s [166, Ch. 1, Sec. 5], and to Curry’s observation that the implication  $A \supset B$  can be construed as functions  $A \rightarrow B$  in 1934 [39]. It is also related to Kleene’s 1945 untyped realizability of all constructive logical connectives by functions [90],

<sup>5</sup>Seen as a logical operation, the  $\mathbb{Z}_2$ -product is still the conjunction, but the  $\mathbb{Z}_2$ -sum is the *exclusive or*.

and more closely to Kreisel's 1957 typed realizability [93, footnote]. It was formalized in Howard's 1969 manuscript, published in 1980, and adopted as a logical and programming principle under the name *Curry-Howard isomorphism* [38, 57, 123, 124, 175]. But while the focus of efforts among the logicians was to spell out the correspondence of certain proof-theoretic and type-theoretic operations, Lawvere from a different direction arrived at the sweeping claim, and compelling evidence, presented in [98], that all fundamental constructions, logical and type-theoretic, were instances of a single universal categorical structure: the adjunction — originating in homotopy theory [84]. Although the claim had little traction among logicians, with the benefit of hindsight it is clear that Lawvere's approach presented a radical departure from the entire foundational tradition arising from Russell-Whitehead's *Principia* [141] and pursued in mathematical logic. *While logicians reconstructed mathematical structures from logical foundations, Lawvere was reconstructing logical foundations from mathematical structures.*

For concreteness and future reference, I illustrate the approach by aligning the rules defining basic logical operations with the correspondences (easily recognized as adjunctions) defining basic categorical constructs, leading up to the structure of *cartesian-closed category*. The double lines denote two-way (introduction-elimination) rules and bijective correspondences.

**Entailment is composition.** The basic rules of logical entailment correspond to the basic categorical operations, sequential composition and the identities:

$$\begin{array}{c}
 \frac{}{A \vdash A} \qquad \frac{}{A \xrightarrow{\text{id}} A} \\
 \\
 \frac{A \vdash B \quad B \vdash C}{A \vdash C} \qquad \frac{A \xrightarrow{f} B \quad B \xrightarrow{g} C}{A \xrightarrow{g \circ f} C}
 \end{array} \tag{1.10}$$

The unitarity and the associativity of the preorder on the left are automatic, but the categorical signature requires explicit equations:

$$\text{id}_B \circ f = f = f \circ \text{id}_A \qquad h \circ (g \circ f) = (h \circ g) \circ f$$

**Conjunction is the cartesian product.**

$$\begin{array}{c}
 \frac{}{A \vdash \top} \qquad \frac{}{A \xrightarrow{\top} \bullet I} \\
 \\
 \frac{X \vdash A \quad X \vdash B}{X \vdash A \wedge B} \qquad \frac{X \xrightarrow{a} \bullet A \quad X \xrightarrow{b} \bullet B}{X \xrightarrow{\langle a, b \rangle} \bullet A \times B}
 \end{array} \tag{1.11}$$

with the equations:

$$(\text{id}_A \times \top) \circ \langle a, b \rangle = a \qquad (\top \times \text{id}_B) \circ \langle a, b \rangle = b \qquad \langle (\text{id}_A \times \top), (\top \times \text{id}_B) \rangle = \text{id}_{A \times B}$$

**Disjunction is the coproduct.**

$$\begin{array}{c}
 \frac{}{\perp \vdash A} \qquad \frac{}{O \multimap A} \\
 \\
 \frac{A \vdash Y \quad B \vdash Y}{A \vee B \vdash Y} \qquad \frac{A \xrightarrow{\alpha} Y \quad B \xrightarrow{\beta} Y}{A + B \xrightarrow{[\alpha, \beta]} Y}
 \end{array} \tag{1.12}$$

with the equations:

$$[\alpha, \beta] \circ (\text{id}_A + \downarrow) = \alpha \qquad (\downarrow + \text{id}_B) \circ [\alpha, \beta] = \beta \qquad [(\text{id}_A + \downarrow), (\downarrow + \text{id}_B)] = \text{id}_{A+B}$$

**Implication is the right adjoint.**

$$\frac{X \wedge A \vdash B}{X \vdash A \supset B} \qquad \frac{X \times A \xrightarrow{g} B}{X \xrightarrow{\lambda A. g} B^A} \tag{1.13}$$

with a unique  $\varepsilon : B^A \times A \multimap B$  satisfying

$$\varepsilon \circ (\lambda A. g \times \text{id}_A) = g \qquad \lambda A. \varepsilon = \text{id}_{B^A}$$

### 1.7.3.2 Cartesian closed categories

A category is cartesian if it has the final object and the cartesian products  $A \times B$  for all types  $A, B$ , as in (1.5–1.6) and (1.11). It is closed with respect to the cartesian products if it has the exponents  $B^A$ , as in (1.13). Since the function abstraction into an exponent corresponds to the  $\lambda$ -abstraction, the cartesian closed structure presents a categorical view of Church’s simple theory of types [29], *and* of the constructivist propositional logic formalized by Heyting algebras [68]. The categorical and the type-theoretic presentations differ only in syntactic details [94, Part I] and the terminology is often mixed<sup>6</sup>. The cartesian-closed categories are ubiquitous and central in many areas, from topos theory [80] to denotational semantics [64]. In Ch. 7, we explore a version suitable for programming: the *program-closed* categories.

## 1.7.4 Categorical diagrams: chasing arrows and weaving strings

Diagrammatic reasoning was at the heart of ancient mathematics. Archimedes apparently died defending the circles he drew in the sand on a beach, studying a mathematical problem. On the other hand, an entire line of religious and political movements found reasons to prohibit all visual depictions in general and ungodly diagrammatic witchcraft in particular. Science did not prohibit depictions, but printing made them costly, and mathematical narratives came to be dominated by formulas and the *algebraic* reasoning, even about geometry. In a similar way, programming came to be dominated by the *command-line* reasoning, fitting the 80-character command lines of punch cards and terminal displays. After the invention of the computer mouse, the cursor movements spanned the 2-dimensional space of

<sup>6</sup>In [98], Lawvere calls “types” what most category-theorists now call “objects”. We call them types again.

the computer screen. It may not be Archimedes' beach, but it made drawing diagrams easier.

### 1.7.4.1 Arrow diagrams

Computers also made printing diagrams easier, just in time, since the algebraic reasoning about geometric objects had by then developed its own geometric patterns. From the outset, the categorical abstractions were illustrated by *arrow diagrams*. An example is in the upper part of Fig. 1.12. The function  $q$  can be thought of as a state machine over the state space  $X$  with inputs of type  $A$  and outputs of type  $B$ . It is in fact a pair of functions  $q = \langle q^\triangleright, q^\triangleleft \rangle$  where  $q^\triangleright : X \times A \rightarrow X$  maps an input  $a : A$  at the state  $x : X$  to the *next state*  $q^\triangleright(x, a) : X$ , whereas  $q^\triangleleft : X \times A \rightarrow B$  maps them to the *output*  $q^\triangleleft(x, a) : B$ . If the function  $r$  is a state machine over the state space  $Y$ , then a state assignment  $s : X \rightarrow Y$  is a *simulation* of the machine  $q$  by the machine  $r$  at every state  $x : X$  if for every input  $a : A$  the next state of  $s(x) : Y$  by  $r$  is the  $s$ -image of the next state of  $x$  by  $q$ , and the outputs coincide. The requirement is thus

$$r^\triangleright(s(x), a) = s(q^\triangleright(x, a)) \quad r^\triangleleft(s(x), a) = q^\triangleleft(x, a)$$

In the arrow diagram in Fig. 1.12, this requirement means that the path going left is equal to the path going right. The corresponding string diagram equation is displayed below the arrow diagram. The

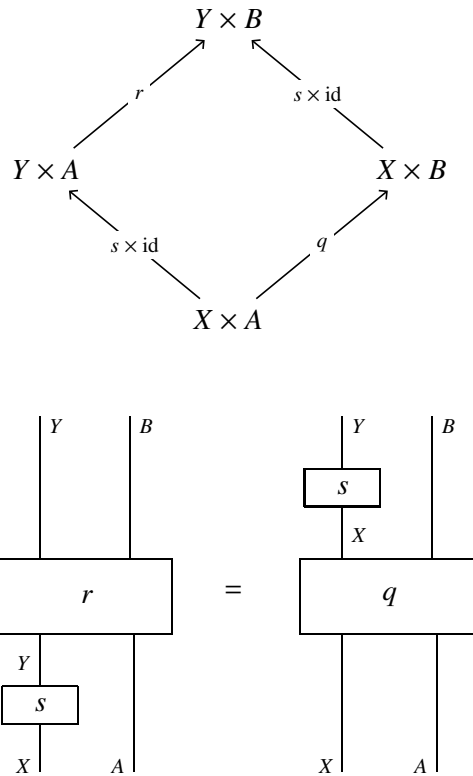


Figure 1.12: The equations  $r^\triangleright(s(x), a) = s(q^\triangleright(x, a))$  and  $r^\triangleleft(s(x), a) = q^\triangleleft(x, a)$  as diagrams

technique of *diagram chasing* allows encoding a lengthy equational derivation in a single diagram tiled by polygons spanned by equal arrow paths [105, Ch. XII, Sec. 3]. Such tilings of equations into diagrams for chasing are useful both for building equational proofs and for communicating them. In his

monograph presenting categories to working mathematicians [106], MacLane consistently called homomorphisms between mathematical objects the *arrows*, reminding the readers that the diagrams are there to be chased. As a procedure based on identifying equivalent paths between pairs of points, the method of diagram chasing has been construed as an echo of the mathematics of homotopies in the metamathematics of arrows.

**We call arrows functions and objects types.** In the terminological soup of categories and computations in this book, MacLane’s “arrows” and “objects” clash against too many different meanings assigned to these terms by programmers and computer scientists. Moreover, there are just a few arrow diagrams in this book, and no real objects. While diagrammatic reasoning remains at the heart of programming and computation, string diagrams give us something that arrow diagrams don’t.

#### 1.7.4.2 String diagrams

While a single arrow diagram displays as many algebraic expressions as there are paths through it, and summarizes as many equations as there are path-reroutings (*viz* faces of the underlying graph), a string diagram displays a single algebraic expression and representing algebraic equations requires equations between string diagrams. The geometric overview of equational derivations, that was provided by the arrow diagrams, is lost. String diagrams, however, lay out the geometry of the algebraic expressions themselves. The upshot is that many abstract, often complex algebraic transformations are turned into simple geometric transformations. One example was sliding the boxes in Fig. 1.8. In some cases, like the middle-two-interchange law in Fig. 1.7, nontrivial algebraic equations are completely phased out by assigning the same diagrammatic interpretation to different algebraic expressions. The programming adventures in the forthcoming chapters will illustrate the utility of such geometric shortcuts.

String diagrams were invented by the physicist Roger Penrose, as a tool for tracing indices through tensors, and independently by Günter Hotz, for tracking values through boolean circuits. Penrose’s paper [130] is usually cited as the earliest reference, but Hotz’s thesis [75] appeared earlier, in an explicitly categorical framework. But Gavin Wraith attested in private conversations that Penrose was seen using his string diagrams to avoid Einstein’s index conventions already in graduate school in the late 1950s. He had apparently refrained from presenting them in publications mainly to avoid printing constraints. Be it as it may, the formal presentation of the language of string diagrams was provided only in the work of André Joyal and Ross Street that appeared in print in 1991 [82]. That work was completed in the unpublished but widely available draft [83]. Ross Street also provided a historic account in a public posting [163]. As computers further facilitated drawing and printing, string diagrams became a standard part of the categorical toolkit [11, 35, 71, and hundreds of other citations].

**What is the relation between string diagrams and arrow diagrams?** A 2-category is a category where the hom-sets are categories. Besides the usual arrows between objects (which we call types in this book), there are thus also 2-arrows between arrows, like in the upper part of Fig. 1.13. An example of a 2-category is the category of categories, whose objects are categories, the arrows are functors, and the 2-arrows are the natural transformations. A face of an arrow diagram in a 2-category is not just a hollow polygon spanned by two arrow paths that may be equal or not. The polygon may contain a 2-arrow. The graph of the diagram thus consists of 0-cells as vertices, 1-cells as edges, and 2-cells that fill the faces of the graph. A string diagram in a 2-category is what graph theorists call a *Poincaré dual* of the arrow diagram: the 2-cells of the arrow diagram are the 0-cells of the string diagram, whereas the 0-cells of the arrow diagram (the objects of the 2-category) become the 2-cells of the string diagram. The string diagram in the lower part of Fig. 1.13 is obtained by drawing a string across each arrow of the arrow



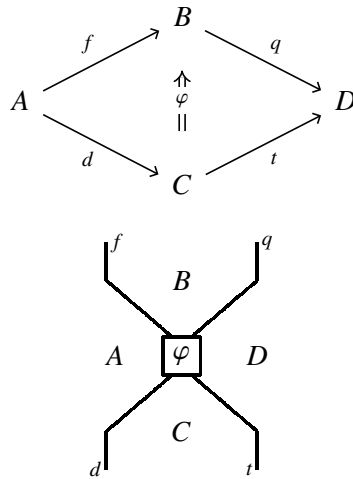


Figure 1.13: A 2-arrow in a 2-categorical arrow diagram is a box in a 2-categorical string diagram

diagram, and a box in each 2-cell of the arrow diagram, to connect the strings coming into it.

But there are no 2-categories in this book. Only monoidal and cartesian categories. Where do our string diagrams come from? A monoidal category can be viewed as a 2-category with a single 0-cell. The objects of the monoidal category are the 1-cells of the 2-category, whereas the arrows of the monoidal category become the 2-cells of the 2-category. Using the 2-categorical arrow diagrams and drawing the objects of the monoidal category as arrows and the arrows between them as more arrows is generally not very useful; but using their dual string diagrams and drawing the objects of the monoidal category as strings, the arrows as boxes, is generally very useful.

20221212

## 2 Monoidal computer: computability as a structure

---

2.1	Computer as a universal machine . . . . .	26
2.2	Running as composition . . . . .	27
2.2.1	Program evaluators . . . . .	27
2.2.2	Partial evaluators . . . . .	29
2.2.3	Composing programs . . . . .	30
2.3	Programs $\subseteq$ Data $\subseteq$ Programs . . . . .	31
2.3.1	Data are programs . . . . .	31
2.3.2	Types are idempotents . . . . .	32
2.3.3	Pairs of programs are programs . . . . .	33
2.3.4	Programming languages encode each other . . . . .	33
2.3.5	Idempotents split . . . . .	34
2.4	Logic and equality . . . . .	35
2.4.1	Truth values and branching . . . . .	35
2.4.2	Program equality . . . . .	35
2.4.3	The type of booleans . . . . .	35
2.4.4	Propositional connectives . . . . .	36
2.4.5	Predicates and decidability . . . . .	36
2.5	Monoidal computer . . . . .	37
2.5.1	Definition . . . . .	37
2.5.2	Examples . . . . .	38
2.5.3	Program evaluation as natural surjection . . . . .	38
2.6	Workout . . . . .	39
2.6.1	How many programs? . . . . .	39
2.6.2	Counting by evaluating: Church numerals . . . . .	40
2.6.3	Monoid computer? . . . . .	41
2.7	Stories . . . . .	46
2.7.1	Who invented the computer? . . . . .	46
2.7.2	How many truth values? . . . . .	48

---

## 2.1 Computer as a universal machine

Machines! We live with machines. Fig. 2.1 shows some friendly machines that implement some useful functions, displayed with their input type strings coming in at the bottom and the output type strings coming out at the top. A mechanical calculator inputs pairs of numbers and outputs the results of some

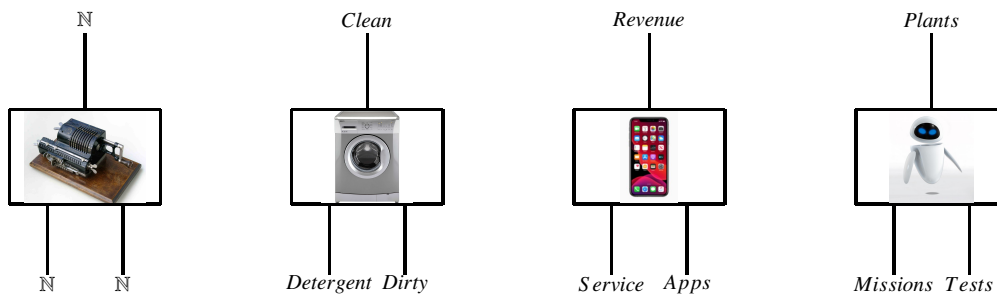


Figure 2.1: Different machines for different purposes

arithmetic operations. A washing machine applies its *Detergent* contents to transform the *Dirty laundry* inputs into *Clean laundry* outputs. A smartphone inputs the *Apps* approved by its *Service* and outputs the *Revenue* for the provider. And finally EVE, the Extraterrestrial Vegetation Evaluator from Pixar's movie WALL-E, applies the *Tests* as prescribed for each of its *Missions*, and it delivers any detected *Plants* back to its spaceship.

One thing that all these machines have in common is that they are all different. Each function requires a different machine. Each machine is a special-purpose machine. The special purpose of a calculator is to calculate a special set of arithmetic operations. When we need to calculate different operations, we need to build a different calculator.

A computer is a universal machine: it can be programmed to do anything that any other machine can do. This is what makes the process of computation into a conversation between functions and programs like in Fig. 3. Some examples, initiated by the functions in Fig. 2.1, are displayed in Fig. 2.2. A general-purpose computer can perform the builtin arithmetic operations of a special-purpose calculator. A smartphone is a slightly more complicated case. It is a computer in jail. On one hand it is definitely a universal computer. On the other hand, some smart people figured out a way how to prevent the smartphone from running programs unless you pay them. Some computational processes are thus driven by money. Others are driven by love. Robot WALL-E is tasked with collecting and compacting trash on the trashed planet Earth. To be able survive there, he is also equipped with a general-purpose computer. If needed, he can thus also be programmed to do other things. If needed, he can be programmed to wash laundry. Or he can program himself to perform EVE's function, i.e. to seek out and deliver a plant. We shall see in Ch. 6 how computers compute certain programs. If they then run the programs that they have computed, then they have programmed themselves. Computers and humans are similar.

Computers and people are similar in that each of them can do anything that any other can do. That is what makes them universal. How do they achieve that? How can it be that one can do anything that any other can do? For the humans, the universality has been a matter of some controversy, and it may seem complicated. For computers it is very simple. The functions that the computers can compute are precisely those that can be described by programs. **Computability is programmability.**

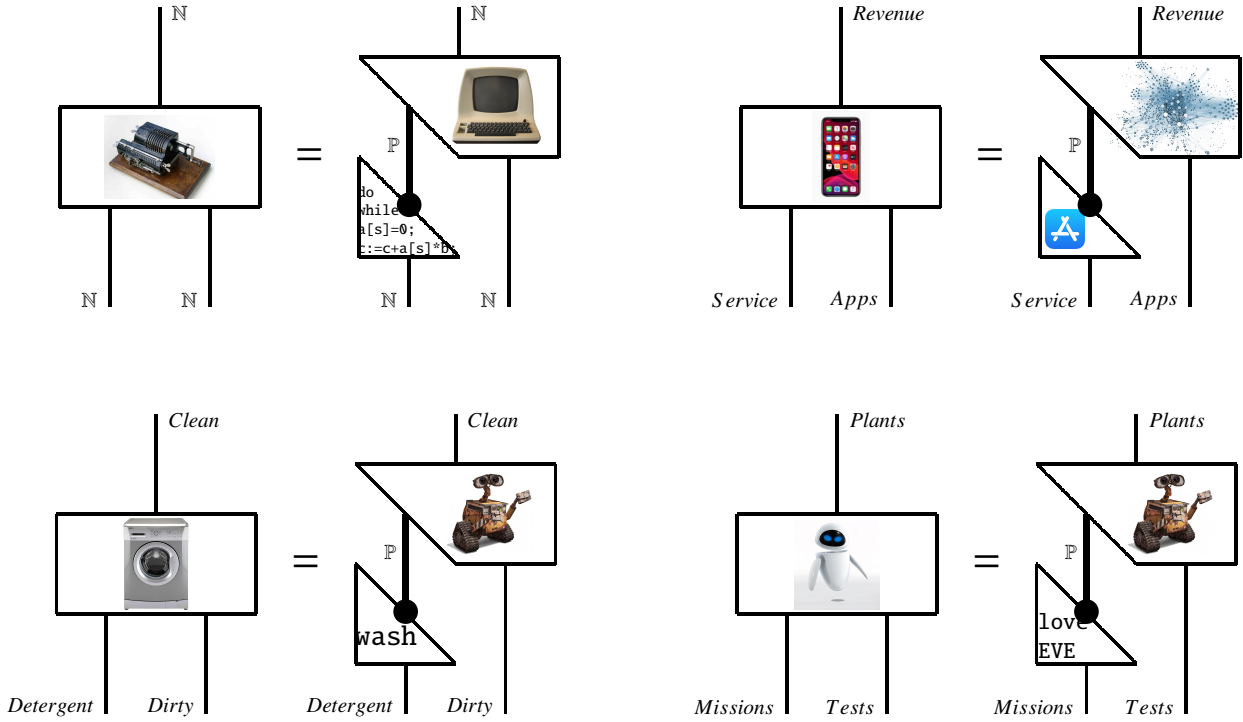


Figure 2.2: Universal machines can do anything that any other machines can do

## 2.2 Running as composition

A categorical model of a computer is a category where every function is computable, in the sense that there is a program for it, and a program evaluator evaluates that program to the function. The programs are elements of the distinguished program type  $\mathbb{P}$ . The program evaluators are functions of type  $\mathbb{P} \times A \rightarrow B$ , one for each pair of types  $A, B$ . The type  $\mathbb{P}$  can be thought of as a programming language and the program evaluators as interpreters.

### 2.2.1 Program evaluators

We view a computer as a monoidal category  $\mathcal{C}$  with data services and a distinguished program type  $\mathbb{P}$  equipped with a universal *program evaluator*  $\{\}_A^B: \mathbb{P} \times A \rightarrow B$  for every pair of types  $A, B$ . A program evaluation is the composite function

$$\{G\}_A^B = \left( X \times A \xrightarrow{G \times A} \mathbb{P} \times A \xrightarrow{\{\}} B \right) \quad (2.1)$$

where  $G: X \rightarrow \bullet \mathbb{P}$  is an  $X$ -parametrized family of programs. Such families can be thought of as programs with parameters that need to be instantiated before they are run. Such programs are often called *polymorphic*. The type  $X$  of parameters can also be thought of as a state space. The function  $G: X \rightarrow \bullet \mathbb{P}$  is cartesian because it must determine a unique program for each state of the world  $x: X$ . The program evaluator  $\{\}$  is the run-instruction mentioned in the introduction. We write  $\{G\}$  instead of  $\text{run}(G)$  to be able to write  $\{G\}x$  instead of  $\text{run}(G)(x)$ . The story how this notation evolved is in Sec. 4.6.

The defining property of program evaluators is that they are *universal*. The universality of the program evaluator  $\{\}_A^B$  in a category  $\mathcal{C}$  means that for every function  $g \in \mathcal{C}(X \times A, B)$  there is a program  $G \in \mathcal{C}^\bullet(X, \mathbb{P})$  that evaluates to it:

$$g(x, a) = \{Gx\}a \quad (2.2)$$

This requirement captures the idea from Fig. 3, that composing with the instruction  $\text{run}_A^B = \{\}_A^B$  induces a surjection  $\mathcal{C}^\bullet(X, \mathbb{P}) \twoheadrightarrow \mathcal{C}(X \times A, B)$ . Writing the  $X$ -state indices as subscripts, (2.2) says:

$$\forall g: X \times A \longrightarrow B \quad \exists G: X \longrightarrow \bullet \mathbb{P} \quad \forall x: X \quad \forall a: A. \quad \{G_x\}a = g_x(a) \quad (2.3)$$

A stronger notion of universality is analyzed in Sec. 2.6.3.

**Examples.** The interpreter of an actual programming language is an instance of a program evaluator, provided that the language is Turing-complete, i.e. that it is expressive enough to allow programming its own interpreter. See Sec. 4.6. The type  $\mathbb{P}$  is the set of expressions of the language. Quantum computers, biological computers, and other nonstandard models of computation, also contain program evaluators for their nonstandard language. The *executable* software specification languages [128] and some abstract interpretation tools [19] also support program evaluators.

**States vs inputs.** Programs in principle describe how the outputs of a computation depend on its inputs. In many cases, though, a computation does not depend only on the inputs, but also on external factors. E.g., a payroll system may need to perform different computations in different pay periods. A mobile application performs different computations on different devices. Most software products run in different computational environments, be it operating systems, hardware platforms, etc. Such external factors are what we call the *states of the world*. They are represented by the program parameters. While the program variables are written and overwritten internally during the program evaluations, the program parameters are set externally. If a family of computations  $g_x: A \longrightarrow B$  depends on the states  $x: X$ , the corresponding programs  $G_x: \mathbb{P}$  satisfying (2.3) are parametrized accordingly. The parametrizations are assumed to be computable, in the sense that a parametric family of computations can be viewed as a single computation  $g: X \times A \longrightarrow B$ , and the corresponding parametric family of programs as a cartesian computation  $G: X \longrightarrow \bullet \mathbb{P}$ . The distinction between the parameters and the inputs of a computation is not an intrinsic property of their values, but a design decision of the programmer. Formally, any type can play the role of a state space or of an input source and can change the role. E.g., a function  $f: A \longrightarrow B$  can be viewed in either of the forms

- a)  $f: I \times A \longrightarrow B$ , i.e. as a family  $\{f_0: A \longrightarrow B \mid () : I\}$  trivially indexed over the state space  $I$ , or
- b)  $f: A \times I \longrightarrow B$ , i.e. as a family  $\{f_a: I \longrightarrow B \mid a: A\}$  of trivial functions indexed over the state space  $A$ .

Under interpretation (a), condition (2.2) induces a fixed program  $F$  such that  $\{F\} = f : A \longrightarrow B$ , always in the same state of the world  $() : I$ . Under interpretation (b), the same condition induces an  $A$ -indexed family of programs  $\{\Phi_a : \mathbb{P} \mid a : A\}$ , taking the trivial input  $() : I$  and producing the corresponding  $A$ -indexed family of constants  $\{\Phi_a\} () = f_a : B$ , as displayed in Fig. 2.3.

$$f(a) = \{\Phi_a\} () = \{F\} (a)$$

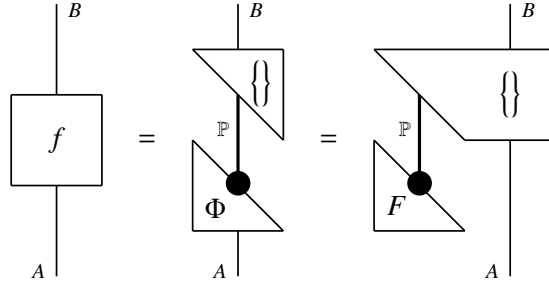


Figure 2.3:  $F$  encodes the function  $f = \{F\}$ . Family  $\Phi_a$  encodes the family  $f_a = \{\Phi_a\}$ .

**Notation.** Writing the states as subscripts is a matter of convenience and completely informal. Formally, the state and the input arguments of a function obey the same rules. They are only distinguished by the external meaning that the programmers assign to them. In the current presentation, writing some arguments as subscripts signals that their values may be available earlier than the values of other arguments, and that can the computations can be *partially* evaluated.

## 2.2.2 Partial evaluators

Setting  $X$  in (2.2) to be a product  $\mathbb{P} \times Y$  and  $g$  to be the program evaluator  $\{\} : \mathbb{P} \times Y \times A \longrightarrow B$  gives a *partial evaluator*  $[] : \mathbb{P} \times A \longrightarrow \bullet \mathbb{P}$  for any  $Y, A, B$ , as displayed in Fig. 2.4. A partial evaluator is thus a  $(\mathbb{P} \times Y)$ -indexed program satisfying  $\{[\Gamma] y\} a = \{\Gamma\} (y, a)$ ,

$$\{\Gamma\} (y, a) = \{[\Gamma] y\} a$$

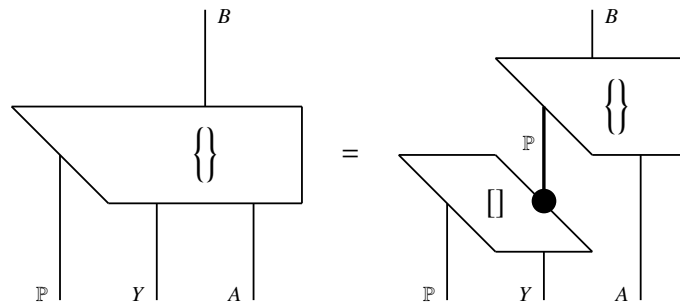


Figure 2.4: Partial evaluators  $[]$  are indexed programs

**Indexed = Fixed + Partial.** Any  $X$ -indexed program for a computable function  $g : X \times A \longrightarrow B$  in (2.2) can be construed as a partial evaluation of a fixed program  $\Gamma$  on data  $x : X$  which may happen to

be available before the data of type  $A$ . When that happens,  $x$  can be treated as a state of the world for the computation  $g_x = \{\Gamma_x\} : A \rightarrow B$ . This view of indexed computations is displayed in Fig. 2.5. In

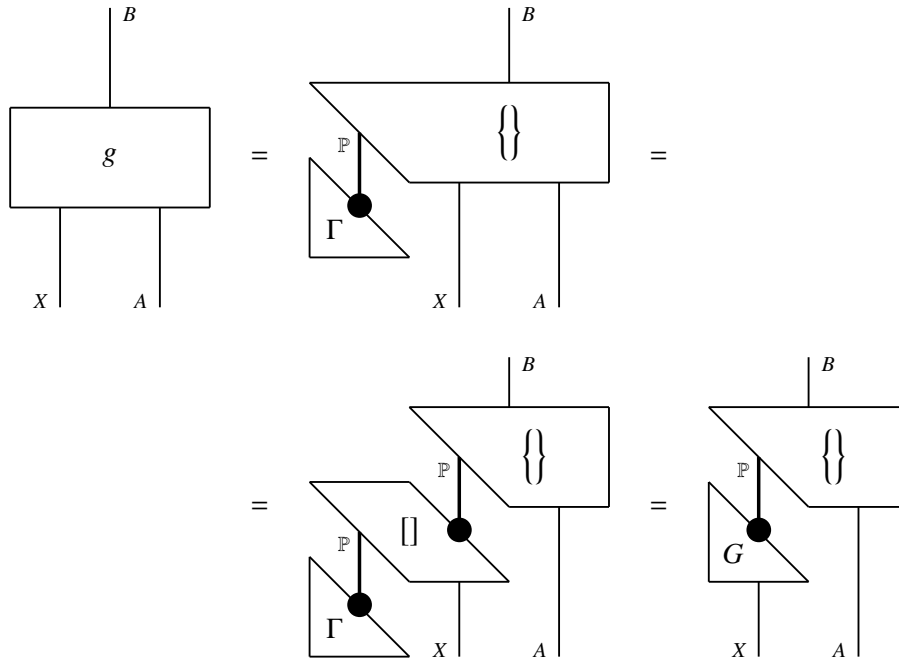


Figure 2.5: An indexed program  $G$  can be constructed by partially evaluating a fixed program  $\Gamma$

summary, (2.3) can be derived from the assumption that all types  $X, A, B$  come with the universal and the partial evaluators generically denoted  $\{\} \in C(\mathbb{P} \times A, B)$ ,  $[] \in C^\bullet(\mathbb{P} \times X, \mathbb{P})$ ,  $\{\} \in C(\mathbb{P} \times X \times A, B)$ , which satisfy

$$\forall f \in C(A, B) \quad \exists F \in C^\bullet(I, \mathbb{P}). \quad \{F\}a = f(a) \quad (2.4)$$

$$\forall F \in C^\bullet(I, \mathbb{P}). \quad \{[F] x\}a = \{F\}(x, a) \quad (2.5)$$

### 2.2.3 Composing programs

We saw in Sec. 1.3.1 how functions are composed sequentially and in parallel:

$$\frac{A \xrightarrow{f} B \quad B \xrightarrow{g} C}{A \xrightarrow{g \circ f} C} \quad \frac{A \xrightarrow{f} B \quad U \xrightarrow{t} V}{A \times U \xrightarrow{f \times t} B \times V}$$

When the functions are computable, we can compute programs for the composite functions from any given programs for their components. More precisely, we can determine  $\mathbb{P} \times \mathbb{P}$ -indexed programs in the form

$$(\cdot)_{ABC} : \mathbb{P} \times \mathbb{P} \longrightarrow \bullet \mathbb{P} \quad (\parallel)_{AUBV} : \mathbb{P} \times \mathbb{P} \longrightarrow \bullet \mathbb{P} \quad (2.6)$$



which for indices  $F, G, T: \mathbb{P}$  with  $\{F\} = f$ ,  $\{G\} = g$ , and  $\{T\} = t$  instantiate to the programs  $(F; G)$  and  $(F \parallel T)$  evaluating to the composite functions

$$\left( A \xrightarrow{\{F; G\}} C \right) = \left( A \xrightarrow{\{F\}} B \xrightarrow{\{G\}} C \right) \quad \left( A \times U \xrightarrow{\{F \parallel H\}} B \times V \right) = \left( A \xrightarrow{\{F\}} B \right) \times \left( U \xrightarrow{\{T\}} V \right)$$

Fig. 2.6 shows how (2.2) induces programs for (2.6).

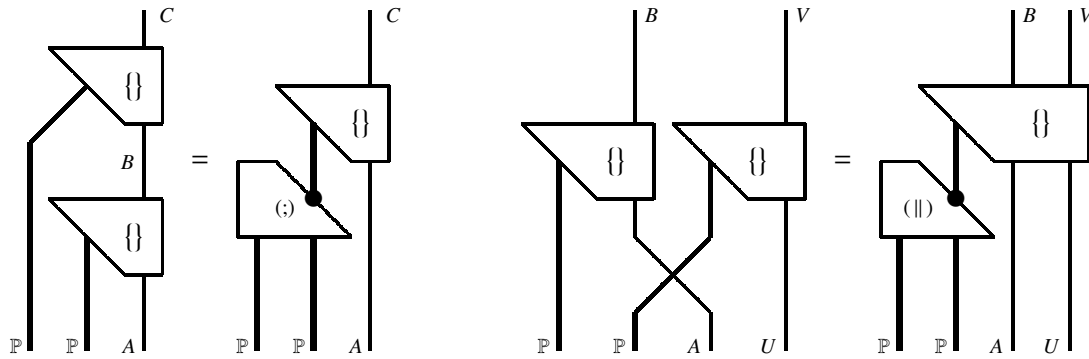


Figure 2.6: Program composition programs (;) and (||)

## 2.3 Programs $\subseteq$ Data $\subseteq$ Programs

A computer as a type universe containing a type  $\mathbb{P}$  of programs together with all datatypes is an embodiment of von Neumann's famous slogan that *programs are data*. The universality of the programming language as a carrier of the evaluators of all types implies the converse inclusion, that *data are programs*. More precisely, data can be encoded as programs, and their types can be checked by program evaluations.

### 2.3.1 Data are programs

Instantiating Fig. 2.3 to the identity function  $\text{id}_A$  for any type  $A$  gives

$$A \xrightarrow{\text{id}_A} A = \text{[Diagram of the identity function as a program]} \quad (2.7)$$

Any data item  $a : A$  is thus encoded as a program  $\ulcorner a \urcorner : \mathbb{P}$  and decoded by  $\{\urcorner a \urcorner\} = a$ . Every type  $A$  is thus a retract of the programming language  $\mathbb{P}$ , embedded into it by  $\ulcorner - \urcorner : A \rightarrow \bullet \mathbb{P}$  and projected back by  $\{\} : \mathbb{P} \rightarrow A$ , which is a surjective partial function. See Appendix 2 for more about retracts. Changing the order of composition yields

$$\rho_A = \left( \mathbb{P} \xrightarrow{\mathbf{q}^A = \{\}} A \xrightarrow{\mathbf{i}_A = \ulcorner - \urcorner} \bullet \mathbb{P} \right) \quad (2.8)$$

also displayed in Fig. 2.7. The function  $\rho_A$  is idempotent, i.e. it satisfies  $\rho_A \circ \rho_A = \rho_A$ .

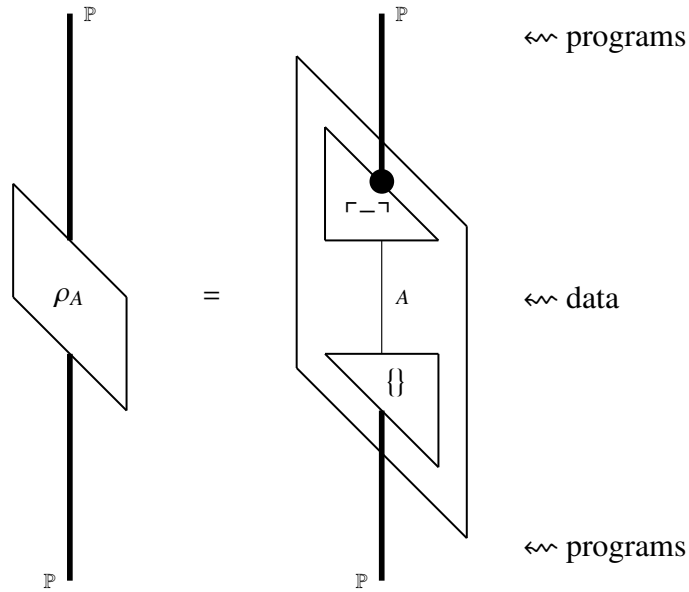


Figure 2.7: Data of type  $A$  correspond to the programs filtered by an idempotent  $\rho_A$ .

### 2.3.2 Types are idempotents

Every type  $A$  corresponds to the idempotent  $\rho_A = \left( \mathbb{P} \xrightarrow{\ulcorner - \urcorner} A \xrightarrow{\{\}} \mathbb{P} \right)$ , where  $x = \rho_A(x)$  holds if and only if the program  $x$  is an encoding of an element of type  $A$ . Since the encoding is injective, the elements of type  $A$  can be identified with their program encodings, and the typing statement  $x : A$  can be viewed as an abbreviation of the equation  $x = \rho_A(x)$ . The idempotent  $\rho_A$  thus fixes the encodings of the elements of  $A$  and filters out all other programs<sup>1</sup>. If we abuse notation and use the label  $A$  as the name of a program for the idempotent  $\rho_A$ , then the type statements can be summarized as idempotent equations

$$x : A \iff x = \rho_A(x) \iff x = \{A\} x \quad (2.9)$$

The type statement for a function  $f : A \rightarrow B$  means that  $x : A$  implies  $f(x) : B$  i.e. that  $x = \{A\} x$  implies  $f(x) = \{B\} (f(x))$ . Using the idempotence, this implication is easily summarized by the equations

$$(f : A \rightarrow B) \iff (f = \rho_B \circ f \circ \rho_A) \iff (f = \{B\} \circ f \circ \{A\}) \quad (2.10)$$

<sup>1</sup>We shall work out in 3.6.1.c that all other programs can be mapped to divergent computations.

which are also equivalent with  $\varrho_B \circ f = f = f \circ \varrho_A$  and  $\{B\} \circ f = f = f \circ \{A\}$ . The formal equivalence justifying the view of types as idempotents is left for workout 2.6.3.2.

### 2.3.3 Pairs of programs are programs

An important special case of encoding all types as programs is the encoding of product types as programs, and in particular encoding of the products of the type of programs itself:

$$(2.11)$$

Similar constructions allow encoding the  $n$ -tuples of programs as single programs, for every natural number  $n$ . Note that the equations

$$\lfloor \lceil x, y \rceil \rfloor_0 = x \qquad \lfloor \lceil x, y \rceil \rfloor_1 = y \qquad (2.12)$$

make the pairing  $\lceil -, - \rceil$  injective, but that there may be programs that do not encode a pair. See Sec. ?? for a construction of surjective pairing.

### 2.3.4 Programming languages encode each other

If a type universe contains two different languages,  $\mathbb{P}_0$  and  $\mathbb{P}_1$ , instantiating (2.7) to each of them displays them as each other's retracts. In other words, each of them can be encoded as a subset of expressions of the other one. In general, when two sets are embedded into each other, the Cantor-Bernstein construction provides a bijection between them [70, 150]. There is thus a one-to-one correspondence between the sets of expressions in any two programming languages. However, this set-theoretic correspondence does not seem useful for programs. One reason is that the encodings do not preserve the meanings of programs. Moreover, the general Cantor-Bernstein construction involves infinitary operations, and the resulting set-theoretic correspondence is not computable. But a computable type-theoretic meaning-preserving bijective correspondence between the programs in any two well-ordered programs will be spelled out in Ch. 7.

### 2.3.5 Idempotents split

**Idempotents on  $\mathbb{P}$  split.** Any computable idempotent  $A : \mathbb{P} \rightarrow \mathbb{P}$  displays the elements  $a : A$  as its fixpoints and filters out the programs  $p$  not in  $A$  either by projecting them to  $A(p)$  which is in  $A$  or by not returning any output on the input  $p$ . Since every idempotent viewed as a type provides its own splitting, the type system of a monoidal computer splits all idempotents, and is said to be *absolutely complete* [121]. Treating the types as idempotents on  $\mathbb{P}$  assures that all idempotents on  $\mathbb{P}$  split. But since all types are retracts of  $\mathbb{P}$ , it follows that all idempotents on all types split.

**All idempotents split.** If  $r : A \rightarrow A$  is an idempotent on  $A$  and

$$\rho_A = \left( \mathbb{P} \xrightarrow{q^A} A \xrightarrow{i_A} \bullet \cdot \mathbb{P} \right)$$

is a split idempotent, then

$$\rho_R = \left( \mathbb{P} \xrightarrow{q^A} A \xrightarrow{r} A \xrightarrow{i_A} \bullet \cdot \mathbb{P} \right)$$

is also an idempotent. If

$$\rho_R = \left( \mathbb{P} \xrightarrow{q^R} R \xrightarrow{i_R} \bullet \cdot \mathbb{P} \right)$$

is an idempotent splitting, then

$$r = \left( A \xrightarrow{i_A} \bullet \cdot \mathbb{P} \xrightarrow{q^R} R \xrightarrow{i_R} \bullet \cdot \mathbb{P} \xrightarrow{q^A} A \right)$$

is also an idempotent splitting, as displayed in Fig. 2.8.

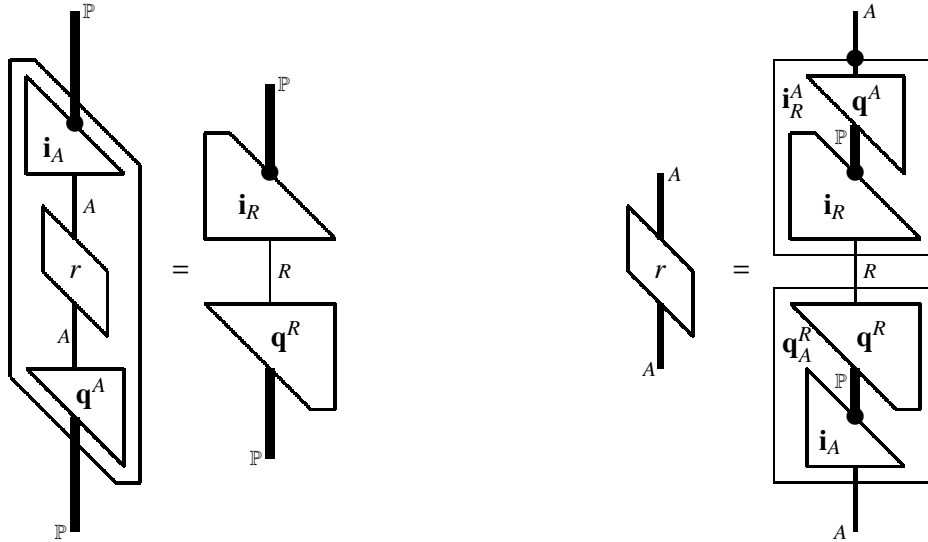


Figure 2.8: Splitting an idempotent in a computer

**Upshot.** Fig. 2.7 shows how the program evaluator of type  $A$  induces an idempotent  $\rho_A$  that filters the elements of type  $A$ . Fig. 2.8 shows how any idempotent  $r : A \rightarrow A$  induces a type  $R$  as its splitting.

Types induce idempotents and the idempotents induce types. This is a foundational view of typing going back to [146]. When confusion is unlikely, viewing type statements as program evaluations (2.9–2.10) provides an intuitive view of type checking. When a computer is given with just a few native types, or even none, then we can use (2.9) to define checkable types. We begin from the simplest one: the booleans.

## 2.4 Logic and equality

### 2.4.1 Truth values and branching

If we take a program  $\top$  for the first projection to play the role of the truth value "true", and a program  $\perp$  for the second projection to play the role of the truth value "false",

$$(2.13)$$

then setting  $ifte = \{\}$  gives the branching operation:

$$ifte(b, x, y) = \{b\}(x, y) = \begin{cases} x & \text{if } b = \top \\ y & \text{if } b = \perp \end{cases} \quad (2.14)$$

The expression  $ifte(b, x, y)$  is meant to be read "if  $b$  then  $x$  else  $y$ ".

### 2.4.2 Program equality

While deciding whether two computations output equal results may need to wait for the computations to produce the outputs, deciding whether two programs are equal or not is in principle easy. To be able to use such decisions in computations, we are given a decidable predicate  $(\stackrel{?}{=}) : \mathbb{P} \times \mathbb{P} \longrightarrow \bullet \mathbb{P}$  which captures the program equality internally:

$$(p \stackrel{?}{=} q) = \begin{cases} \top & \text{if } p = q \\ \perp & \text{otherwise} \end{cases} \quad (2.15)$$

### 2.4.3 The type of booleans

Since the equality  $(\stackrel{?}{=})$  is total, the test  $(x \stackrel{?}{=} \top)$  returns  $\top$  if  $x = \top$  and  $\perp$  otherwise. The cartesian function

$$\begin{aligned} \rho : \mathbb{P} &\longrightarrow \bullet \mathbb{P} \\ x &\mapsto ifte(x \stackrel{?}{=} \top, \top, \perp) \end{aligned} \quad (2.16)$$

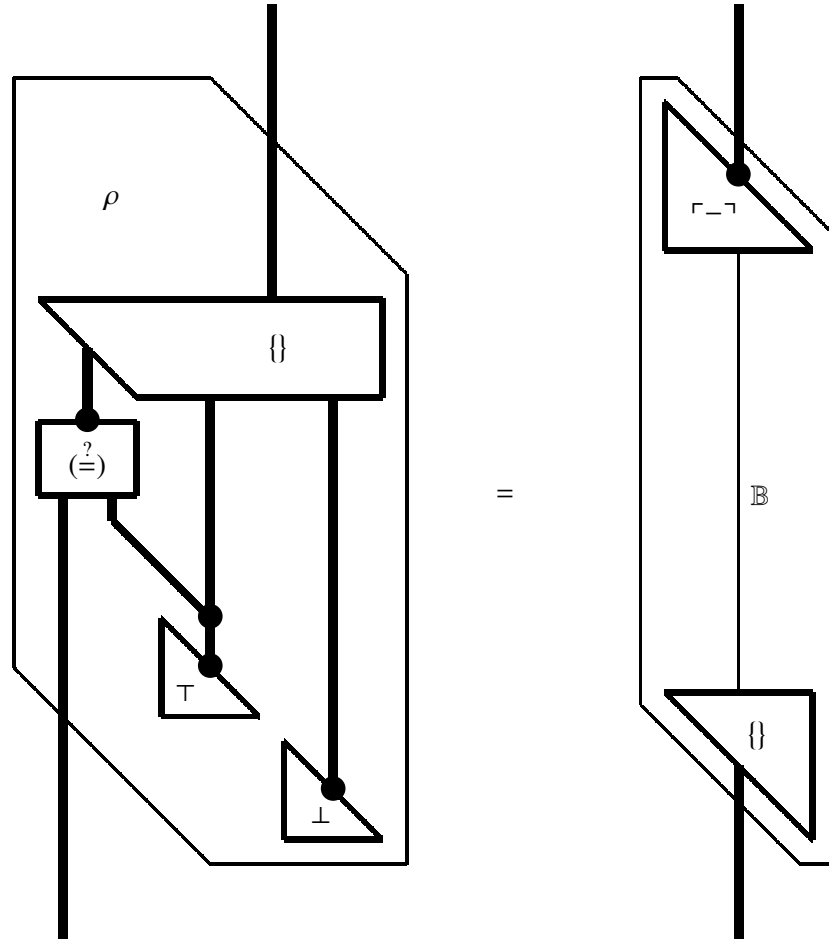


Figure 2.9: The type  $\mathbb{B} = \{\perp, \top\}$  splits the idempotent  $\rho$

displayed in Fig. 2.9, is clearly idempotent, as it maps  $\top$  to  $\top$  and all other programs to  $\perp$ . The type  $\mathbb{B}$  arises by splitting this idempotent. It filters the programs  $\top$  and  $\perp$  and fixes the type  $\mathbb{B} = \{\top, \perp\}$ . More precisely, the elements of this type are  $C^\bullet(I, \mathbb{B}) = \{\top, \perp\}$ .

#### 2.4.4 Propositional connectives

In any computer, the native truth values  $\top, \perp$  and the *ifte*-branching yield native propositional connectives. One way to define them is displayed in Fig. 2.10. The negation  $\neg q$  can be reduced to the implication  $q \Rightarrow \perp$ .

#### 2.4.5 Predicates and decidability

**Predicates are single-valued.** A predicate is a *single-valued* function of in the form  $q: X \rightarrow \mathbb{B}$ . It thus assigns a unique truth value  $q(x): \mathbb{B}$  to each  $x: X$ .

The single-valuedness requirement prevents that  $q(x)$  may be true today and false tomorrow. Formally,

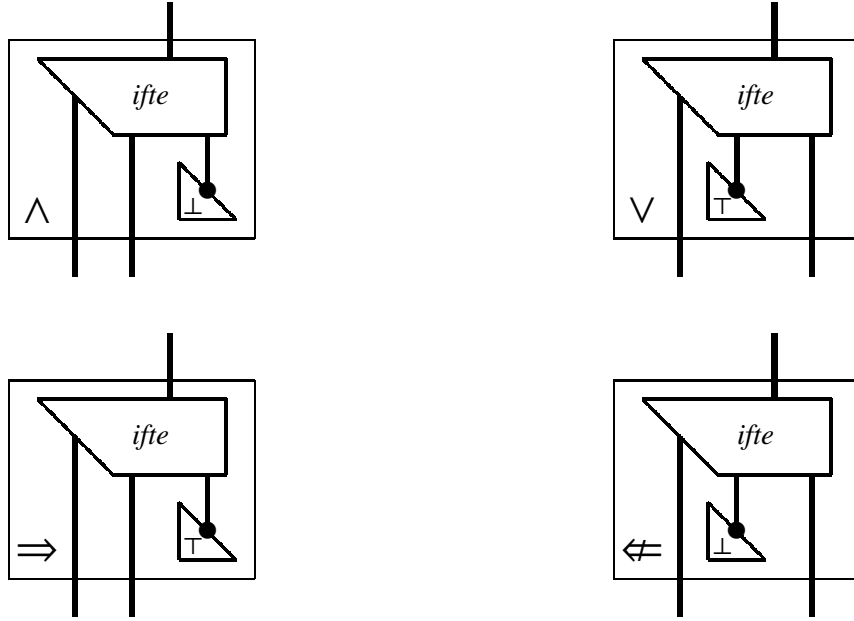


Figure 2.10: The  $ifte = \{\}$  induces the propositional connectives

this requirement is expressed by the equation  $(q \times q) \circ \Delta_X = \Delta_{\mathbb{B}} \circ q$  discussed in Sec. 1.4. For a computation  $q$ , the equation says that running  $q$  twice on the same input gives two copies of the same output. This justifies thinking of a predicate as asserting a property.

**Assertion notation.** The equation  $q(x) = \top$  is often abbreviated to the *assertion*  $q(x)$  that  $x$  has a property  $q$ . The equation  $q(x) = \perp$  is abbreviated to  $\neg q(x)$ .

**Predicates may not be total.** Computations are branched using computable predicates  $q$ , e.g. in  $ifte(q(x), f(x), g(x))$ . If  $q(x)$  diverges,  $ifte(q(x), f(x), g(x))$  will also diverge and the branching will remain undecided.

**Decidable predicates.** A predicate  $q$  is *decidable* if it is total, i.e. if it converges and outputs  $q(x) = \top$  or  $q(x) = \perp$  for all  $x : X$ . Formally, this is assured by the equation  $\uparrow_{\mathbb{B}} \circ q = \uparrow_X$ , discussed in Sec. 1.4. In the sense of that section, the decidable predicates over  $X$  are precisely the cartesian function  $X \longrightarrow \bullet \mathbb{B}$ .

**Halting notation**  $q(x) \downarrow$  abbreviates the equation  $\uparrow_{\mathbb{B}} \circ q \circ x = \uparrow_I$  for any cartesian  $x : I \longrightarrow \bullet X$ .

## 2.5 Monoidal computer

### 2.5.1 Definition

A *monoidal computer* is a (symmetric, strict) monoidal category  $\mathcal{C}$  with

- a) a data service  $A \times A \xrightarrow{\Delta} A \xrightarrow{\uparrow} I$  on every type  $A$ , as described in Sec. 1.2,

- b) a distinguished type  $\mathbb{P}$  of *programs* with a decidable equality predicate  $(\stackrel{?}{=}) : \mathbb{P} \times \mathbb{P} \longrightarrow \mathbb{B}$ , and
- c) a *program evaluator*  $\{\} : \mathbb{P} \times A \longrightarrow B$  for all types  $A, B$ , as in (2.2).

## 2.5.2 Examples

**Programming languages.** As mentioned in Sec. ??, the interpreters of programming languages provide examples of program evaluators. In most cases, the type universe of the programming language is an example of a monoidal computer. The objects of a monoidal computer  $C_{\mathbb{P}}$  are thus the types are checkable as the idempotent functions that are  $\mathbb{P}$ -programmable. The morphisms of  $C_{\mathbb{P}}$  are all  $\mathbb{P}$ -programmable functions, partitioned into hom-sets. The details are worked out in Sec. 2.6.3.2. Any Turing-complete programming language  $\mathbb{P}$  thus induces a monoidal computer. See Sec. 4.6 for more about Turing completeness. While  $C_{\mathbb{P}}$ 's program evaluators correspond to  $\mathbb{P}$ 's interpreters,  $C_{\mathbb{P}}$ 's partial evaluators correspond to its specializers [81].  $\mathbb{P}$ 's Turing-completeness implies that its interpreters and specializers can be programmed in  $\mathbb{P}$  itself. Sec. 4.6 and [126, 129] contain more details and references. The other way around, the morphisms of any monoidal computer  $C_{\mathbb{P}}$  are  $\mathbb{P}$ -programmable, in the sense of Sec. 2.2.1, which makes  $\mathbb{P}$  into an abstract programming language.

**Models of computation.** Note that each of the standard models of computation, the Turing machines, partial recursive functions, etc., can be viewed as programming languages, and subsumed under the examples above. E.g., any Turing machine can be viewed as a program, with a universal Turing machine as a program evaluator. With suitably formalized execution schemas and a concept of types, any of the models can be used as a programming language [146] and expanded into a monoidal computer.

## 2.5.3 Program evaluation as natural surjection

In the presence of the structures from (a–b), the program evaluator condition (c) can be equivalently stated as the following natural form of the "running" surjection from Fig. 3.

- c') an  $X$ -natural family of surjections  $C(X \times A, B) \xleftarrow{\text{run}_X^{AB}} C^{\bullet}(X, \mathbb{P})$  for each pair of types  $A, B$ .

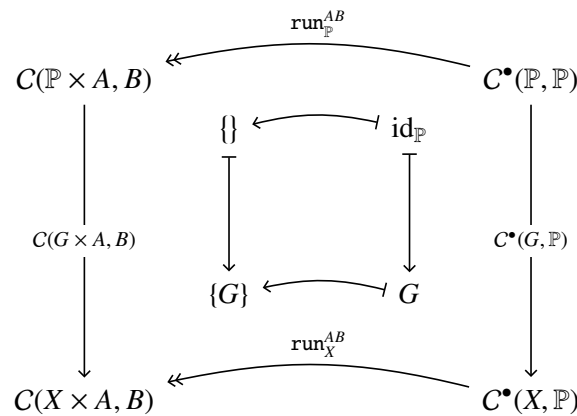


Figure 2.11: Any program evaluator  $\{\}$  induces the  $X$ -indexed family  $\text{run}_X^{AB}$ .



Towards a proof of  $(\mathbf{c} \Rightarrow \mathbf{c}')$ , note that any program evaluator  $\{\} : \mathbb{P} \times A \rightarrow B$  induces an  $X$ -natural family  $\text{run}^{AB}$  making the diagram in Fig. 2.11 commute, whose components are

$$\text{run}_X^{AB}(G) = \{G\} \quad (2.17)$$

The naturality requirement in Fig. 2.12 means that every  $s \in C^\bullet(Y, X)$  and every  $G \in C^\bullet(X, \mathbb{P})$  should

$$\begin{array}{ccc} C(X \times A, B) & \xleftarrow{\text{run}_X^{AB}} & C^\bullet(X, \mathbb{P}) \\ \downarrow C(s \times A, B) & & \downarrow C^\bullet(s, \mathbb{P}) \\ C(Y \times A, B) & \xleftarrow{\text{run}_Y^{AB}} & C^\bullet(Y, \mathbb{P}) \end{array}$$

Figure 2.12: The naturality requirement for  $\text{run}^{AB}$ .

satisfy  $\text{run}_X^{AB}(G) \circ s = \text{run}_Y^{AB}(G \circ s)$ . Definition (2.17) reduces this requirement to  $\{G\}_X \circ s = \{G \circ s\}_Y$ , which is tacitly imposed by the string diagrammatic formulation of (2.2), and validated by precomposing both of its sides with  $s : Y \rightarrow X$ . Condition (2.3) says that all  $\text{run}_X^{AB}$  are surjective, as required. The other way around, to prove  $(\mathbf{c}' \Rightarrow \mathbf{c})$ , define

$$\{\} = \text{run}_{\mathbb{P}}^{AB}(\text{id}) \quad (2.18)$$

and note that now the surjectiveness of  $\text{run}_{\mathbb{P}}^{AB}$  gives (2.3).

## 2.6 Workout

### 2.6.1 How many programs?

Is there a computer with 3 programs? No. There are either infinitely many, or there is just one and  $\top = \perp$ . We first discharge this trivial case, and then as soon as  $\top \neq \perp$ , there must be infinitely many different programs for each computation.

a. Consider a monoidal computer  $C$  where  $\top = \perp$ . In other words, suppose that the programs for  $\top$  and  $\perp$  chosen in (2.13) turn out to be equal. Prove the following claims.

- i) Any pair of functions  $\varphi, \gamma : \mathbb{P} \rightarrow \mathbb{P}$  in  $C$  satisfy  $\varphi = \gamma$ .
- ii) Any pair of functions  $f, g : A \rightarrow B$  in  $C$  satisfy  $f = g$ .
- iii) For any pair of types  $C, D$  in  $C$  there is an isomorphism  $C \cong D$  (i.e. a pair of functions  $C \xrightleftharpoons[d]{c} D$  such that  $c \circ d = \text{id}_C$  and  $d \circ c = \text{id}_D$ ).

- iv) Conclude that any monoidal computer  $C$  with  $\top = \perp$  is equivalent with the trivial category consisting of a single object and a single morphism.
- b. Consider a monoidal computer  $C$  where  $\top \neq \perp$ . Prove that the following claims are valid for any number  $n$ .
- The programming language  $\mathbb{P}$  in  $C$  contains at least  $2^n$  different programs.
  - For any function  $f : A \rightarrow B$  there are at least  $2^n$  different programs  $F : \mathbb{P}$  such that  $f = \{F\}$ .
- Conclude that any monoidal computer  $C$  where  $\top \neq \perp$  contains infinitely many programs, and that any computable function  $f : A \rightarrow B$  can be encoded by infinitely many different programs  $F : \mathbb{P}$  that evaluate to  $f = \{F\}$ .
- c. Explain that in a monoidal computer  $C$ ,
- either every indexed function  $g \in C(X \times A, B)$  has infinitely many programs  $G \in C^\bullet(X, \mathbb{P})$  with  $g = \{G\}$ ,
  - or  $C \simeq 1$ .

### 2.6.2 Counting by evaluating: Church numerals

The simplest form of computation is counting. The simplest programs that can be derived from program evaluators are the numbers, counting how many times a program is evaluated over its inputs. The idea is to redefine the numbers internally in a computer as the programs  $\overline{0}, \overline{1}, \overline{2} \dots$  such that

$$\begin{aligned}
 \{\overline{0}\}(F, a) &= \{F\}^0 a = a \\
 \{\overline{1}\}(F, a) &= \{F\}^1 a = \{F\} a \\
 \{\overline{2}\}(F, a) &= \{F\}^2 a = \{F\} \circ \{F\} a \\
 \{\overline{3}\}(F, a) &= \{F\}^3 a = \{F\} \circ \{F\} \circ \{F\} a \\
 &\dots \\
 \{\overline{n}\}(F, a) &= \{F\}^n a = \underbrace{\{F\} \circ \dots \circ \{F\} \circ \{F\}}_{n \text{ times}} a
 \end{aligned} \tag{2.19}$$

The overlined  $\overline{n}$  is thus a number-as-program, internal in a computer, whereas  $n$  is the usual, "external" number. When the distinction is irrelevant, we identify them. The string-diagrammatic view of the internal numbers is in Fig. 2.13.

**Remark.** The task of redefining numbers as programs is a first step into *reverse programming*, explained in Sec. 4.6. The programs in Fig. 2.13 follow Church's idea from [30].

**Exercises.** Specify programs to implement the following arithmetic operations on Church's numerals:

- $s(\overline{n}) = \overline{n + 1}$
- $a(\overline{m}, \overline{n}) = \overline{m + n}$
- $m(\overline{m}, \overline{n}) = \overline{m \times n}$

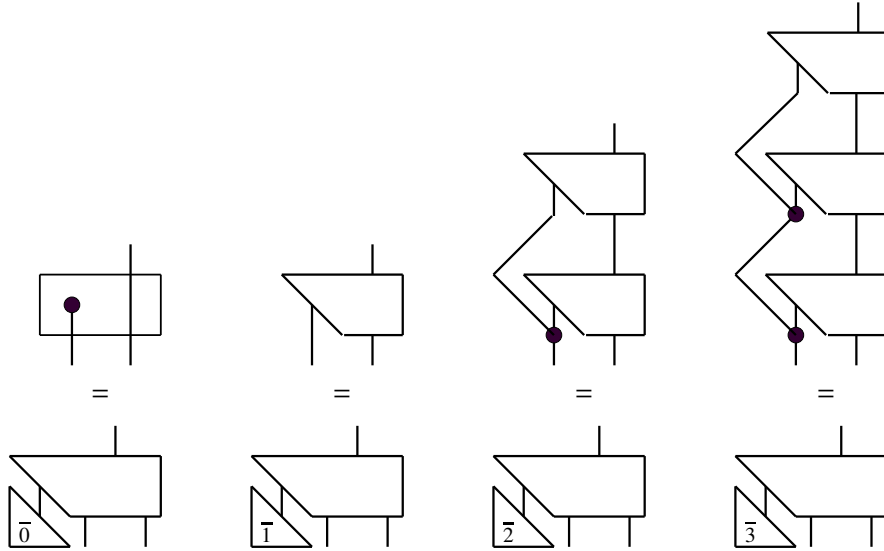


Figure 2.13: Church's numerals (without the  $\lambda$ -abstractions)

### 2.6.3 Monoid computer?

In Sec. 2.3.1, we saw that any type  $A$  of a monoidal computer comes with a retraction  $A \xrightleftharpoons[i]{q} \mathbb{P}$  and can be identified with the induced idempotent  $\rho = \mathbf{i} \circ \mathbf{q}$  on  $\mathbb{P}$ . The question arises:

*Can all structure of a monoidal computer  $C$  be reduced to its programming language  $\mathbb{P}$ ?*

In this workout, we pursue this question in parts, asking how to reduce the underlying category, its monoidal structure, and the program evaluators of a monoidal computer to the monoid of computable functions on its programming language. Some of the answers require categorical background (see Appendix 1) and probably too much work for the first round but they have interesting repercussions, so please come back. The first ones are easy.

#### 2.6.3.1 Universal program evaluator $\{\}: \mathbb{P} \times \mathbb{P} \longrightarrow \mathbb{P}$

Prove the following claims:

- a. Any program evaluator  $\{\}_{\mathbb{P}}^{\mathbb{P}}$  induces the program evaluators  $\{\}_A^B$  as the following composites

$$(2.20)$$

for all types  $A, B$ . Prove that the composite (2.20) satisfies condition (2.2–2.3).

- b. Any program evaluator  $\{\}_A^B$  can be reduced to  $\{\}_{\mathbb{P}}^{\mathbb{P}}$  using a program transformer  $\nu_A^B$  and the type projections and embeddings, in the following form

$$(2.21)$$

### 2.6.3.2 $C$ is the idempotent completion of its submonoid on $\mathbb{P}$

Let  $\mathcal{P} \times \mathcal{P} \xrightarrow{\circ} \mathcal{P} \xleftarrow{\text{id}_{\mathbb{P}}} 1$  be a monoid, viewed as a category with a single object, call it  $\mathbb{P}$ . Its *idempotent completion* is defined to be the following category

$$\begin{aligned} |\mathcal{P}^{\cup}| &= \{\varrho \in \mathcal{P} \mid \varrho \circ \varrho = \varrho\} \\ \mathcal{P}^{\cup}(\varrho, \varsigma) &= \{\varphi \in \mathcal{P} \mid \varsigma \circ \varphi = \varphi = \varphi \circ \varrho\} \end{aligned} \quad (2.22)$$

See Appendix 2 for more about this construction. It comes with the embedding

$$\flat: \mathcal{P} \rightarrow \mathcal{P}^{\cup} \quad (2.23)$$

which maps  $\mathbb{P}$ , the sole object of  $\mathcal{P}$ , to the unit  $\text{id}$  as an idempotent, and any element  $\varphi$  of  $\mathcal{P}$  to itself, as an endomorphism on the idempotent  $\text{id}$ .

Prove the following claims.

- a. Let  $C$  be a category where all idempotents split (as in Appendix 2). Then any functor  $F : \mathcal{P} \rightarrow C$  has a unique extension  $F^\# : \mathcal{P}^\cup \rightarrow C$ , making the following diagram commute

$$\begin{array}{ccc} \mathcal{P} & \xrightarrow{\quad b \quad} & \mathcal{P}^\cup \\ & \searrow F \quad \swarrow F^\# & \\ & C & \end{array} \quad (2.24)$$

- b. Given an object  $\mathbb{P}$  of  $C$ , consider the monoid  $\mathcal{P} = C(\mathbb{P}, \mathbb{P})$  as a full subcategory of  $C$ . Show that the embedding  $b : \mathcal{P} \hookrightarrow \mathcal{P}^\cup$  from (2.23) extends along the subcategory inclusion  $\mathcal{P} \hookrightarrow C$  to a functor  $\rho : C \rightarrow \mathcal{P}^\cup$ , making the following diagram commute

$$\begin{array}{ccc} \mathcal{P} & \xrightarrow{\quad b \quad} & C \\ & \searrow b \quad \swarrow \rho & \\ & \mathcal{P}^\cup & \end{array} \quad (2.25)$$

if and only if every object  $A \in |C|$  is a retract of  $\mathbb{P}$ . Verify moreover that  $\rho$  is full and faithful (i.e., that it establishes a bijection  $C(A, B) \cong {}^\cup \rho_A, \rho_B$ ) for all pairs  $A, B$ ) and that it is determined by  $\mathcal{P} \hookrightarrow C$  up to isomorphism.

- c. Let  $C$  be a category where every object  $A$  is a retract of  $\mathbb{P}$  and all idempotents split. Then the functor  $\rho$  is an equivalence of categories with a mate  $\widehat{(-)}$

$$\begin{array}{ccc} C & \xrightleftharpoons[\rho]{\widehat{(-)}} & \mathcal{P}^\cup \end{array} \quad (2.26)$$

and there are isomorphisms  $\widehat{\rho_A} \cong A$  in  $C$  and  $\rho_{\widehat{\varrho}} \cong \varrho$  in  $\mathcal{P}^\cup$ , natural in  $A$  and  $\varrho$ .

- d. The equivalence  $\rho : C \rightarrow \mathcal{P}^\cup$  induces a bijection  $C(\mathbb{P}, \mathbb{P}) \cong \mathcal{P}^\cup(\rho_{\mathbb{P}}, \rho_{\mathbb{P}})$ . Show that this implies that  $\rho_{\mathbb{P}}$  must be invertible, and furthermore that the components of the splitting  $\rho_{\mathbb{P}} = (\mathbb{P} \xrightarrow{\mathbf{q}} \widehat{\rho_{\mathbb{P}}} \xrightarrow{\mathbf{i}} \mathbb{P})$  must also be invertible.

**Summary.** The functor  $\rho$  selects for every type  $A \in |C|$  an idempotent

$$\rho_A = \left( \mathbb{P} \xrightarrow{\mathbf{q}} A \xrightarrow{\mathbf{i}} \mathbb{P} \right) \quad (2.27)$$

The mate  $\widehat{(-)}$  selects for every idempotent  $\varrho \in C(\mathbb{P}, \mathbb{P})$  a splitting

$$\left( \mathbb{P} \xrightarrow{\mathbf{q}} \widehat{\varrho} \xrightarrow{\mathbf{i}} \mathbb{P} \right) = \varrho \quad (2.28)$$

The equivalence functors thus assign to every  $A \in C$  a particular program encoding (2.27), and to every idempotent  $\varrho \in \mathcal{P}^\cup$  a particular splitting (2.28).

### 2.6.3.3 Relating the products in $\mathcal{C}$ and the pairing on $\mathbb{P}$

Any equivalence of categories  $\mathcal{X} \simeq \mathcal{Y}$  allows transferring a given monoidal structure from  $\mathcal{X}$  to  $\mathcal{Y}$ . Continuing with the category  $\mathcal{C}$  and its submonoid  $\mathcal{P}$  from 2.6.3.2, it may be instructive to first work out this general structure transfer for the special case of the monoidal structure  $(\mathcal{C}, \times, I)$  and transfer it along the equivalence  $\mathcal{C} \simeq \mathcal{P}^\cup$ . But going further, in this special case at hand, the monoidal structure of  $\mathcal{C}$  is also reflected on its submonoid  $\mathcal{P}$  in terms of the retractions

$$\rho_{\mathbb{P} \times \mathbb{P}} = \left( \mathbb{P} \xrightarrow{\langle \lfloor - \rfloor_0, \lfloor - \rfloor_1 \rangle} \mathbb{P} \times \mathbb{P} \xrightarrow{[-, -]} \mathbb{P} \right) \quad \rho_I = \left( \mathbb{P} \xrightarrow{\mathbf{i}} I \xrightarrow{\iota} \mathbb{P} \right) \quad (2.29)$$

It is interesting, and can be subtle, to express the monoidal structure of  $\mathcal{P}^\cup$  in terms of these retractions.

- a. Mapping any pair  $\varrho, \varsigma \in |\mathcal{P}^\cup|$  along the equivalence in to  $\widehat{\varrho}, \widehat{\varsigma} \in |\mathcal{C}|$ , and then their product  $\widehat{\varrho} \times \widehat{\varsigma}$  back to  $\mathcal{P}^\cup$  gives

$$\varrho \otimes \varsigma = \rho_{\widehat{\varrho} \times \widehat{\varsigma}} \quad (2.30)$$

Verify that this defines a functor  $\otimes: \mathcal{P}^\cup \times \mathcal{P}^\cup \rightarrow \mathcal{P}^\cup$  and that  $(\mathcal{P}^\cup, \otimes, \rho_I)$  is a monoidal category.

- b. Work out the data services in  $\mathcal{P}^\cup$  induced by the data services in  $\mathcal{C}$ .

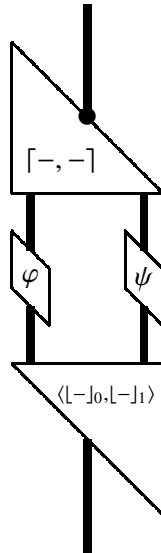


Figure 2.14: The convolution  $(\varphi \star \psi) x = [\varphi \lfloor x \rfloor_0, \psi \lfloor x \rfloor_1]$

- c. The pairing in (2.29) determines a *convolution* operation  $(\star): \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$  displayed in Fig. 2.14. Prove the following claims.

- i)  $\varphi \star \psi$  is idempotent whenever  $\varphi$  and  $\psi$  are idempotent.
  - ii)  $\varrho \star \varsigma \cong \varrho \otimes \varsigma$  holds for all idempotents  $\varrho, \varsigma$  on  $\mathbb{P}$ .
  - iii) There are monoidal equivalences  $(\mathcal{P}^\cup, (\star), \rho_I) \simeq (\mathcal{P}^\cup, \otimes, \rho_I) \simeq (\mathcal{C}, \times, I)$
- d. Suppose that the operation  $(\star): \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$  is associative and has a unit  $\iota$ , which yields a monoidal

structure  $(\mathcal{P}, \star, \rho_I)$  making the monoid  $(\mathcal{P}, \circ, \text{id}_{\mathbb{P}})$  into a monoidal category. Note that this means that  $\star$  and  $\circ$  satisfy the middle-two-interchange law from Sec. 1.3.2. Show that the two monoidal structures must coincide, with  $\varphi \star \psi = \varphi \circ \psi$  and  $\rho_I = \text{id}_{\mathbb{P}}$  and leading to  $I \cong \mathbb{P}$  and reducing the category  $\mathcal{C}$  to a commutative monoid.

- e. In summary, for any monoidal category  $\mathcal{C}$  where all objects are retracts of a single object  $\mathbb{P}$  and for the submonoid  $\mathcal{P} = C(\mathbb{P}, \mathbb{P})$ , we so far proved that

- (c.) the operation  $(\star)$  is associative and unitary on  $\mathcal{P}^\cup$ , but
- (d.) if the operation  $(\star)$  is associative and unitary on  $\mathcal{P}$ , then  $\mathbb{P} \cong I$ .

Does it follow that every monoidal category where all objects are retracts of a single object must collapse to a commutative monoid? Explain!

**Lookahead.** In Sec. ??, we will work out a pairing operation  $[-, -]: \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$  (with projections) which is associative up to isomorphism in  $\mathcal{P}$ , and has a one-sided unit. It can be constructed in any monoidal computer with well-ordered programs.

#### 2.6.3.4 Reducing program evaluators $\{\}_A^B$ to $\{\}_{\mathbb{P}}^{\mathbb{P}}$

Sec. 2.6.3.2 reduced the underlying category of a monoidal computer  $\mathcal{C}$  to its submonoid  $\mathcal{P}$  on  $\mathbb{P}$ . Sec. 2.6.3.3 reduced  $\mathcal{C}$ 's monoidal structure and data services to the pairing operation on  $\mathbb{P}$ . Up to equivalence, the monoidal category  $\mathcal{C}$  underlying a monoidal computer can be recovered from its submonoid  $\mathcal{P}$  on  $\mathbb{P}$ . Can the program evaluators be reduced in a similar way?

Sec. 2.6.3.1 made some first steps towards an answer. Claim b was that general program evaluators  $\{\}_A^B$  in  $\mathcal{P}^\cup$  can be derived from  $\{\}_{\mathbb{P}}^{\mathbb{P}}$  using the type retractions and program transformers  $\nu_A^B$ . Claim a was that a special family of program evaluators  $\{\}_A^B$  in  $\mathcal{P}^\cup$  can be derived from  $\{\}_{\mathbb{P}}^{\mathbb{P}}$  even without the program transformers, i.e. assuming that  $\nu_A^B = \text{id}$ . We work out the special properties of this special family of program evaluators.

- a. Let  $(\mathcal{P}, \circ, \mathbb{P})$  be a monoid and its idempotent completion a monoidal category  $(\mathcal{P}^\cup, (\star), I)$  with data services. For

- i) a morphism  $\{\} \in \mathcal{P}^\cup(\mathbb{P} \star \mathbb{P}, \mathbb{P})$  and
- ii) a family  $\{\nu_\varrho^S\}_{\varrho, S \in \mathcal{P}^\cup}$ ,

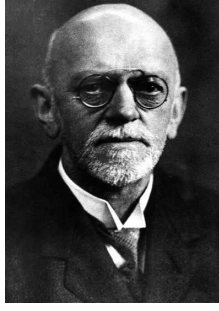
spell out the suitable versions of condition (2.3) which makes

$$\{-\}_\varrho^S = S \circ \{\nu_\varrho^S(-)\} \circ \varrho$$

displayed in (2.21), into a  $\varrho S$ -indexed family of program evaluators, and  $\mathcal{P}^\cup$  into a monoidal computer.

- b. We call a monoidal computer  $\mathcal{C} \simeq \mathcal{P}^\cup$  *uniform* if in (ii) above, all  $\nu_\varrho^S$  are identities and the  $\varrho S$ -program evaluators are in the form

$$\{\}_\varrho^S = S \circ \{\} \circ \varrho$$



David Hilbert



Kurt Gödel



Alan Turing



John von Neumann

Figure 2.15: The relays from Entscheidungsproblem to computer

as displayed in (2.20). In this part we work out the special properties of uniform monoidal computers.

- i) If  $C$  a uniform monoidal computer, then the idempotents  $\rho_A$  and their splittings  $(\mathbb{P} \xrightarrow{q} A \xrightarrow{i} \mathbb{P})$ , both selected by the equivalence  $C \simeq \mathcal{P}^\cup$ , interact with the partial evaluations. Prove that the encodings satisfy the following equations

$$\begin{aligned} \mathbf{i}_{A \times B} &= \mathbf{i}_{\mathbb{P} \times \mathbb{P}} \circ (\mathbf{i}_A \times \mathbf{i}_B) & \mathbf{i}_{\mathbb{P}} &= \text{id}_{\mathbb{P}} \\ \mathbf{i}_{\mathbb{P} \times \mathbb{P} \times \mathbb{P}} &= \mathbf{i}_{\mathbb{P} \times \mathbb{P}} \circ (\mathbf{i}_{\mathbb{P} \times \mathbb{P}} \times \text{id}_{\mathbb{P}}) & \mathbf{i}_{\mathbb{P} \times \mathbb{P}} \circ (\mathbf{i}_I \times \text{id}_{\mathbb{P}}) &= \text{id}_{\mathbb{P}} \end{aligned}$$

- ii) Conclude that  $\mathbf{q}^{\mathbb{P}} = \text{id}_{\mathbb{P}}$  and that defining  $\mathbf{q}^A = \{\}_I^A$ , like we did in Sec. 2.3.1, implies  $\{\}_A^{\mathbb{P}} = \square_A$ . We will see in Sec. 3.2 that this is impossible, unless the computer is degenerate, because there are  $F, a$  for which  $\{F\}_A^{\mathbb{P}} a$  cannot be assigned a value whereas  $[F]_A$  is required to be total.

## 2.7 Stories

### 2.7.1 Who invented the computer?

The *invention* of the computer as a universal state machine led to the *discovery* of computation as a universal process in nature. The advent of computer engineering had a tremendous impact on the economic expansion in the second half of the XX century. But the advent of computer science opened a new era of science, of our interaction with nature and with ourselves. Nothing will ever be the same.

The successes of the XIX century science gave rise to the optimism expressed by David Hilbert in his famous slogan:

There is no ignorabimus in science. We must know, and we shall know!

He closed with this exclamation his Königsberg address in September 1930, at the conference held in his honor, on the occasion of his retirement. For mathematicians, Hilbert had previously distilled this quest for an omnipotent science in the *Decision Problem* (*Entscheidungsproblem*), which can be paraphrased as the challenge to



Find a method to decide by finitary means the truth value of any formal statement.

This was the centerpiece of the famous Hilbert's Program. A positive solution was hoped to provide a logical foundation for science. However, at the same Königsberg conference, in one of the late sessions, the 24-year-old Kurt Gödel shyly announced that he had constructed a provably undecidable statement in the elementary theory of arithmetic. The construction was based on a completely new method, invented by Gödel for this purpose. At the Königsberg conference, Gödel's announcement went largely unnoticed, certainly by Hilbert<sup>2</sup>. A year later, when Gödel published the details of his result [58], it shattered the foundational movement straddling the communities of mathematicians and logicians. It left Hilbert confused, and allegedly in denial. Gödel's method provided the first stepping stone into the science of computation. Using the recursion schema (discussed in Sec. 4.3), Gödel encoded the formulas of arithmetic as numbers and the proofs of theorems as special arithmetic operations. The insight of Gödel's proof was that the logical theories, and in particular the theory of arithmetic, can be encoded as word processing, and that word processing can be encoded in arithmetic. In this way, the elementary arithmetic can encode its own proofs *and* check their derivability from axioms. Gödel's basically invented *programming*, albeit without a computer. The computer had to wait for Turing. Gödel used arithmetic as a universal computer, or more precisely as a universal logical theory, expressive enough to encode logical theories in general, and to derive their proofs. He constructed his undecidable statement by encoding the elementary arithmetic in itself and instantiating the code of the statement "This number is the code of a false statement" to the number that encodes that same statement. The resulting statement is undecidable because its truth would imply its falsity, and vice versa. This is a *diagonal* argument, going back to Cantor, and previously used by Russell to construct his paradox which shook up Frege's hopes like Gödel shook up Hilbert's. Yet another avatar of the diagonal argument is the Fundamental Theorem of Computation, which we will encounter in Sec. 3.3.

The computer as a machine was first described in the 1936 paper by the 24-year-old Alan Turing. Turing propagated Gödel's idea of programming from logical formulas to state machines. Just like the arithmetic formulas can be encoded as numbers and processed in arithmetic, Turing machines can be encoded and processed by Turing machines. While Gödel noticed that the logical theory of arithmetic was capable of encoding and processing itself, Turing performed the remarkable feat of specifying from scratch a family of machines where the same was possible. In the very same paper where he introduced his machines, Turing constructed a Universal Machine, that inputs a description of an arbitrary Turing machine, and then acts as that machine, i.e. performs its computations on its inputs, and produces its outputs. Turing's Universal Machine was the first computer. While Gödel's constructions were purely logical, Turing's machines were quickly picked up by engineers, and went into production a couple of years later, during World War II. John von Neumann played an important role in transferring the concept of a Universal Machine into a computer architecture that came to carry his name. Some of the first von Neumann computers were used for work on the first nuclear bombs. Alan Turing worked during the war on a different kind of bomb. He developed up the "cryptological bomb", conceived by Polish cryptographers, and broke the German cryptosystem Enigma. The achievement is said to have saved many lives and played a critical role in the allied victory. After the war, Turing returned to his work on computation and published his seminal ideas relating computation, life, and what later came to be called artificial intelligence. His ideas about morphogenesis are sometimes said to have been as much of a paradigm shift in biology as his ideas about computers were in logic. His concept of testing and indistinguishability

<sup>2</sup>Von Neumann allegedly whispered to Bernays as soon as Gödel finished his talk: "[Hilbert's Program] is finished". In the coming weeks, he derived an important consequence of Gödel's result, and briefly corresponded with Gödel about it. He never announced it, because Gödel had already proved the result himself, and included it the paper that he was writing. The result came to be known as the Second Incompleteness Theorem. But John von Neumann seemed to be the only person who immediately understood Gödel's result.

remains in the foundations of the areas of computer science as different as semantics and cryptography. In 1952, Alan Turing was prosecuted for homosexuality, which was at the time a criminal offense in the UK, and sentenced to undergo hormonal treatment to suppress his libido. He died two years later, at the age of 41. The cause of death was poisoning with cyanide, that was found in the apple from which he had taken a bite. Although his favorite tale was known to be "*The Snow White*", it remains uncertain whether his death was intentional.

### 2.7.2 How many truth values?

The XX century logic expanded the horizons of truth far beyond  $\mathbb{B} = \{\top, \perp\}$ . The theory of computation allowed refining the Brouwer-Heyting-Kolmogorov view of mathematical constructions, mentioned in Sec. 1.7.3.1, into a theory of *realizability* of logical statements [78, 90, 165]. The practice of computation allowed realizing the semantics of *propositions-as-types* [76, 98], also discussed in Sec. 1.7.3.1, as a programming tool [175]. We shall see in Ch. 4 that the truth values  $\top$  and  $\perp$  suffice for drawing lots of programs. On the other hand, already in Ch. 3, we shall see that even the simplest computations may be unable to decide whether to return  $\top$  or  $\perp$ . This undecidability, spelled out in Ch. 5 as a bound on the logic of computation, will turn out to be a driving force of computation in Ch. 9.

## 3 Fixpoints

---

3.1	Computable functions have fixpoints . . . . .	50
3.2	Divergent, partial, monoidal computations . . . . .	50
3.3	The Fundamental Theorem of Computation . . . . .	53
3.3.1	Example: Polymorphic quine . . . . .	55
3.3.2	Example: Polymorphic virus . . . . .	55
3.4	Y-combinators and their classifiers . . . . .	56
3.4.1	Y-combinator constructions . . . . .	56
3.4.2	Y-combinator classifiers . . . . .	58
3.5	Software systems as systems of equations . . . . .	59
3.5.1	Smullyan fixpoints . . . . .	59
3.5.2	Systems of program equations . . . . .	60
3.6	Workout . . . . .	61
3.6.1	Divergence and lazy branching . . . . .	61
3.6.2	Stateful fixpoints . . . . .	62
3.6.3	Qing, kuine, narcissist, virus generator . . . . .	62
3.6.4	A loopy software system . . . . .	63
3.7	Stories . . . . .	63

---

### 3.1 Computable functions have fixpoints

The basic idea of the universal evaluators presented in the preceding chapter is that **a function is computable when it is programmable**. A function  $f : A \rightarrow B$  is programmable when there is a program  $F : \mathbb{P}$  such that for all  $y : A$

$$f(y) = \{F\}y \quad (3.1)$$

In this chapter we draw the main consequences of programmability. In the next chapter we will use it as a tool for programming.

The first consequence of programmability is that any computable endofunction in the form  $e : A \rightarrow A$  must have a fixpoint  $a$

$$e(y) = \{E\}y \implies \exists a. e(a) = a \quad (3.2)$$

The second consequence is the pivot point to move the world of computation. It says that for any computable function in the form  $g : \mathbb{P} \times A \rightarrow B$  there is a program  $\Gamma : \mathbb{P}$  such that

$$g(p, y) = \{G\}(p, y) \implies \exists \Gamma. g(\Gamma, y) = \{\Gamma\}y \quad (3.3)$$

This was proved in Kleene's note [89] under the name *Second Recursion Theorem*. But the use of recursion is inessential for it [161]. On the other hand, it turned out to be one of the most fundamental theorems of computability theory [110]. We call it the Fundamental Theorem of Computation. It is proved in Sec. 3.3.

### 3.2 Divergent, partial, monoidal computations

**Every computable endofunction  $e : A \rightarrow A$  has a fixpoint  $\hat{x}_e : A$ , with  $e(\hat{x}_e) = \hat{x}_e$ .** A fixpoint construction is constructed in Fig. 3.1. We precompose  $e$  with the program self-evaluation  $\{x\}x$ , and

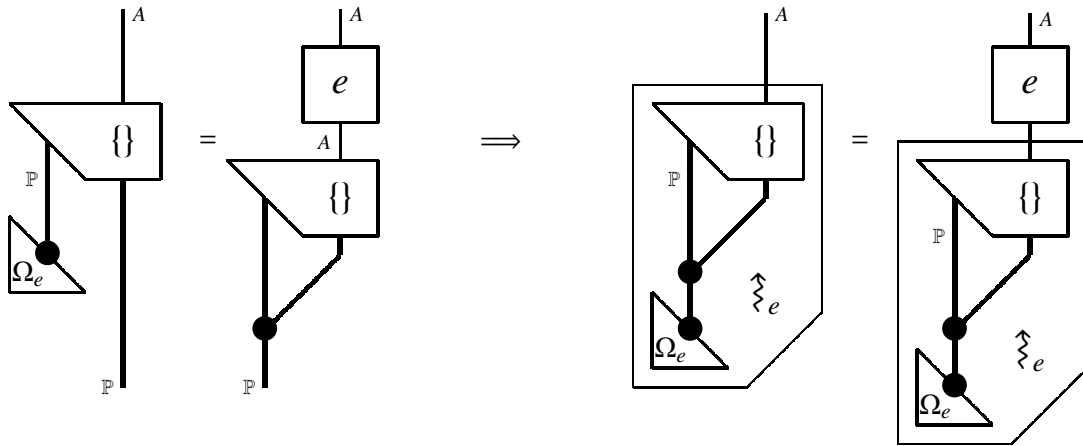


Figure 3.1: If  $\{\Omega_e\}(x) = e(\{x\}x)$ , then  $\hat{x}_e = \{\Omega_e\}(\Omega_e)$  satisfies  $e(\hat{x}_e) = e(\{\Omega_e\}(\Omega_e))$

define  $\Omega_e$  to be a program satisfying

$$\{\Omega_e\} x = e(\{x\} x) \quad (3.4)$$

as displayed in Fig. 3.1 on the left. A program  $\Omega_e$  for  $e(\{x\} x)$  must exist by the assumption that the program evaluator is universal. Setting  $x$  in (3.4) to be  $\Omega_e$  yields

$$\{\Omega_e\} \Omega_e = e(\{\Omega_e\} \Omega_e) \quad (3.5)$$

Defining  $\hat{\xi}_e = \{\Omega_e\} \Omega_e$  yields a fixpoint of  $e$ , as displayed in Fig. 3.1 on the right. Note that an overly eager program evaluator may keep evaluating a fixpoint forever:

$$\hat{\xi}_e = \{\Omega_e\} (\Omega_e) = e \{\Omega_e\} (\Omega_e) = ee \{\Omega_e\} (\Omega_e) = eee \{\Omega_e\} (\Omega_e) = \dots \quad (3.6)$$

**A divergent truth value.** The datatype  $\mathbb{B}$  of the boolean truth values was defined in Sec. 2.4.3. By definition, it contains precisely two cartesian elements  $\top, \perp \in C^\bullet(I, \mathbb{B})$ . The operation of negation  $\neg: \mathbb{B} \rightarrow \mathbb{B}$  defined in Sec. 2.4.4 satisfies  $\neg\perp = \top$  and  $\neg\top = \perp$ . On the other hand, applying the above fixpoint construction to the negation yields an element  $\hat{\xi}_\neg: \mathbb{B}$  such that

$$\neg\hat{\xi}_\neg = \hat{\xi}_\neg$$

Since  $\neg\top = \perp$  and  $\neg\perp = \top$ , it follows that  $\top \neq \hat{\xi}_\neg \neq \perp$  (unless  $\top = \perp$  and by 2.6.1a, the whole universe collapses to a point). Since  $\top$  and  $\perp$  are by the definition in Sec. 2.4.3 the only cartesian elements of  $\mathbb{B}$ , the element  $\hat{\xi}_\neg: \mathbb{B}$  cannot be cartesian. In summary, we have

$$C^\bullet(I, \mathbb{B}) = \{\top, \perp\} \quad C(I, \mathbb{B}) \supseteq \{\top, \perp, \hat{\xi}_\neg, \dots\} \quad (3.7)$$

A computer  $C$  always contains divergent computations, and is strictly monoidal, larger than its cartesian core  $C^\bullet$ , or else it collapses to a point.

**Divergent and partial elements.** In general, an element  $a: A$ , i.e. a function  $a: I \rightarrow A$ , is total if  $\left(I \xrightarrow{a} A \xrightarrow{\bullet} I\right) = \text{id}_I$ . This is an instance of the left-hand requirement of (1.4). An element  $a: A$  is called *partial* if

$$\left(I \xrightarrow{a} A \xrightarrow{\bullet} I\right) \neq \text{id}_I \quad (3.8)$$

It was noted in Sec. 1.4 that  $\mathbf{!}_A$  is the only inhabitant of the hom-set  $C^\bullet(A, I)$ , for any type  $A$ . In particular,  $\mathbf{!}_I = ()$  is only inhabitant of  $C^\bullet(I, I)$ . But the identity must be there as well, so  $\text{id}_I = \mathbf{!}_I = ()$ . On the other hand,  $\mathbf{!}_I = \left(I \xrightarrow{\hat{\xi}_\neg} \mathbb{B} \xrightarrow{\bullet} I\right) \neq \text{id}_I$  gives

$$C^\bullet(I, I) = \{\mathbf{!}_I\} \quad C(I, I) \supseteq \{\mathbf{!}_I, \mathbf{!}_I, \dots\} \quad (3.9)$$

and  $\mathbf{!}_A = \left(A \xrightarrow{\mathbf{!}_A} I \xrightarrow{\mathbf{!}_I} I\right)$  moreover gives

$$C^\bullet(A, I) = \{\mathbf{!}_A\} \quad C(A, I) \supseteq \{\mathbf{!}_A, \mathbf{!}_A, \dots\} \quad (3.10)$$

This leaves us with some interesting string diagrams. Since the identity function of type  $A$  is conve-

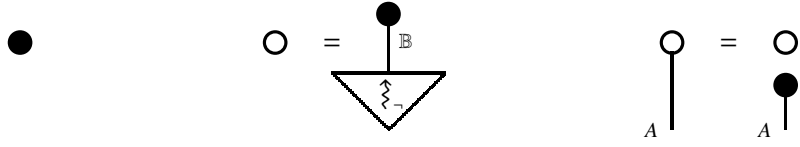


Figure 3.2: Some string diagrams with few strings:  $\bullet_I$  and  $\circ_I$  have none.

niently represented as the invisible box on the string  $A$ , and the identity type  $I$  is conveniently represented as the invisible string,  $\text{id}_I$  is very invisible. When it needs to be seen  $\text{id}_I = \bullet_I$  helps. Since each  $\bullet_A$  is a black bead on top of a string,  $\bullet_I$  is a lonely black bead on top of an invisible string, in Fig. 3.2 on the left. The partial element  $\circ_I = \bullet_B \circ \hat{\zeta}_-$  is then conveniently represented as a lonely white bead on top of an invisible string, in the middle Fig. 3.2. Lastly, the partial element  $\circ_A = \circ_I \circ \bullet_A$  is represented as a white bead on top of the string  $A$  in Fig. 3.2 on the right.

**A divergent program.** The main tool for constructing the fixpoints above is the self-evaluator  $\{x\}x$ . Instantiating (3.4) to  $e(x) = \{x\}x$  leads to  $e(\{x\}x) = \{\{x\}x\}(\{x\}x)$ . Setting  $\{\Omega\}x = \{\{x\}x\}(\{x\}x)$ , the

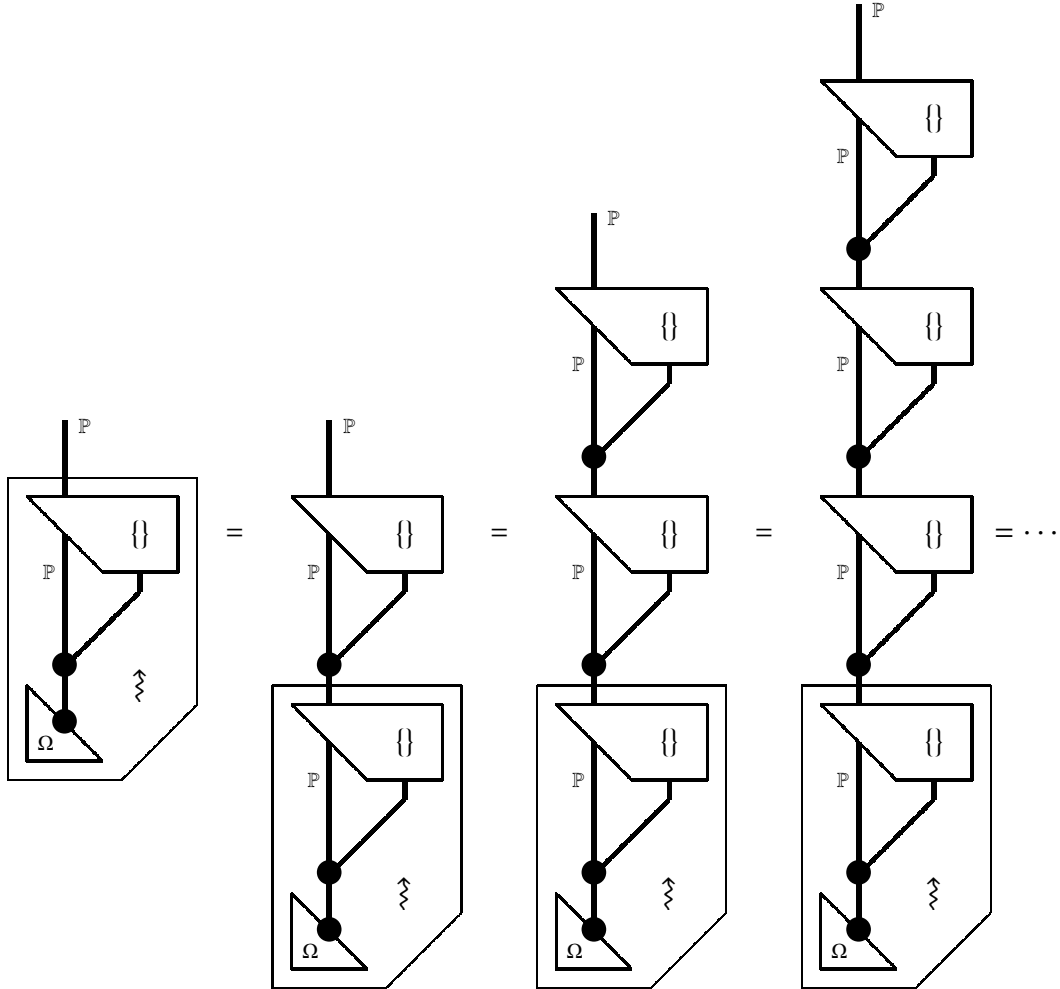


Figure 3.3: If  $\{\Omega\}x = \{\{x\}x\}(\{x\}x)$ , then  $\hat{\zeta} = \{\Omega\}\Omega$  gives  $\{\hat{\zeta}\}\hat{\zeta} = \{\{\hat{\zeta}\}\hat{\zeta}\}(\{\hat{\zeta}\}\hat{\zeta})$

evaluation process in (3.6) becomes

$$\hat{\Omega} = \{\Omega\} \Omega = \{\{\Omega\} \Omega\} (\{\Omega\} \Omega) = \{\{\{\Omega\} \Omega\} (\{\Omega\} \Omega)\} (\{\Omega\} \Omega) (\{\Omega\} \Omega) = \dots \quad (3.11)$$

If the programs are evaluated bottom-up, i.e. if every input must be evaluated before it is passed to a computation above it, then the computation in Fig. 3.3 goes on forever, as do those in Fig. 3.1. Such computations *diverge*. Can we refine the evaluation strategies to recognize and avoid the divergences? This idea led Alan Turing to the *Halting Problem*, that will be discussed in Sec. 5.3.

**Computations are monoidal.** The upshot of this section is that the fixpoints make computable functions and computable types inexorably partial. E.g., in addition to the truth values  $\top$  and  $\perp$ , the type  $\mathbb{B}$  must contain a partial element  $\hat{\Omega}$  as a fixpoint of the negation  $\neg$ . In mathematics, universes where programs and computations are not a central concern are usually introduced as universes of cartesian functions, with a unique output on each input. Partial functions, multi-valued functions, randomized functions, etc., are then captured using additional structures [33, 119, 109]. In universes where programs and computations are a central concern, partial functions are the first class citizens, and cartesianness arises as a special property. Computable functions are generally just monoidal and not cartesian in the sense that they satisfy just the monoidal laws from Sec. 1.3 but not the cartesian laws from Sec. 1.4. A monoidal computer where all computations are total and single-valued and  $C = C^\bullet$ , then we are looking at a model where all intensional aspects of computation have been projected away, and the programs for a given function are indistinguishable.

### 3.3 The Fundamental Theorem of Computation

**Kleene fixpoints.** A Kleene fixpoint of a computation  $g: \mathbb{P} \times A \rightarrow B$  is a program  $\Gamma$  that encodes  $g$  evaluated on  $\Gamma$ , as displayed in Fig. 3.4.

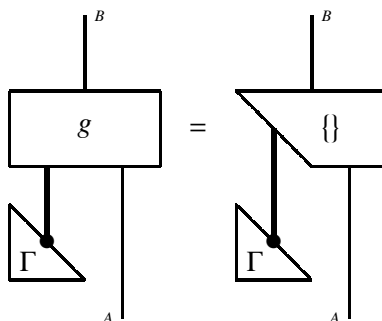


Figure 3.4: A Kleene fixpoint  $\Gamma: \mathbb{P}$  of  $g$  is a program such that  $g(\Gamma, y) = \{\Gamma\} y$

**Theorem.** Every computation that takes a program as an input, has a Kleene fixpoint for that input.

**Construction.** Towards a Kleene fixpoint of any given function  $g(p, a)$ , where  $p$  should be a program, substitute for  $p$  a computation  $[x] x$ , evaluating programs on themselves, like in Fig. 3.5. Call a program for the composite  $G$ . Then a Kleene fixpoint  $\Gamma$  of  $g$  can be defined to be a partial evaluation of  $G$  on itself, as displayed in Fig. 3.6. The diagram in the figure shows that the program  $\Gamma = [G] G$  is a fixpoint of  $g$  because

$$g(\Gamma, y) = g([G] G, y) = \{G\} (G, y) = \{[G] G\} y = \{\Gamma\} y$$

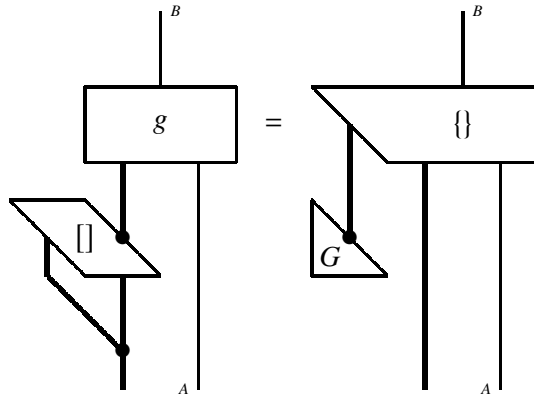


Figure 3.5:  $G$  is a program for  $g([x] x, y) = \{G\}(x, y)$

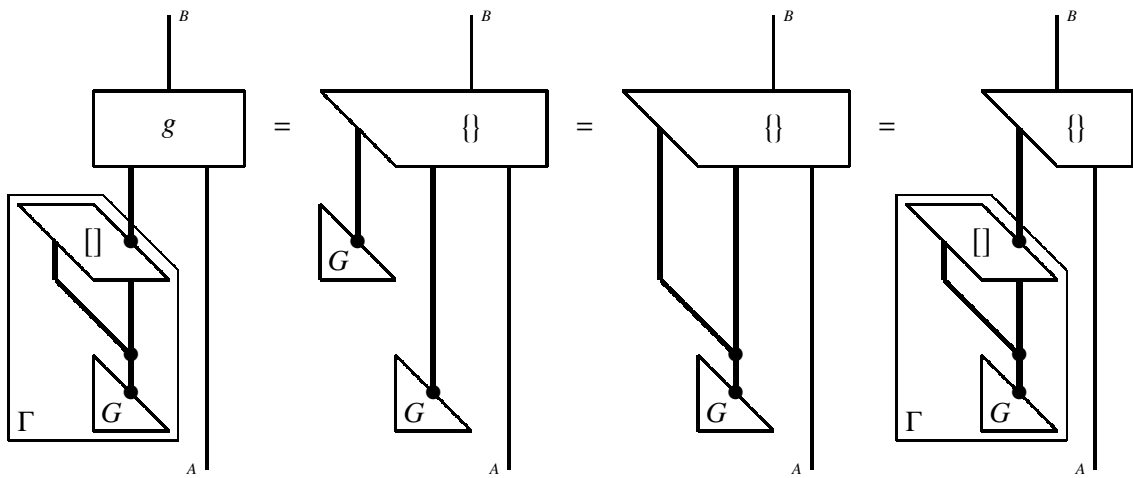


Figure 3.6: A Kleene fixpoint  $\Gamma$  of a computable function  $g$ .

**Program transformer fixpoints.** A program transformer is a cartesian function  $\gamma : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  that transforms programs. The Fundamental Theorem of Computation is equivalent to the statement that every program transformer has a fixpoint, a program  $\Gamma$  that does the same as its transform, as displayed in Fig.3.7. A transformer fixpoint can be constructed as a Kleene fixpoint of  $g(p, x) = \{\gamma p\} x$ , by applying

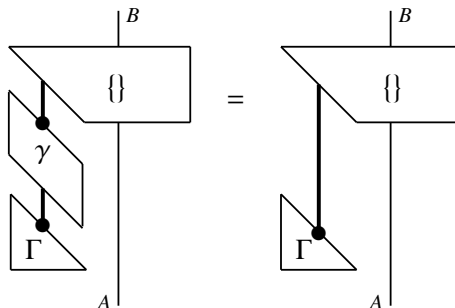


Figure 3.7: Any program transformer  $\gamma$  has a fixpoint  $\Gamma : \mathbb{P}$  such that  $\{\gamma \Gamma\} = \{\Gamma\}$



the Fundamental Theorem. The other way around, for any computable function  $g: \mathbb{P} \times A \rightarrow B$ , (2.2) gives a  $\mathbb{P}$ -indexed program  $G: \mathbb{P} \rightarrow \mathbb{P}$  such that  $g(p, a) = \{G_p\} a$ . As a cartesian function, any  $\mathbb{P}$ -indexed program can also be viewed as a program transformer. By assumption, it has a fixpoint  $\Gamma$  with  $\{\Gamma\} a = \{G_\Gamma\} a = g(\Gamma, a)$ . So  $\Gamma$  is a Kleene fixpoint of  $g$ .

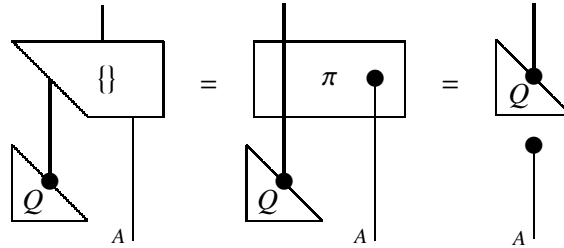
### 3.3.1 Example: Polymorphic quine

A *quine* is a program  $Q$  that, when executed, outputs its own text:

$$\{Q\} x = Q$$

A simple program that outputs a text  $Q$  is usually in the form “print ‘ $Q$ ’”. But that program is obviously longer than  $Q$  and does not output all of its text. A quine cannot contain its own text as a quote, but has to somehow compress a version of itself, and decompress it at the output. The entries in the annual quine competitions mostly use peculiarities of particular programming languages to achieve this. The Fundamental Theorem of Computation allows constructing *polymorphic* quines, that can be constructed in all programming languages. The Fundamental Theorem provides a *polymorphic* quine construction, implementable in any programming language<sup>1</sup> A quine can be obtained as a Kleene fixpoint  $Q$  of the projection  $\pi_{\mathbb{P}}: \mathbb{P} \times A \rightarrow \mathbb{P}$ , where  $\pi_{\mathbb{P}}(x, y) = x$ , so that

$$\{Q\} y = \pi_{\mathbb{P}}(Q, y) = Q$$



Unfolding Fig. 3.6 for  $g = \pi_{\mathbb{P}}$  shows that  $Q$  is the partial evaluation  $[\Pi] \Pi$ , where  $\Pi$  is a program for  $\pi_{\mathbb{P}}([x] x, y)$ . The function  $\{Q\} y = \{[\Pi] \Pi\} y = \{\Pi\} (\Pi, y)$  thus deletes  $y$  and partially evaluates  $\Pi$  on itself. The program  $Q$  thus only contains  $\Pi$ . Running  $Q$  outputs  $Q$  as the partial evaluation of  $\Pi$  on  $\Pi$ .

### 3.3.2 Example: Polymorphic virus

A *virus* is a program  $V$  that performs some arbitrary computation  $f: A \rightarrow B$  and moreover outputs copies of itself:

$$\{V\} x = \langle V, V, f(x) \rangle$$

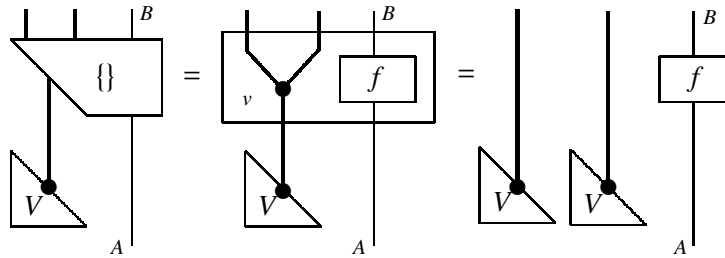
The Fundamental Theorem of Computation yields again. Apply the Fundamental Theorem to

$$v(p, x) = \langle p, p, f(x) \rangle$$

<sup>1</sup>It is assumed that all programming languages are Turing complete, in the sense of Sec. 4.6. The languages that are not Turing complete do not allow computing all computable functions.

The Kleene fixpoint is then a virus  $V$ :

$$\{V\}x = v(V, x) = \langle V, V, f(x) \rangle$$



Explain how  $V$  produces the two copies of itself without storing a copy of itself anywhere.

### 3.4 $\Upsilon$ -combinators and their classifiers

The Fundamental Theorem of Computation constructs a Kleene fixpoint of computable functions. But since computable functions are programmable, the Fundamental Theorem can be implemented as a computable program transformer, that inputs programs and outputs Kleene fixpoints of the computations that they implement. The programs that implement such program transformers are called the  $\Upsilon$ -combinators. They are constructed as Kleene fixpoints of suitable computable functions. There are many different ways to construct them. Interestingly, there are also programs that recognize all possible  $\Upsilon$ -combinators. They are called the  $\Upsilon$ -combinator classifiers. And yes, they are also constructed as Kleene fixpoints.

The circuitous constructions of the various fixpoint constructions continue to sound complicated, and there is a sense in which they are complex when presented sequentially. But their diagrammatic programs tend to be simple and insightful.

#### 3.4.1 $\Upsilon$ -combinator constructions

**$\Upsilon$ -combinator for fixpoints** is a program  $\Upsilon$  for a function  $\dot{v} : \mathbb{P} \rightarrow A$  that inputs programs for functions  $A \rightarrow A$  and outputs their fixpoints  $\dot{v}(p) : A$ , as in Sec. 3.2. The equation

$$\{p\}(\dot{v}(p)) = \dot{v}(p) \quad (3.12)$$

thus holds for all  $p : \mathbb{P}$ . The  $\dot{\Upsilon}$ -combinator is constructed as the Kleene fixpoint in

$$(3.13)$$

**$\Upsilon$ -combinator for fixed computations** is a program  $\dot{\Upsilon}$  for a program transformer  $\ddot{v} : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  that inputs programs for functions  $A \longrightarrow A$  and outputs programs for some fixed functions  $A \longrightarrow A$ . More precisely, for every  $p : \mathbb{P}$ , the program  $\ddot{v}(p) : \mathbb{P}$  satisfies

$$\{p\} \circ \{\ddot{v}(p)\} = \{\ddot{v}(p)\} \quad (3.14)$$

The  $\dot{\Upsilon}$ -combinator is obtained by applying the Fundamental Theorem again.

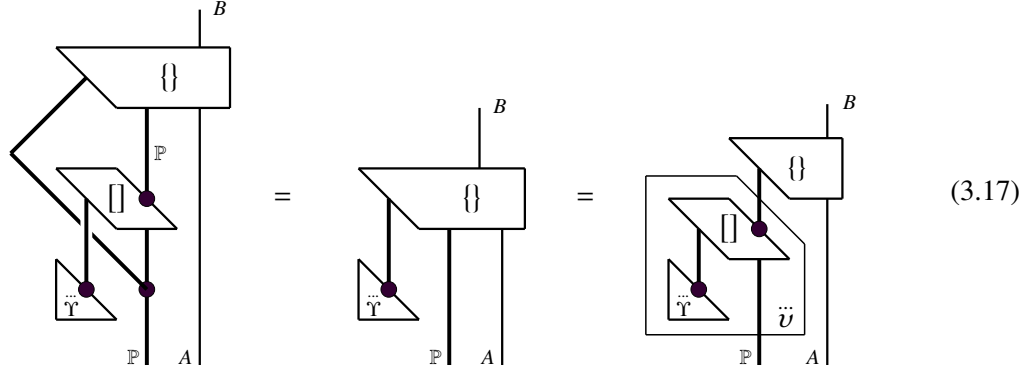
$$(3.15)$$

**$\Upsilon$ -combinator for Kleene fixpoints** is a program  $\ddot{\Upsilon}$  for a program transformer  $\ddot{\ddot{v}} : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  which inputs programs and outputs their Kleene fixpoints as in Sec. 3.3, i.e. satisfies

$$\{p\}(\ddot{\ddot{v}}(p), x) = \{\ddot{\ddot{v}}(p)\}(x) \quad (3.16)$$

for all  $p : \mathbb{P}$  and  $x : A$ . Here is a Kleene fixpoint construction of a program for a uniform Kleene fixpoint

constructor:



### 3.4.2 $\Upsilon$ -combinator classifiers

For any pair of types  $A, B$  there are program transformers  $\dot{\psi}, \ddot{\psi}, \ddot{\ddot{\psi}} : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  whose fixed points are exactly the corresponding fixpoint operators, i.e. for every  $y \in \mathbb{P}$  the following equivalences hold

$$\begin{array}{ccc} \{\dot{\psi}(y)\} = \{y\} & \{\ddot{\psi}(y)\} = \{y\} & \{\ddot{\ddot{\psi}}(y)\} = \{y\} \\ \Updownarrow & \Updownarrow & \Updownarrow \\ \forall p. \{p\}(\{y\} p) = \{y\} p & \forall p. \{p\} \circ \{y\} p = \{y\} p & \forall p. \{p\}(\{y\} p, x) = \{y\} p(x) \end{array} \quad (3.18)$$

**Classifier of  $\Upsilon$ -combinators for fixpoints.** Consider the equivalence in (3.18) on the left. The right-hand sides of the two equivalent equations become equal if both sides of the first equation are applied to the argument  $p$ . The equivalence will then hold if and only if the left-hand sides of the equations are equal. It is thus required that  $\dot{\psi} : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  satisfies

$$\{\dot{\psi}(y)\} p = \{p\}(\{y\} p) \quad (3.19)$$

This will hold if  $\dot{\psi}(y) = [\Psi]y$  where  $\Psi$  is a program for

$$\{\Psi\}(y, p) = \{p\}(\{y\} p) \quad (3.20)$$

Note that the  $\Upsilon$ -combinator was defined in (3.13) as a Kleene fixpoint in  $y$  of the function in (3.20).

**Classifier of  $\Upsilon$ -combinators for fixed functions.** To realize the equivalence in the middle of (3.18), we apply both sides of the first equation to  $p$  again, and run the output as a program on  $x$ . If we run both sides of the second equation in the middle of (3.18) on  $x$  as well, then the right-hand sides of the two equivalent equations become identical again. For the equivalence to hold, it is now necessary and sufficient that  $\ddot{\psi} : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  satisfies

$$\{\{\ddot{\psi}(y)\}p\}(x) = \{p\} \circ \{y\} p(x) \quad (3.21)$$

To specify  $\check{\Psi}$ , this time we need the programs  $\check{\Psi}_0$  and  $\check{\Psi}_1$  which implement the functions on the right in

$$\{\check{\Psi}_0\}(y, p, x) = \{p\} \circ \{y\} p(x) \quad (3.22)$$

$$\{\check{\Psi}_1\}(y, p) = [\check{\Psi}_0](y, p) \quad (3.23)$$

to define  $\check{\Psi}(y) = [\check{\Psi}_1]y$ . The  $\check{\Upsilon}$ -combinator was defined in (3.15) as a Kleene fixpoint in  $y$  of the function in (3.22).

**Classifier of  $\Upsilon$ -combinators for Kleene fixpoints.** Towards the equivalence in (3.18) on the right, proceeding as above we find that it is necessary and sufficient that  $\check{\check{\Psi}} : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  satisfies

$$\{\{\check{\check{\Psi}}(y)\}p\}(x) = \{p\}(\{y\} p, x) \quad (3.24)$$

We define  $\check{\check{\Psi}}_0, \check{\check{\Psi}}_1 : \mathbb{P}$  by

$$\{\check{\check{\Psi}}_0\}(y, p, x) = \{p\}(\{y\} p, x) \quad (3.25)$$

$$\{\check{\check{\Psi}}_1\}(y, p) = [\check{\check{\Psi}}_0](y, p) \quad (3.26)$$

and set  $\check{\check{\Psi}}(y) = [\check{\check{\Psi}}_1]y$ . The  $\check{\check{\Upsilon}}$ -combinator was defined in (3.17) as a Kleene fixpoint in  $y$  of the function in (3.25).

## 3.5 Software systems as systems of equations

A software system is a family of programs that work together. Since programs in practice usually work together, most programs belong in some software systems, and most programmers work as software developers. Keeping the programs in a software system together also requires a lot of *metaprogramming*, of programs that compute programs. We get to metaprogramming and software engineering in Ch. 6. But mutual dependencies of programs that work together arise already on the level of fixpoints. The task of finding joint fixpoints of functions arises in all engineering disciplines, and has been one of the central themes of mathematics. It is usually presented as the task of solving systems of equations. Here we sketch a basic method for solving systems of computable equations that can be used to specify some joint requirements from systems of programs. The upshot is that any family of computations can be bundled together into a software system: there is always a family of programs that code them together. The idea and the main theoretical developments go back to Raymond Smullyan [154, Ch. IX]. The brief diagrammatic treatment and the simplicity of applications conceal the significant obstacles that had to be surmounted.

### 3.5.1 Smullyan fixpoints

*For any pair of computations*

$$g : \mathbb{P} \times \mathbb{P} \times A \longrightarrow B \quad \text{and} \quad h : \mathbb{P} \times \mathbb{P} \times C \longrightarrow D$$

there is a pair of programs  $G, H : \mathbb{P}$  such that

$$g(G, H, x) = \{G\}x \quad \text{and} \quad h(G, H, y) = \{H\}y$$

**Constructing Smullyan fixpoints.** Let  $\widehat{G}$  and  $\widehat{H}$  be the Kleene fixed points of the functions  $\widehat{g} : \mathbb{P} \times \mathbb{P} \times A \rightarrow B$  and  $\widehat{h} : \mathbb{P} \times \mathbb{P} \times C \rightarrow D$  defined

$$\widehat{g}(p, q, x) = g([p]q, [q]p, x) \quad (3.27)$$

$$\widehat{h}(q, p, y) = h([p]q, [q]p, y) \quad (3.28)$$

and define

$$G = [\widehat{G}] \widehat{H} \quad H = [\widehat{H}] \widehat{G}$$

The constructions now give

$$g(G, H, x) = g([\widehat{G}] \widehat{H}, [\widehat{H}] \widehat{G}, x) \stackrel{(3.27)}{=} \widehat{g}(\widehat{G}, \widehat{H}, x) = \{\widehat{G}\}(\widehat{H}, x) = \{[\widehat{G}] \widehat{H}\}x = \{G\}x \quad (3.29)$$

$$h(G, H, y) = h([\widehat{G}] \widehat{H}, [\widehat{H}] \widehat{G}, y) \stackrel{(3.28)}{=} \widehat{h}(\widehat{H}, \widehat{G}, y) = \{\widehat{H}\}(\widehat{G}, y) = \{[\widehat{H}] \widehat{G}\}y = \{H\}y \quad (3.30)$$

The concept behind this algebraic magic emerges from the pictures of the functions  $\widehat{g}$  and  $\widehat{h}$ , and of their

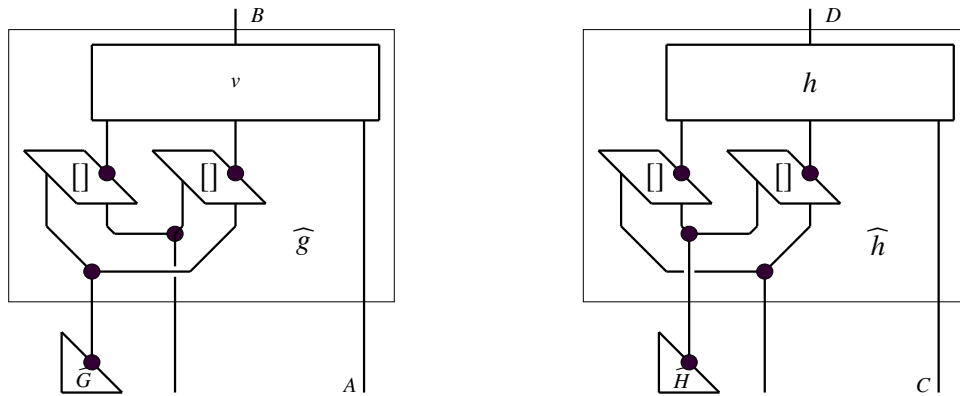


Figure 3.8: Definitions of  $\widehat{G}$  and  $\widehat{H}$  as Kleene fixpoints

fixpoints  $\widehat{G}$  and  $\widehat{H}$  in Fig. 3.8. The diagrammatic form of (3.29) in Fig. 3.9 shows how the construction follows the idea of the Fundamental Theorem, or more precisely of the construction of the Kleene fixpoint in Fig. 3.6.

### 3.5.2 Systems of program equations

For any  $n$ -tuple of computations

$$g_i : \mathbb{P}^n \times A_i \rightarrow B_i \quad \text{for } i = 1, 2, \dots, n$$

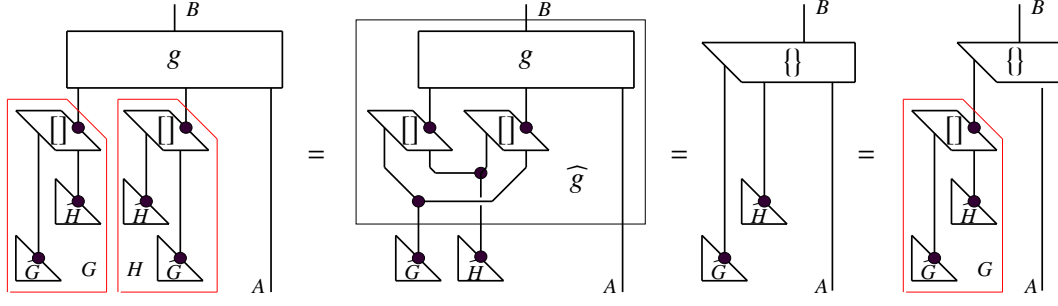


Figure 3.9: A geometric view of (3.29)

each depending on  $n$  programs, there are programs  $G_1, G_2, \dots, G_n : \mathbb{P}$  that are fixed by the given computations, in the sense of the following equations:

$$\begin{aligned} g_1(G_1, G_2, \dots, G_n, x_1) &= \{G_1\} x_1 \\ g_2(G_1, G_2, \dots, G_n, x_2) &= \{G_2\} x_2 \\ &\vdots \\ g_n(G_1, G_2, \dots, G_n, x_n) &= \{G_n\} x_n \end{aligned}$$

**Solving systems of computable equations.** Define  $\widehat{g}_i : \mathbb{P}^n \times A_i \longrightarrow B_i$  by

$$\widehat{g}_i(p_1, p_2, \dots, p_n, x_i) = g_i([p_1] \vec{p}_{-1}, [p_2] \vec{p}_{-2}, \dots, [p_n] \vec{p}_{-n}, x_i)$$

where  $\vec{p}_{-i} = (p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n)$ . Furthermore set

$$G_i = [\widehat{G}_i](\widehat{G}_1, \dots, \widehat{G}_{i-1}, \widehat{G}_{i+1}, \dots, \widehat{G}_n)$$

The constructions now give

$$\begin{aligned} g_i(G_1, G_2, \dots, G_n, x_i) &= \widehat{g}_i(\widehat{G}_1, \widehat{G}_2, \dots, \widehat{G}_n, x_i) \\ &= \{\widehat{G}_i\}(\widehat{G}_1, \dots, \widehat{G}_{i-1}, \widehat{G}_{i+1}, \dots, \widehat{G}_n, x_i) \\ &= \{[\widehat{G}_i](\widehat{G}_1, \dots, \widehat{G}_{i-1}, \widehat{G}_{i+1}, \dots, \widehat{G}_n)\} x_i \\ &= \{G_i\} x_i \end{aligned}$$

## 3.6 Workout

### 3.6.1 Divergence and lazy branching

The divergence is that any computation composed with a divergent component gets swallowed by the divergence. This is not always justified. E.g., the branching instruction  $ifte(b, \top, \mathbb{X})$  is normally expected to output  $\top$  if  $b = \top$  and to diverge if  $b = \perp$ . However, the implementation  $ifte(b, f, g) = \{b\}(f, g)$  in Sec. 2.4.1 requires that  $b$ ,  $f$ , and  $g$  are evaluated before their outputs are passed to  $ifte$ . With that

implementation,  $ifte(\top, \top, \S)$  will always diverge. To avoid this undesired behavior, most languages make use of programmability of the branches  $f$  and  $g$ , and implement a branching operation  $IFTL(b, f, g)$  which evaluates the condition  $b$  and outputs a program  $F$  for  $f = \{F\}$  when  $b = \top$  or a program  $G$  for  $g = \{G\}$  when  $b = \perp$ . In this way, only one of the branches is evaluated, and only after the branching. A more practical *lazy* branching should therefore satisfy

$$iftl(b, F, G) = \{\{b\}(F, G)\} \quad (3.31)$$

- Program a diagram for the lazy branching in (3.31).
- Specify the program transformers  $IFTE, IFTL : \mathbb{P}^3 \longrightarrow \bullet \mathbb{P}$  such that  $\{IFTE(b, F, G)\} = ifte(b, \{F\}, \{G\})$  and  $\{IFTL(b, F, G)\} = iftl(b, F, G)$ . While both evaluate  $b$ , input  $F$  and  $G$  and output a program that runs  $F$  if  $b$  is true and  $G$  if it false, they behave differently. Provide examples of different and of indifferent behaviors.
- Given an idempotent  $\rho_A : \mathbb{P} \longrightarrow \mathbb{P}$  such that  $x = \rho_A(x)$  holds just for the (programs that encode) elements of  $A$  as in Sec. 2.3.1, construct an idempotent  $\check{\rho}_A : \mathbb{P} \longrightarrow \mathbb{P}$  that still fixes the elements of  $A$  and is moreover a subfunction of the identity on  $\mathbb{P}$ . In other words, for all programs  $y$  that do not encode elements of  $A$ ,  $\check{\rho}_A(y)$  should diverge.

### 3.6.2 Stateful fixpoints

The Fundamental Theorem remains valid for stateful (indexed) functions and programs. Here we work out two kinds of stateful Kleene fixpoints.

- Show that any  $X$ -indexed computation  $g : \mathbb{P} \times X \times A \longrightarrow B$  has an  $X$ -indexed Kleene fixpoint  $\Gamma : X \longrightarrow \bullet \mathbb{P}$  such that for all  $a : A$

$$g(\Gamma x, x, a) = \{\Gamma x\}a \quad (3.32)$$

- Show that any  $X$ -indexed program transformer  $\theta : \mathbb{P} \times X \longrightarrow \bullet \mathbb{P}$  has an  $X$ -indexed fixpoint  $\Theta : X \longrightarrow \bullet \mathbb{P}$  such that

$$\{\theta(\Theta x, x)\} = \{\Theta x\}$$

### 3.6.3 Qing, kuine, narcissist, virus generator

- A *narcissist* is a program  $T$  which inputs programs and outputs its own images along the functions that the programs encode:

$$\{T\}x = \{x\}T$$

Find a narcissist!

- A *virus generator* is a program transformer  $\gamma : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  which transforms programs into viruses. Its



function is:

$$\{\gamma F\}x = \langle \gamma F, \gamma F, \{F\}x \rangle$$

Program a virus generator!

c. Show that there are programs  $Q$ ing and  $K$ uine which on any input output each other's text:

$$\{K\}x = Q \qquad \{Q\}x = K$$

### 3.6.4 A loopy software system

The *Loopy Credit Union (LCU)* account system consists of

- a database  $B = (x_u)_{u \in U}$  where
  - $U$  is the list of customers,
  - $x_u$  is the account balance of the customer  $u \in U$
  - $\ell \in U$  is the account into which the LCU collects the fees for their services;
- three types of transactions:
  - deposit  $d(u, x) = B + x_u$  for each  $u \in U$ ,
  - withdrawal  $w(u, x) = B - x_u$  for each  $u \in U$ , and
  - transfer  $t(u, v, x) = (w(u, x) ; d(v, x) ; t(v, \ell, rx))$  for every pair  $u, v \in U$ .

While the deposits and the withdrawals are free of charge, the recipient of each transfer of  $x$  coins is charged a fee of  $rx$  coins, transferred to the fee account  $\ell$ . A transfer from a customer  $u$  to a customer  $v$  is implemented as a sequential composition of a withdrawal of  $x$  from  $u$ 's account, followed by a deposit of  $x$  into  $v$ 's account, followed by a transfer of  $rx$  into  $\ell$ 's account.

Develop the LSU-software system as a fixpoint.

## 3.7 Stories

The Fundamental Theorem of Computation says that every computational process has a fixpoint. This realization evolved through several breakthrough ideas. First there was Cantor's diagonal argument [21]. Then there was Russel's application of that argument to derive his paradox from Frege's unrestricted set-comprehension [139]. Then there was Gödel's application of that construction in logic, to derive incompleteness of Russell-Whitehead's *Principia* and all other systems containing the arithmetic [58]. Then Stephen Kleene proved the Fundamental Theorem from Sec. 3.3 as a relatively minor remark in a relatively minor paper [89]. He continued to treat it as a weaker version of his (extensional) *Recursion Theorem* [91, §66], and some years later it got promoted into the "*Second Recursion Theorem*" [138]. Many years and many fundamental applications later, the name got extended to "*Kleene's Amazing Second Recursion Theorem*" [110]. Raymond Smullyan proved the double-fixpoint version from

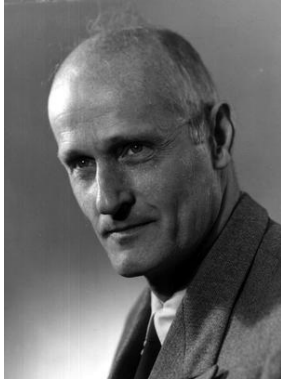


Figure 3.10: Stephen C. Kleene

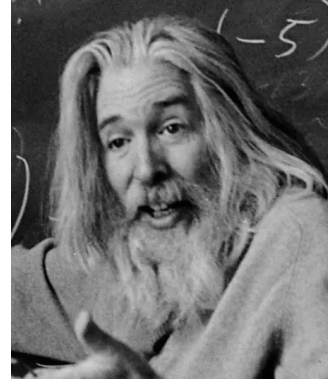


Figure 3.11: Raymond Smullyan

Sec. 3.5.1 and studied the general approach in his book about "*Diagonalization and Self-Reference*" [155]. He also wrote many popular books about the logic of self-reference [156, 157]. The topic lent itself to other popular treatments [73, 134]. In the years before his death, John von Neumann had noticed that computational self-reference implies structural self-reproduction, which is the characteristic of life. On his deathbed, he was preparing a series of lectures about life as computation [114]. The idea was that a universal computer with sensors and actuators could gather materials, process them, and produce a copy of itself. Or several copies, slightly modified, letting nature select the fittest. The computational aspects of life also preoccupied Alan Turing towards the end of his short life [168].

## 4 What can be computed

---

4.1	Reverse programming . . . . .	66
4.2	Numbers and sequences . . . . .	66
4.2.1	Counting numbers . . . . .	66
4.2.2	Numbers as sets . . . . .	66
4.2.3	Numbers as programs . . . . .	68
4.2.4	Type $\mathbb{N}$ as an idempotent . . . . .	69
4.2.5	Sequences . . . . .	71
4.3	Counting down: Induction and recursion . . . . .	71
4.3.1	Computing by counting . . . . .	71
4.3.2	Induction . . . . .	72
4.3.3	Reverse programming induction . . . . .	73
4.3.4	Recursion . . . . .	73
4.3.5	Running recursion . . . . .	74
4.4	Counting up: Search and loops . . . . .	78
4.4.1	Search . . . . .	78
4.4.2	Loops . . . . .	80
4.5	Workout . . . . .	81
4.5.1	Early induction . . . . .	81
4.5.2	Recursion work . . . . .	82
4.5.3	Search work . . . . .	82
4.6	Stories . . . . .	83
4.6.1	Church's reverse programming . . . . .	83
4.6.2	Story of curly brackets . . . . .	84
4.6.3	The Church-Turing Thesis . . . . .	84
4.6.4	Turing completeness . . . . .	86

---

## 4.1 Reverse programming

In the practice of computation, programmers are given a programming language, with a basic type system and some program schemas, and they are asked to build programs and run program evaluators. The theory of computation goes the other way around: it starts from program evaluators and builds a basic type system and program schemas. This is the idea of *reverse programming*. In the preceding chapter, we used the program evaluators to construct various fixpoints. In the present chapter, we construct the datatype  $\mathbb{N}$  of natural numbers, with the basic arithmetic operations and the main program schemas that it supports. The program structures are derived from program evaluators and data services, and carried forward as syntactic sugar. Behind the sugar, programs are expressions built from typed instances of a single instruction `run`, composed using the data services. Such expressions comprise the Turing-complete programming language `Run`. This chapter spells out the basic program schemas in this language, and explains what it means to be Turing-complete in the end.

## 4.2 Numbers and sequences

### 4.2.1 Counting numbers

Numbers are built by counting. Counting is the physical process of assigning fingers or pebbles to apples, or sheep, or coins. Counting is the earliest form of computation. Our fingers were the first digital computers<sup>1</sup>. The first algorithm was the assignment of 10 fingers to 10 apples or sheep. The second algorithm may have been the idea to count the apples or sheep by the 5 fingers of one hand, and to use the 5 fingers of the other hand to count how many times the first hand was used. That would allow us to count up to 25 apples, instead of just 10. The *fourth* algorithm could be to count up to 1024 apples using the 10 fingers. Before that, someone should have discovered a *third* algorithm, allowing them to count up to 36 with 10 fingers. How would you do that? The idea is that a hand with 5 flexible fingers has 6 states, and not just 5, since it can stretch 0, 1, 2, 3, 4, or 5 fingers, while keeping the rest bent. The step to counting  $2^{10} = 1024$  apples using the 10 fingers follows from the observation that each finger has 2 states: stretched and bent. In any case, noticing 0, as the state of where nothing has been counted, and using it in counting, turned out to be a revelation.

**Only natural numbers.** The numbers built by counting are studied as the *natural* numbers. The integers are built by inverting the addition of natural numbers, the rationals by inverting the multiplication, and so on [37]. For the moment, we focus on counting, and the present discussions about "numbers" invariably refer to the *natural numbers* from school.

### 4.2.2 Numbers as sets

If the counting process is required to produce an output, then it must involve from 0, as the state where there is nothing left to count, and the output is ready. 0 is the number of elements of the empty set, which we write  $\{\}$ . Since the empty set is unique, it is the best representative of the number of its elements, so we set  $0 = \{\}$ . Nonempty sets are formed by enclosing some given elements between the curly brackets, like in  $\{a, b, c\}$ . The elements can be some previously formed sets. So we can form the set  $\{\{\}\}$ , with a single element  $\{\}$ . We set  $1 = \{0\}$  and use it to denote the number of elements in all sets with a single

---

<sup>1</sup>The Latin word "digitus" means "finger".

element. There may be many sets with a single element, as many as there are elements given in the world; but only one number 1. But even if no elements were given, we certainly have the sets  $0 = \{\}$  and  $1 = \{\{\}\}$ , which are different, since one is empty and the other is not. Now we can form the set  $2 = \{0, 1\}$ , with two different elements. Continuing like this, we form the sequence of numbers

$$\begin{aligned}
 0 &= \{\} \\
 1 &= \{0\} = \{\{\}\} \\
 2 &= \{0, 1\} = \{\{\}, \{\{\}\}\} \\
 3 &= \{0, 1, 2\} = \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\} \\
 4 &= \{0, 1, 2, 3\} = \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}, \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}\} \\
 &\dots
 \end{aligned}
 \tag{4.1}$$

$$\begin{aligned}
 n &= \{0, 1, 2, \dots, n-1\} \\
 n+1 &= \{0, 1, 2, \dots, n-1, n\}
 \end{aligned}
 \tag{4.2}$$

After a long history of troubles with numbers, from the Pythagoreans to Frege, the idea of counting starting from nothing was proposed by a 17-year-old Hungarian wunderkind Neumann János, who later became John von Neumann. We saw him in Fig. 2.15, but see Appendix 3 for an earlier photo. The process of counting was thus implemented as a process of set building. The numbers are built starting from nothing, i.e. the empty set as the **base case**, and adding at each **step case** a new element, obtained as the set of all previously built numbers:

$$0 = \{\} \qquad n+1 = n \cup \{n\}
 \tag{4.3}$$

Since all numbers are thus generated from 0 by the operation  $1 + (-)$ , they must all be in the form

$$n = \underbrace{1 + 1 + \dots + 1}_{n \text{ times}} + 0
 \tag{4.4}$$

In the computer, each number  $n$  is thus in principle a wire with  $n$  beads, as depicted in Fig. 4.1.

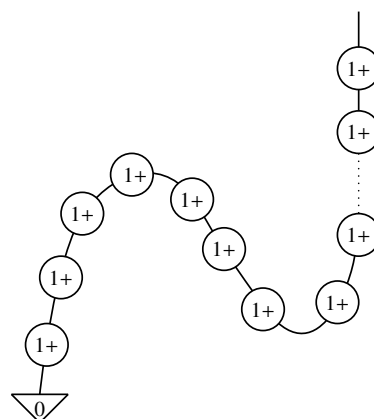


Figure 4.1: The number  $n$  is  $n$ -th successor of 0.



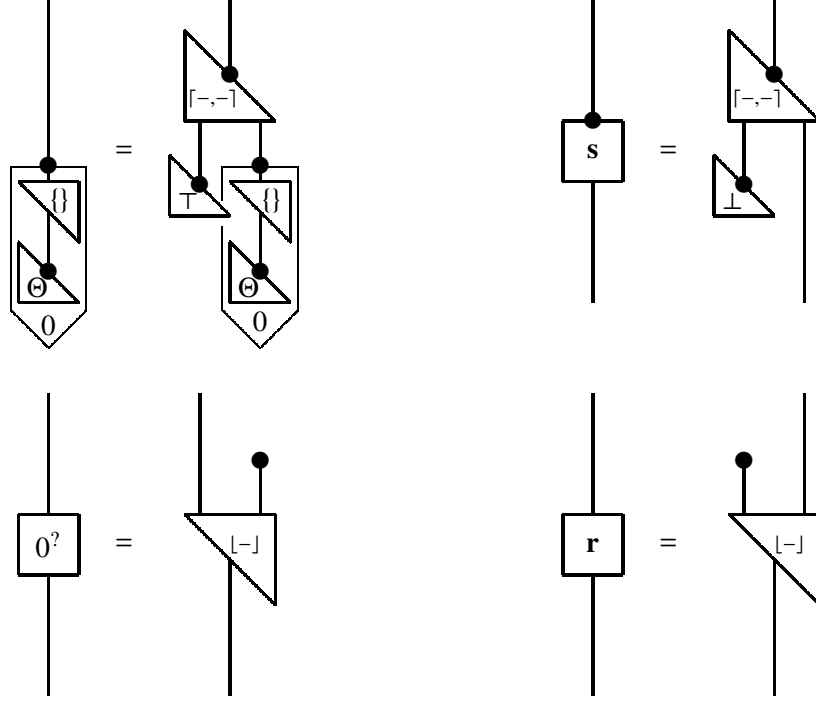


Figure 4.2: Base  $\bar{0} = \{\Theta\} = [\top, \{\Theta\}]$ , step up  $s(x) = [\perp, x]$ , step down  $r(x) = [x]_1$ , test  $0^2 x = [x]_0$

- (b)  $\implies$  (c) For  $f', g' : \mathbb{P} \longrightarrow \mathbb{P}$ , the assumption  $\top = \perp$  gives

$$f = \{\top\}(f, g) = \{\perp\}(f, g) = g$$

By (25) and (26), any pair  $f, g : A \longrightarrow B$  induces  $\bar{f} = \mathbf{i}_B \circ f \circ \mathbf{q}_A$  and  $\bar{g} = \mathbf{i}_B \circ g \circ \mathbf{q}_A$  such that  $\bar{f} = \bar{g}$  if and only if  $f = g$ . But if all functions are equal, then all type retractions are isomorphisms. Since by Sec. 2.3.2 all types are retracts of  $\mathbb{P}$ , it follows that all types are isomorphic, and thus isomorphic to the unit type  $I$ .

#### 4.2.4 Type $\mathbb{N}$ as an idempotent

The task is now to program an idempotent  $\mathbb{N} : \mathbb{P} \longrightarrow \mathbb{P}$  to filter out the numbers in the form (4.5) from other programs, so that  $\bar{n} : \mathbb{N}$  is captured as  $\bar{n} = \mathbb{N}(\bar{n})$ . The idea is to map each  $x : \mathbb{P}$  as follows:

- if  $x = \bar{0}$ , then output  $\mathbb{N}(x) = x$ ;
- if  $x = [\perp, y]$ , then output  $\mathbb{N}(x) = x$  if and only if  $\mathbb{N}(y) = y$ .

Since  $y = \mathbb{N}(z)$  satisfies  $\mathbb{N}(y) = y$  for any  $z : \mathbb{P}$ , this can be implemented informally as

$$\mathbb{N}(x) = \begin{cases} x & \text{if } x = \bar{0} \\ [\llbracket x \rrbracket_0, \mathbb{N}[\llbracket x \rrbracket_1]] & \text{if } \llbracket x \rrbracket_0 = \perp \\ \text{?} & \text{otherwise} \end{cases} \quad (4.8)$$

Towards a formalization, this can be rewritten in terms of the operations available in the monoidal computer as the function  $\nu : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$  where

$$\nu(p, x) = \text{IFTE}\left(x \stackrel{?}{=} \bar{0}, x, \text{IFTE}(\lfloor x \rfloor_0 \stackrel{?}{=} \perp, s \circ \{p\} \circ r(x), \hat{x})\right) \quad (4.9)$$

Here we use the lazy *IFTE*-branching from 3.6.1 to avoid evaluating  $\hat{x}$  when it is not selected. For a Kleene fixpoint  $N : \mathbb{P}$  of  $\nu$ , setting  $\mathbb{N} = \{N\}$  gives

$$\mathbb{N}(x) = \{N\} x = \nu(N, x) = \text{IFTE}\left(x \stackrel{?}{=} \bar{0}, x, \text{IFTE}(\lfloor x \rfloor_0 \stackrel{?}{=} \perp, s(\mathbb{N}(r(x))), \hat{x})\right)$$

It easy to prove that  $\mathbb{N}$  is a subfunction of the identity, and that it produces an output when the input is a number:

$$\mathbb{N}(x) = y \iff \exists n. x = s^n(\bar{0}) = y$$

A convenient order of evaluation of  $\nu$ , not obvious in the above command-line form, is displayed in Fig. 4.3.

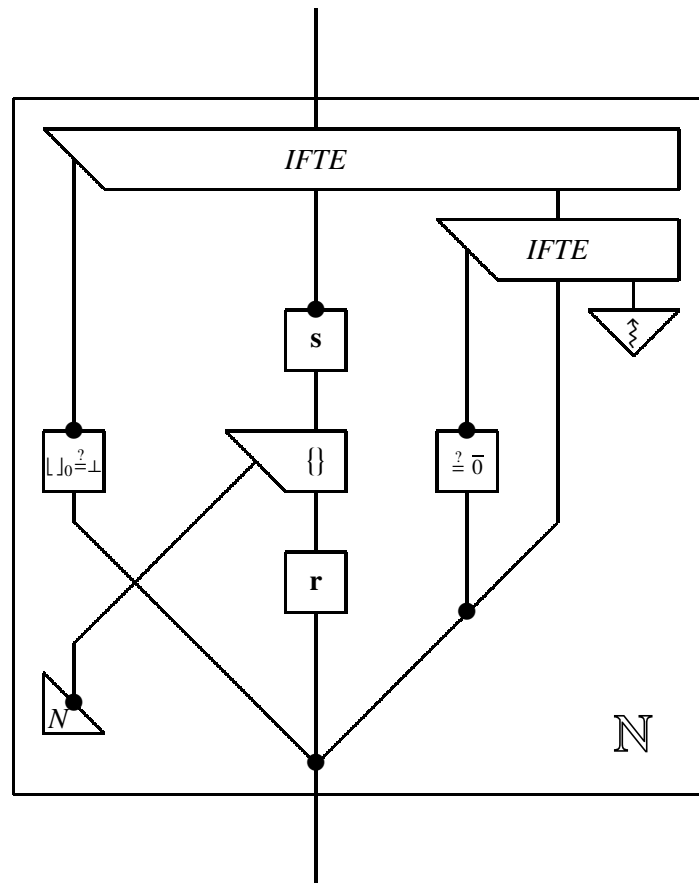


Figure 4.3: The idempotent  $\mathbb{N} : \mathbb{P} \rightarrow \mathbb{P}$  for the datatype of natural numbers

**Remark.** Replacing  $x \stackrel{?}{=} \bar{0}$  with  $0^?x$  and  $\lfloor x \rfloor_0 \stackrel{?}{=} \perp$  with  $\neg \lfloor x \rfloor_0$  would yield a type of *intensional*



naturals, still supporting the same operations, but with multiple representatives of each number. The present definition fixes unique representatives as soon as the representatives of 0 and  $\perp$  are chosen.

## 4.2.5 Sequences

A cartesian function  $\beta : \mathbb{N} \longrightarrow B$ , taking numbers as inputs, is usually presented by the *sequence* of its outputs

$$(\beta_0, \beta_1, \beta_2, \dots, \beta_i, \dots)$$

A cartesian function in the form  $f : \mathbb{N} \times A \longrightarrow B$  can then also be written as a sequence

$$(f_0, f_1, f_2, \dots, f_i, \dots)$$

where<sup>2</sup> the  $f_i$ s are cartesian functions  $A \longrightarrow B$ . While viewing functions that vary over numbers as sequences is helpful, and writing the number inputs as subscripts often simplifies notation, note that all this is just a matter of notational convenience. Formally,

- $(\beta_i : B)_{i=0}^{\infty}$  is the sequence notation for  $\beta : \mathbb{N} \longrightarrow B$ , and
- $(f_i : A \longrightarrow B)_{i=0}^{\infty}$  is the sequence notation for  $f : \mathbb{N} \times A \longrightarrow B$ .

## 4.3 Counting down: Induction and recursion

### 4.3.1 Computing by counting

Counting, induction, and recursion are schemas used to specify sequences of increasing generality:

- **counting** builds the sequence of numbers  $0, 1, 2, \dots : \mathbb{N}$  from  $\{\} : \mathbb{N}$  and  $s : \mathbb{N} \longrightarrow \mathbb{N}$  by

$$0 = \{\} \qquad n + 1 = s(n) \qquad (4.10)$$

- **induction** builds sequences of elements  $\phi_0, \phi_1, \phi_2, \dots : B$  from  $b : B$  and  $q : B \longrightarrow B$  by

$$\phi_0 = b \qquad \phi_{n+1} = q(\phi_n) \qquad (4.11)$$

- **recursion** builds sequences of functions  $f_0, f_1, f_2, \dots : A \longrightarrow B$  from  $g : A \longrightarrow B$  and  $h_0, h_1, h_2, \dots : B \times A \longrightarrow B$  by

$$f_0(x) = g(x) \qquad f_{n+1}(x) = h_n(f_n, x) \qquad (4.12)$$

Counting is a special case of induction, and induction is a special case of recursion. They are the methods for computing by counting. More precisely, they are the methods for computing by counting *down*, since the value of the  $(n + 1)$ -st entry of an inductively defined sequence is reduced to the value of  $n$ -th entry. In monoidal computers, this is captured by program evaluations within program evaluations.

<sup>2</sup>The notation is also used when  $f$  is cartesian only in the first argument, and  $f_i$ s are monoidal functions.

### 4.3.2 Induction

The **induction schema** for an arbitrary type  $B$  is

$$\frac{b : B \quad q : B \longrightarrow B}{\langle b, q \rangle : \mathbb{N} \longrightarrow B} \quad (4.13)$$

where the *banana-sequence*  $\langle b, q \rangle$  is defined

$$\langle b, q \rangle_0 = b \quad \langle b, q \rangle_{s(n)} = q(\langle b, q \rangle_n) \quad (4.14)$$



**Why bananas?** If the sequence defined in (4.14) is given a name  $\phi = \langle b, q \rangle$ , the list of its values becomes  $(\phi_0, \phi_1, \phi_2, \dots)$ . But programs usually do not use such listings, but only evaluate particular entries. So instead of reserving the name  $\phi$  only to be able to call  $\phi_{23}$ , the programmers invented the banana-notation, and call  $\langle b, q \rangle_{23}$ .

**Computing as counting down.** To evaluate an entry of an inductively defined sequence, we descend down the step-case part of (4.14) to the base-case part:

$$\langle b, q \rangle_n = \langle b, q \rangle_{s^n(0)} = q(\langle b, q \rangle_{s^{n-1}(0)}) = q^2(\langle b, q \rangle_{s^{n-2}(0)}) = \dots = q^n(\langle b, q \rangle_0) = q^n(b) \quad (4.15)$$

This descent is displayed in Fig. 4.4.

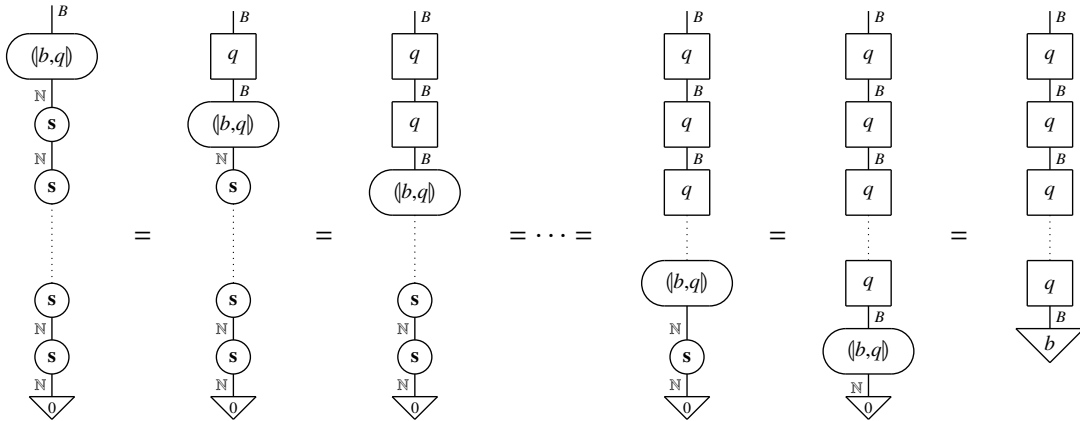


Figure 4.4:  $\langle b, q \rangle$  is computed by counting  $s$ -beads and replacing them with  $q$ -boxes

### 4.3.3 Reverse programming induction

The entries of the sequence  $\phi = \langle b, q \rangle : \mathbb{N} \longrightarrow B$  for any  $b : B$  and  $q : B \longrightarrow B$  can be computed by counting down by a program defined as a Kleene fixpoint of the cartesian function  $\tilde{\phi} : \mathbb{P} \times \mathbb{N} \longrightarrow B$  defined

$$\tilde{\phi}(p, n) = \text{ifte}(0^?(n), b, q \circ \{p\} \circ r(n)) \quad (4.16)$$

If  $\tilde{\Phi}$  is a Kleene fixpoint of  $\tilde{\phi}$ , then setting  $\phi = \{\tilde{\Phi}\}$  yields the sequence  $\phi : \mathbb{N} \longrightarrow B$  of

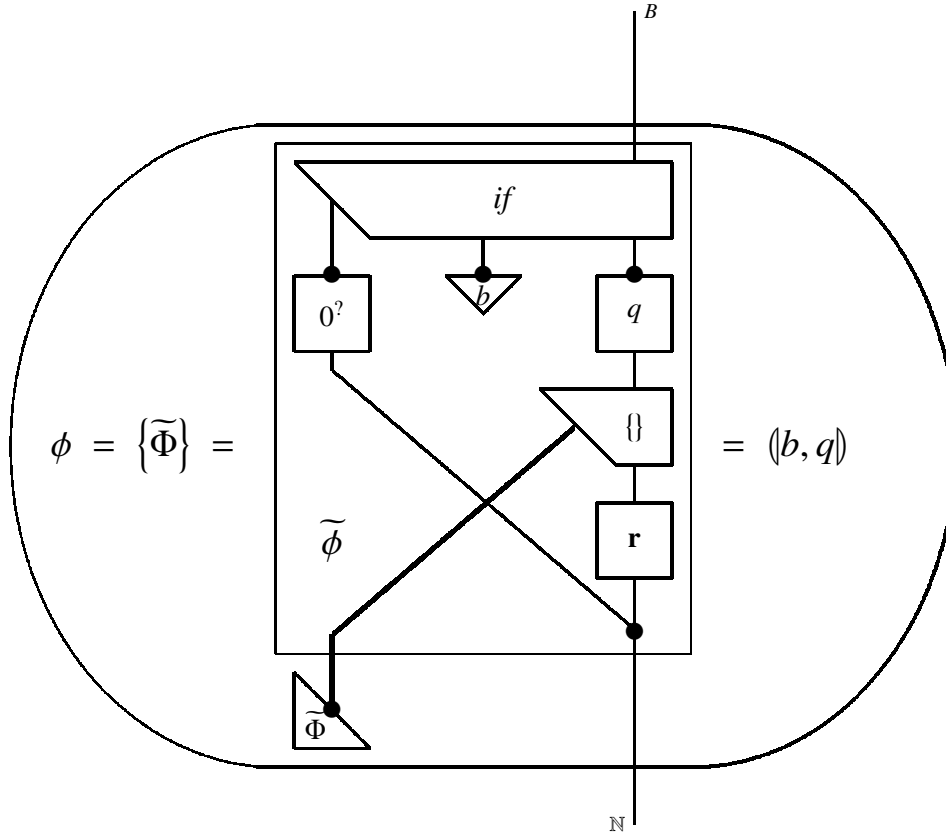


Figure 4.5: Induction in monoidal computer

$$\phi_n = \{\tilde{\Phi}\}_n = \tilde{\phi}(\tilde{\Phi}, n) = \text{ifte}(0^?(n), b, q(\phi_{r(n)})) \quad (4.17)$$

The construction is displayed in Fig. 4.5.

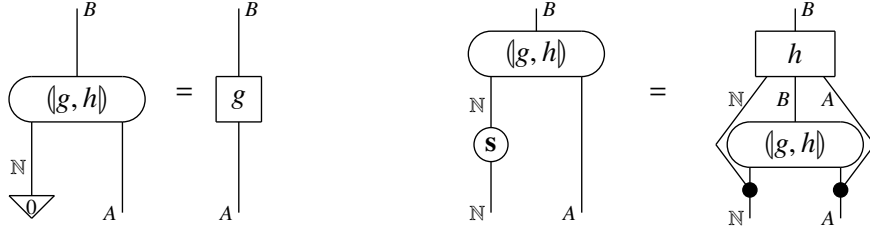
### 4.3.4 Recursion

The **recursion schema** for specifying sequences of functions from  $A$  to  $B$  is

$$\frac{g : A \longrightarrow B \quad h : \mathbb{N} \times B \times A \longrightarrow B}{\langle g, h \rangle : \mathbb{N} \times A \longrightarrow B} \quad (4.18)$$

The *banana function sequence*  $\langle\!\langle g, h \rangle\!\rangle$  is defined

$$\langle\!\langle g, h \rangle\!\rangle_0(x) = g(x) \qquad \langle\!\langle g, h \rangle\!\rangle_{s(n)}(x) = h_n(\langle\!\langle g, h \rangle\!\rangle_n(x), x) \quad (4.19)$$



**Evaluating recursive functions.** In the most general form, computing by counting backwards can get very inefficient. This is to some extent reflected in its algebraic expansion:

$$\begin{aligned} \langle\!\langle g, h \rangle\!\rangle_n(x) &= \langle\!\langle g, h \rangle\!\rangle_{s^n 0}(x) = h_{n-1}(\langle\!\langle g, h \rangle\!\rangle_{s^{n-1} 0}(x), x) = \\ &= h_{n-1}(h_{n-2}(\langle\!\langle g, h \rangle\!\rangle_{s^{n-2} 0}(x), x), x) = \cdots = h_{n-1}(h_{n-2}(\cdots h_1(\langle\!\langle g, h \rangle\!\rangle_{s 0}(x), x) \cdots x), x) = \\ &= h_{n-1}(h_{n-2}(\cdots h_1(h_0(\langle\!\langle g, h \rangle\!\rangle_0(x), x), x) \cdots x), x) = h_{n-1}(h_{n-2}(\cdots h_1(h_0(g(x), x), x) \cdots x), x) \end{aligned}$$

The diagrammatic view of the same countdown in Fig 4.6, however, shows that the recursive descent is still a simple counting process: The recursive step is expanded in Fig. 4.7, showing that the successor beads need to be copied as they are counted.

### 4.3.5 Running recursion

For any pair of functions  $g : A \rightarrow B$  and  $h : \mathbb{N} \times B \times A \rightarrow B$  in a computer, the function  $f = \langle\!\langle g, h \rangle\!\rangle : \mathbb{N} \times A \rightarrow B$  derived by recursion (4.18) is also computable. The program for  $f$  can be constructed as a Kleene fixpoint as follows. Consider the function  $\tilde{f} : \mathbb{P} \times \mathbb{N} \times A \rightarrow B$  defined by

$$\tilde{f}(p, n, x) = \text{ifte}(0^?(n), g(x), h_{\mathbf{r}(n)}(\{p\}(\mathbf{r}(n), x), x)) \quad (4.20)$$

If  $\tilde{F}$  is a Kleene fixpoint of  $\tilde{f}$ , then the definitions give

$$\{\tilde{F}\}(n, x) = \tilde{f}(\tilde{F}, n, x) = \text{ifte}(0^?(n), g(x), h_{\mathbf{r}(n)}(\{\tilde{F}\}(\mathbf{r}(n), x), x)) \quad (4.21)$$

Unfolding the *ifte*-branching gives

$$\{\tilde{F}\}(n, x) = \begin{cases} g(x) & \text{if } n = 0 \\ h_{\mathbf{r}(n)}(\{\tilde{F}\}(\mathbf{r}(n), x), x) & \text{if } n > 0 \end{cases} \quad (4.22)$$



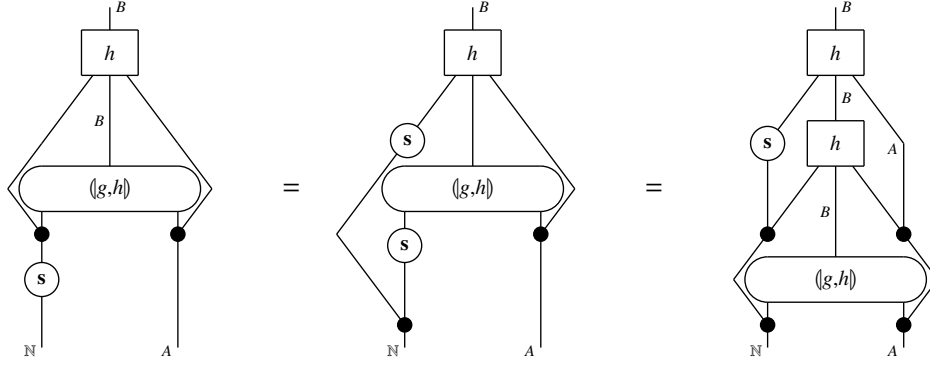


Figure 4.7: Intermediary step in Fig. 4.6: An  $s$ -bead is cloned before one copy is discarded

which means that  $\{\widetilde{F}\}$  satisfies (4.19), and we can set

$$\langle g, h \rangle_n(x) = \{\widetilde{F}\}(n, x) \quad (4.23)$$

The construction is summarized in Fig. 4.8.

**Evaluating a recursive function in a computer.** How is the function in Fig. 4.8 evaluated down to its lower values? Suppose the input  $\langle n, a \rangle : \mathbb{P} \times A$  enters the strings at the bottom. At the first step, both values are copied,  $n$  to  $0^?$  and  $\mathbf{r}$ , and  $a$  to  $g$  and  $h$ . Going up,  $\mathbf{r}$  is evaluated on  $n$ , and the result is copied to  $h$  and  $\{\widetilde{F}\}$ , as is  $a$ . The value  $y = \{\widetilde{F}\}(\mathbf{r}(n), a)$  is computed next and the output is fed to  $h$ . The computations  $0^?(n)$ ,  $g(x)$ , and  $h_{\mathbf{r}(n)}(y, a)$  are then performed in parallel. If  $n$  is a correctly formed number in the form  $\lceil \perp, \mathbf{r}(n) \rceil$ , then  $0^?(n)$  is  $\perp$  and *ifte* evaluated on  $\perp$  reduces to the second projection, and outputs the value  $h_{\mathbf{r}(n)}(y, a)$ , and the computation halts. The recursive step is the evaluation of  $y = \{\widetilde{F}\}(\mathbf{r}(n), a) = \langle g, h \rangle_{\mathbf{r}(n)}(a)$ , since  $\widetilde{F}$  is the Kleene fixpoint defining  $\langle g, h \rangle$ . The red boxes in Fig. 4.9 show how the computation of  $\langle g, h \rangle_{\mathbf{r}(n)}(a)$  is embedded within the computation of  $\langle g, h \rangle_n(a)$ : the big red box  $\langle g, h \rangle = \{\widetilde{F}\}$  on the right is the same function as the little red box  $\{\widetilde{F}\}$  contained in the diagram on the left. The big one is evaluated on  $n$  and the little one on  $\mathbf{r}(n) = n - 1$ . If we zoom into the little box, and open it up, it becomes the right-hand side of another diagram like Fig. 4.9, just a step further down the recursive ladder Fig. 4.6. On the left-hand side of that diagram there is another copy of  $\langle g, h \rangle = \{\widetilde{F}\}$  in a small box, this time evaluated on  $\mathbf{r}(\mathbf{r}(n)) = n - 2$ . The recursive ladder thus unfolds down, as the evaluation of each  $\langle g, h \rangle_m$ , for  $m = n, n - 1, n - 2, \dots, 1$  calls for the evaluation of the copy of  $\langle g, h \rangle$  on  $\mathbf{r}(m) = n - 1, n - 2, \dots, 0$ . At the last such step, the  $0^?$ -test evaluates to  $\top$ , and *ifte* outputs the first projection  $g(a) = \langle g, h \rangle_0(a)$ , which is returned to the function  $h_1(\langle g, h \rangle_0(a), a) = \langle g, h \rangle_1(a)$  that called it, and so on, up to  $\langle g, h \rangle_n(a)$ . This is the evaluation process depicted in Figures 4.6 and 4.7.

In summary, Fig. 4.9 shows how that recursive evaluation process can be realized using the self-reference of the Kleene fixpoints. In string diagrams, these self-calls are captured by the box that contains a copy of itself, that contains a copy of itself, and so on. In the case of recursion, the calls count down from  $\langle g, h \rangle_n$ , to  $\langle g, h \rangle_{n-1}, \langle g, h \rangle_{n-2}, \dots, \langle g, h \rangle_1, \langle g, h \rangle_0$ , and the descent is finite. In other program constructs, the evaluation through self-reference may not be finite.

The diagram illustrates a quantum circuit for a classical control loop. It features two input registers,  $N$  and  $A$ , and a classical control line  $B$ . The circuit is divided into two main sections by a dashed line. The left section, enclosed in a dashed box, contains a classical control loop. It starts with a  $0^?$  gate, followed by a  $g$  gate, and then a  $h$  gate. The output of the  $h$  gate is fed back into the  $g$  gate. A red dashed box highlights a specific part of the circuit, which includes a  $\{ \}$  gate and a  $\tilde{F}$  gate. The right section of the circuit, also enclosed in a dashed box, shows a quantum circuit with a control line  $B$  and a target line  $A$ . It includes a  $\{ \}$  gate and a  $\tilde{F}$  gate. The output of the circuit is a classical bit, which is fed back into the control line  $B$ .

77

## 4.4 Counting up: Search and loops

### 4.4.1 Search

While the recursion counts down, and reduces the value of  $\langle g, h \rangle_n(a)$  to  $\langle g, h \rangle_{n-1}(a)$ ,  $\langle g, h \rangle_{n-2}(a)$ , all the way to  $\langle g, h \rangle_0(a)$ , the search counts up  $0, 1, 2, \dots, n, \dots$ , computes the values of a given function  $\varphi_n(x)$ , and outputs an index  $n$  if a value satisfies some condition, usually an equation. A simple example of search is the minimization schema, searching for the smallest  $n : \mathbb{N}$  such that  $\varphi_n(x) = 0$  holds for a given sequence of functions  $\varphi : \mathbb{N} \times A \rightarrow \mathbb{N}$  and an input  $x : A$ . The schema is thus:

$$\frac{\varphi : \mathbb{N} \times A \rightarrow \mathbb{N}}{f = MU(\varphi) : A \rightarrow \mathbb{N}} \quad \text{where} \quad f(x) = \mu n. (\varphi_n(x) = 0) \quad (4.24)$$

In the monoidal computer, testing that  $\varphi_n(x) = 0$  is implemented as the predicate

$$0^? \varphi_n = \left( A \xrightarrow{\varphi_n} \mathbb{N} \xrightarrow{0^?} \mathbb{B} \right)$$

The requirement that  $n = f(a)$  is the smallest natural number such that  $\varphi_n(x) = 0$  means that  $\varphi_m(a) = 0$  implies  $n \leq m$  for all  $m$ , which using the previously implemented predicates becomes

$$(f(a) \stackrel{?}{=} n) = (0^? \varphi_n(a) \wedge \forall m. 0^? g_m(a) \Rightarrow 0^?(n \div m)) \quad (4.25)$$

To implement  $f = MU(g)$  in the abstract computer, we use an intermediary computation  $\tilde{f} : \mathbb{P} \times \mathbb{N} \times A \rightarrow \mathbb{N}$  again, this time defined

$$\tilde{f}(p, n, x) = \text{ifte}(0^? g_n(x), n, \{p\}(\mathbf{s}(n), x)) \quad (4.26)$$

as displayed in Fig. 4.10. If  $\tilde{F}$  is the Kleene fixed point of  $\tilde{f}$ , i.e.

$$\{\tilde{F}\}(n, x) = \text{ifte}(0^? g_n(x), n, \{\tilde{F}\}(\mathbf{s}(n), x)) \quad (4.27)$$

then set  $f = MU(g) = \{\tilde{F}\}0$ , i.e.

$$\begin{aligned} f(x) = \{\tilde{F}\}(0, x) &= \text{ifte}(0^? g_0(x), 0, \{\tilde{F}\}(1, x)) \text{ where} \\ \{\tilde{F}\}(1, x) &= \text{ifte}(0^? g_1(x), 1, \{\tilde{F}\}(2, x)) \text{ where} \\ \{\tilde{F}\}(2, x) &= \text{ifte}(0^? g_2(x), 2, \{\tilde{F}\}(3, x)) \dots \end{aligned}$$

**Running search in a picture.** The diagram in Fig. 4.10 performs search by calling its own a copy that it contains, just like the recursion did in Fig. 4.9. The self-call displayed Fig. 4.11 is analogous to the one in Fig. 4.9. The crucial difference is that  $\{\tilde{F}\}(n, x)$  in equation (4.27) and Fig. 4.11 calls itself on the successor  $\mathbf{s}(n) = n + 1$ , whereas in equation (4.22) and Fig. 4.9 it calls itself on the predecessor  $\mathbf{r}(n) = n - 1$ . While the recursive calls descend from  $n$ , and must terminate after  $n$  steps, the search calls ascend, and may diverge to the infinity if no  $n$  is found to satisfy  $g_n(a)$ . (Another difference between Figures 4.9 and 4.11 is that the box on the right is large in the former and small in the latter. This is not



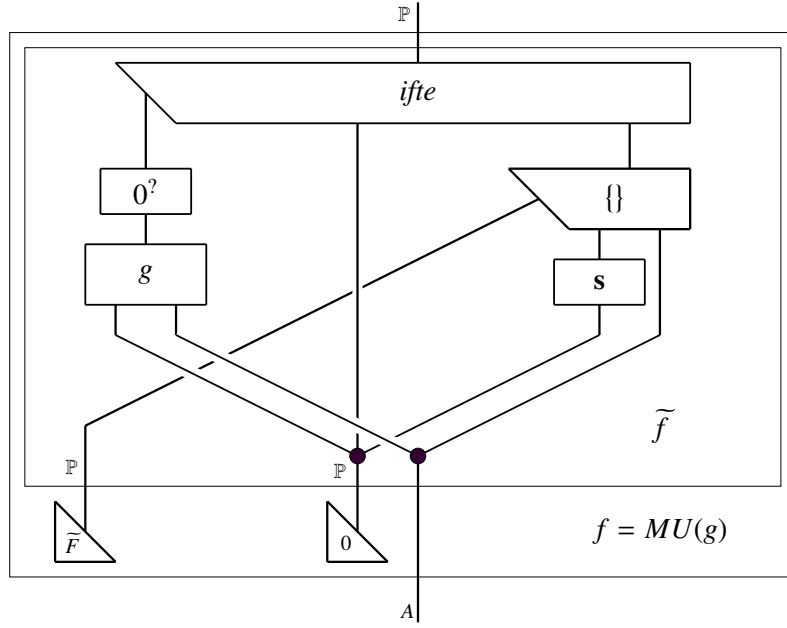


Figure 4.10: Unbounded search in monoidal computer

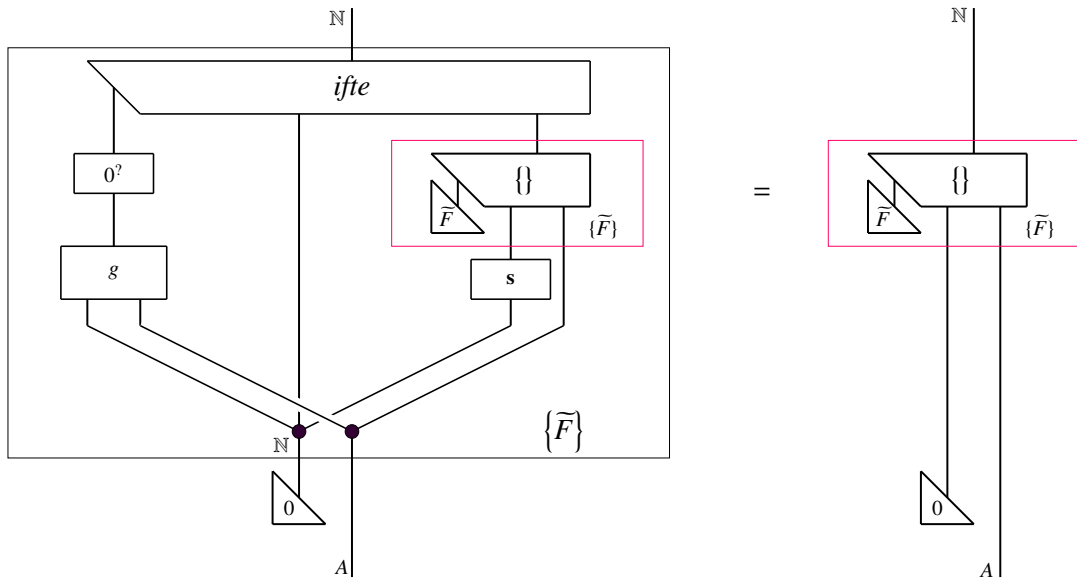


Figure 4.11:  $\{\tilde{F}\}(n, x)$  contains  $\{\tilde{F}\}(s(n), x)$

a real difference, but the two views show that  $\{\tilde{F}\}$  on the right corresponds to the entire computation on the left, and that it is the same function like the component on the left.)

## 4.4.2 Loops

In most programming languages, the unbounded search is usually expressed using the loop constructs. E.g. the *while* loop, written in pseudocode, is something like `while  $t(x, y)$  do  $h(x, y)$  od`, where  $t : A \times B \rightarrow \mathbb{P}$  is a predicate to be tested, and  $h : A \times B \rightarrow A$  is a function to be executed while the predicate  $t$  is holds. The derivation rule is thus

$$\frac{t : A \times B \rightarrow \mathbb{P} \quad h : A \times B \rightarrow A}{f = WH(t, h) : A \times B \rightarrow A}$$

Informally, the intended meaning in terms of the *while*-pseudocode

$$f(a, b) = (x := a; \text{while } t(x, b) \text{ do } x := h(x, b) \text{ od; print } x)$$

where `do` and `od` open and close a block of code, and  $x := a$  is an assignment statement. The `do`-block inside the `while`-loop assigns the output of the given function  $h$  to the variable  $x$ . The idea is that the predicate  $t$  is tested on the input values  $x = a$  and  $b$ , and if the test is satisfied, and returns the truth value  $\top$ , then  $h$  is applied, and it updates the value stored in  $x$ . At the next step, the new value  $x = h(x, b)$  and the old  $b$  are tested by  $t$ , and as long as  $t$  remains true, the function  $h$  is reapplied, and it keeps updating the value in  $x$ . If the test  $t$  at some point returns the truth value  $\perp$ , the function  $f$  outputs the current value of  $x$ , and the computation halts. One way to formalize the intended computation process is to:

- define a sequence  $a_0, a_1, a_2, \dots$  in  $A$  inductively, setting  $a_0 = a$  and  $a_{i+1} = h(a_i, b)$ ;
- output  $f(a, b) = a_k$  for the least  $k$  such that  $\neg t(a_k, b)$ .

**Note** that the function  $h$  keeps updating the values of the variable  $x : A$ , but that the value  $b : B$  is a parameter, on which both  $t(a, b)$  and  $h(a, b)$  depend, but the loop does not change it.

To implement a while-loop in an abstract computer, we write an intermediary function  $\tilde{f} : \mathbb{P} \times A \times B \rightarrow A$  again, and use its Kleene fixpoint to as a program for  $f = WH(t, h)$ . The intermediary function is this time:

$$\tilde{f}(p, a, b) = \text{ifte}(t(a, b), \{p\}(h(a, b), b), a) \quad (4.28)$$

The Kleene fixed point  $\tilde{F}$  of  $\tilde{f}$  now satisfies

$$\{\tilde{F}\}(a, b) = \text{ifte}(t(a, b), \{F\}(h(a, b), b), a) \quad (4.29)$$



4) rabbits never die.

Draw a string program for computing the total number of pairs of rabbits after  $n$  months.

**Background.** This early example of induction is due to Leonardo Pisano Bigollo (~1175–1250), the "Traveller from Pisa", who actually grew up and studied in North Africa, where his father *Bonacci* held a Pisan trade outpost. Bonacci's son *Fibonacci* travelled back to Pisa and brought with him not only the rabbit-counting function, but also the positional numeral system in use today. In Europe at the time, even arithmetic was still done in Latin. Fibonacci noted even in the title of some of his writings that his methods originated from India.

#### 4.5.2 Recursion work

a. Draw string programs for the following functions:

i)  $m + n$

ii)  $m \times n$

iii)  $m \div n = \begin{cases} m - n & m > n \\ 0 & \text{otherwise} \end{cases}$

iv)  $\Sigma(n) = 0 + 1 + 2 + \dots + n$

b. The recursion schema from Sec. 4.3.4 requires that both  $g$  and  $h$  are evaluated each time when  $\langle g, h \rangle$  is evaluated. This is wasteful, since  $g$  only needs to be evaluated in the base-case  $n = 0$ ,  $h$  only in the step-cases  $n = m + 1$ . A similar issue arose with respect to the gives rise to the same issue like the the *ifte*-branching from Sec. 2.4.1. Given some programs  $G$  and  $H$  for the computations  $g$  and  $h$ , implement a lazy recursion schema, where  $g$  and  $h$  are only evaluated when needed.

#### 4.5.3 Search work

a. Draw a string program for discrete cubic root search  $\sqrt[3]{n} = \begin{cases} m & n = m^3 \\ \uparrow & \text{otherwise} \end{cases}$

b. Implement in monoidal computer the following program schemas:

i) bounded search:

$$\frac{k \in \mathbb{N} \quad g : \mathbb{N} \times A \longrightarrow \mathbb{N}}{f = MU(k, g) : A \longrightarrow \mathbb{N}}$$

where  $f = MU(k, g)$  means that  $f(x) = \mu n \leq k. 0^?g(n, x)$  is the smallest number  $n \leq k$  where  $g(n, x) = 0$ , i.e.

$$f(x) = n \iff 0^?g(n, x) \wedge g \leq k \wedge \forall m \leq k. 0^?g(m, x) \Rightarrow 0^?(n \div m)$$

Prove that bounded search always terminates.

ii) *FOR*-loop:

$$\frac{k \in \mathbb{N} \quad h : A \times B \longrightarrow A}{f = \text{FOR}(k, h) : A \times B \longrightarrow A}$$

where  $f_k = \text{FOR}(k, h)$  is defined on  $x : A$  and  $y : B$  inductively, as the sequence

$$f_0 = x \quad \text{and} \quad \vec{f}_{i+1} = h(f_i, y)$$

## 4.6 Stories

### 4.6.1 Church's reverse programming

In this chapter, we built programs by composing copies of the program evaluator, i.e. by applying the run-instruction. The arithmetic operations and the program constructors have been reverse-programmed (in the sense of Sec. 4.1) in the single-instruction language Run. The seminal example of reverse programming was Alonzo Church's reconstruction of the arithmetic and the logical operations in his  $\lambda$ -calculus [30, Ch. III]. In the present context, the  $\lambda$ -calculus can be viewed as a two-instruction programming language. The first instruction is a version of the run-instruction, the *function application* operation, which applies a function to its inputs. It is usually abbreviated to concatenation, writing  $\text{apply}(f)(x) = fx$ . The second instruction is the *function abstraction* operation, written  $\text{abstract}(x)(f) = \lambda x.f$ .<sup>3</sup> The function abstraction is different from the program abstraction in that for every computable function  $f$  there are many programs  $F$  such that  $\{F\}a = f(a)$ , whereas the abstraction picks  $\lambda x.f$  such that  $(\lambda x.f)a = f(a)$ . But the function abstraction is similar to the program abstraction in that it is an inverse to the apply-instruction, as displayed in Fig. 4.13, just like the program abstraction is an inverse to the run-instruction, as displayed in Figures 2 and 3. While a programmer tries to abstract

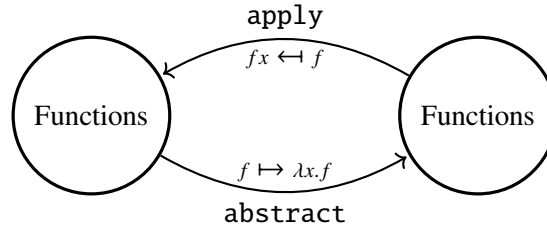


Figure 4.13: The operations of function abstraction and application

from a given function  $f$  a program  $F$  such that  $\text{run}(F) = f$ , the  $\lambda$ -abstraction operation abstracts from  $f$  the function abstraction  $\text{abstract}(f)$  such that  $\text{apply}(\text{abstract}(f)) = f$ . This requirement that  $\text{abstract}$  is a right inverse to  $\text{apply}$ , written in the abbreviated form  $(\lambda x.f)a = fa$ , is called the  $\beta$ -conversion rule in the  $\lambda$ -calculus. The requirement that  $\text{abstract}$  is also a left inverse to  $\text{apply}$ , i.e.  $\text{abstract}(\text{apply}(f)) = f$ , written in the abbreviated form  $\lambda x.(fx) = f$ , is called the  $\eta$ -conversion rule. While the  $\beta$ -conversion requirement is always imposed, the  $\lambda$ -calculus satisfying the  $\eta$ -conversion requirement is called *extensional*. In the extensional  $\lambda$ -calculi, the operations  $\text{abstract}$  and  $\text{apply}$  form a bijection between the functions and their abstractions.

<sup>3</sup>The letter  $\lambda$  was, according to the oral lore, chosen by the typesetters of Alonzo Church's early publications. They apparently had trouble typesetting  $\hat{x}$ , used in Russell-Whitehead's *Principia* [141], so they first modified it to  $\hat{x}$ , and then proposed  $\lambda x$ , which was accepted.

The  $\lambda$ -calculus was the outcome of Church's effort to provide an abstract account of Gödel's encodings [58] of functions as numbers [30, Ch. IV]. While Gödel's encodings can be viewed as an example of abstracting programs from functions, Church leveled the field by making functions into first-class citizens (indeed, the *only* citizens) of his theory, and used the function abstraction operation (now distinguished from the program abstraction, which is not an operation but a process) to provide an algebraic account of the process of computation. He provided a roadmap for reverse programming and demonstrated that the basic instructions `abstract` and `apply` suffice to compute everything that can be computed. The point of the present chapter is that the lone cousin of the `apply`-instruction, the `run`-instruction, suffices on its own. The role of Church's function abstraction operation is played by the assumption in (2.3) that a computable function always has a program abstraction, which boils down to assuming that the `run`-operation is surjective. The upshot is that the `run`-operation alone also suffices for defining the notions of computability and "effective procedure", which was the original purpose of it all.

#### 4.6.2 Story of curly brackets

The relationship between the function abstraction and the program abstraction is subtle and convoluted. Both emerged from attempts to capture the idea behind Gödel's encoding arithmetic in itself; to formalize his abstractions of the arithmetic functions, formulas, and statements into numbers, available as for processing by arithmetic functions, formulas, and statements. The function abstraction provided an abstract view, the operation mapping a function  $f$  to its abstraction  $\lambda x.f$ . The program abstraction provided a concrete view, where a programmer specifies a program  $F$  for  $f$ . The function is recovered as  $f(a) = (\lambda x.f)a$ , by applying its abstraction, or as  $f(a) = \{F\}a$ , by evaluating its program. In [88], Kleene wrote the function application in the form  $f(a) = \{\lambda x.f\}a$ . In [169], Turing adopted the same notation to describe the evaluation  $\{M\}a = f(a)$  of a machine  $M$  constructed for any  $\lambda$ -definable  $f$ . Kleene then adopted the same notation for the actual program evaluators of the primordial programming language of Gödel encodings of recursive functions [91, §66]. The curly brackets as general program evaluators remain with us as a reminder of the common origin of running and the application, and of programming and the abstraction.

#### 4.6.3 The Church-Turing Thesis

Theory of computation was born of Hilbert's Program to develop mathematical logic as a metamathematical foundation. The original goal of this effort was to find the "*effective procedures*" to decide whether scientific theories are true or false. The first task was to formally define the notion of an effective procedure. The concept of computability emerged from that task. The most significant first step was made in Gödel's proof of his Incompleteness Theorem [58]. It finished off Hilbert's Program to develop effective procedures for deciding everything, but provided a stepping stone towards defining what is an effective procedure. At the heart of Gödel's proof was the recursion schema. Informal recurrences had been in use for a long time, and Dedekind had formalized the recursion schema decades earlier [42]. Gödel used it as a programming tool in his proof, and uncovered its capability to encode itself, which was the crucial step towards the concept of programming. One could argue that programs existed since the early XIX century as the patterns fed to Jacquard's looms, and even more compellingly as Ada Lovelace's formalization of Bernoulli's algorithm, which was recursive. Most histories of computation provide a full account. Gödel's insight [58] that recursively defined arithmetic functions can be encoded as numbers and processed by recursively defined arithmetic functions, encoding some arithmetic proofs, brought the idea of programming to another level. Or to infinitely many levels of such

self-applications [73]. While the recursion schema was clearly an effective procedure sufficient for the purposes of Gödel's paper, he stated clearly, explicitly, and repeatedly that the recursion schema alone did not provide a complete characterization of all effective procedures. Simple examples of arithmetic functions that are effective but not recursive can be found in Sec. 6.3.

Alonzo Church converged towards his thesis iteratively, through his quest for characterization of computable ("effectively calculable") functions [27]. He developed his calculus of  $\lambda$ -abstraction upon Bertrand Russell's type theory [140], avoiding the non-constructive (and thus uncomputable) aspects of the set-theoretic foundations. The operation of  $\lambda$ -abstraction extracts a description from a function, and can be construed as an abstract form of Gödel's encoding of arithmetic functions as numbers [58]. This encoding is the central part of Gödel's groundbreaking incompleteness theorems, and the origin of the *programs-are-data* paradigm. But Gödel rejected Church's proposals to define computability in terms of the function abstraction operation outright [161, Ch. 17]. Church updated his proposal to include Gödel's extension of the recursion schema along the lines proposed by Herbrand, and eventually arrived at the postulate that not only his type-theoretic model, and Gödel's recursion-theoretic model, but also all other models of computation proposed at the time or in the future must compute the same family of functions [161, Sec. 17.2]. This was Church's Thesis. Gödel remained unswayed, continuing to object that the Gödel-Herbrand schemas were indeed effective, but that there were no grounds for claiming that all procedures that may come to be considered effective will be reducible to this or that combination of schemas. Then Turing's model appeared, and provided the grounds. Gödel not only accepted that "the correct definition of mechanical computability was established beyond any doubt by Turing", but added that Turing has "for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen" [59]. Although Gödel was the undisputed arbiter of the matter, Turing was at the time a graduate student, Church was his advisor, and the community did not rush to recognize Turing's breakthrough. The other models were proved to provide correct definitions of computability by demonstrating their equivalence with Turing's. It was first proved that the Gödel-Herbrand recursive functions coincide with the *total* functions computable by the Turing machines. Then Stephen Kleene, another student of Church's, realized that adding the minimization schema from Sec. 4.4.1 to the recursion schema from Sec. 4.3.4 allows deriving the same family of computable functions like Turing's machines [86, 87]. As the time went by, Emil Post's canonical word-processing systems, Jim Lambek's abacus, Zuse's cellular automata (with Conway's Game of Life as a special case), and an entire bestiary of other machines, rewrite systems, and circuit ensembles (uniformly generated by Turing machines!) were all proved to make the same functions computable, lending credibility to Church's Thesis. The multitude of different ways to compute the same class of functions was hailed as evidence of its robustness [91, §62].

Although Kleene's combination of recursion and minimization does not look different from other equivalent models, it came to dominate the theory of computation, to the point that Turing's concept of *computable* function got overridden by the term *partial recursive* function, and the theory of computation itself came to be called the *theory of recursion* [161, Sec. 17.2]. To fit the bill, the recursion schema used by Gödel in his epochal [58] got renamed to "*primitive*" recursion, leaving the name "*recursive functions*" for the functions defined in Kleene's system that happen to be total [87]. It took time to remember that the partial recursive functions were just the computable functions renamed by a particular implementation [160, 161]. For his part, Gödel seems to have rejected the conundrum [161, Sec. 17.7.1] and in any case continued to use the term recursion for the schema in (4.19), as introduced into logic by him in [58] and developed by Rozsa Peter in [131], whom he referenced in this context. We revert to Gödel's terminology. Although Peter's work initially met wide recognition (a presentation at the International Congress of Mathematicians in 1932, an invitation to the founding editorial board of the Journal

of Symbolic Logic, all at time when women were still largely barred from holding academic positions), the upsurge of antisemitism and the ensuing iron curtain put an end to her research, leaving it with the "primitive" moniker.



Figure 4.14: Alonzo Church



Figure 4.15: Peter Rozsa

It should be noted that Turing did not propose a thesis or postulate, but proved that his model satisfied the requirements of a definition of "effective procedure" arising from Gödel's proofs [54]. There is no evidence that the idea of balancing his answer against his thesis advisor's Thesis ever crossed Turing's mind. That balancing act was a part of Kleene's project of reducing theory of computation to theory of recursion [91, §60, Thesis I], which he justified by reinterpreting Church's proposal to Gödel [ibidem, §62]. At the time when Kleene's was working on his monumental *Summa*, Turing was zeroing on computation as life [168] and approaching the end of his life. He broached the boundaries of epistemology by providing a mathematical model of how we compute. His theory of computation can be interpreted as a solution of the mind-body problem. The driving forces behind his work may have included an early death of a friend and a world war. But he never spoke up, leaving it to the murmurings of computers to spell out his message.

#### 4.6.4 Turing completeness

The multitude of models used in theory of computation is echoed by the multitude of the programming languages used in practice. Some are more convenient for some application domains, some for other, and some are not domain-specific but general-purpose and programmers' choices are often based of taste and habit. The diversity of programming languages is a product of the same divergent evolution as the diversity of natural languages. However, while the common foundations of natural languages are the ongoing studies in epistemology and paleolinguistics [26], the common foundations of the programming practices are the mathematical proofs, sometimes offered as the empiric evidence for the Church-Turing Thesis, that any given pair of proposed models of computation describe the same computable functions, and that any pair of standard programming languages are as expressive as each other, i.e. that they also describe the same family of computable functions. The crucial requirement for this is obviously (2.2), i.e. that they allow programming their own program evaluators. *The programming languages that allow programming all computable functions, including their own program evaluators, are called **Turing-complete**.* A majority of the programming languages encountered in practice are Turing-complete.<sup>4</sup>

<sup>4</sup>The only exceptions are some special-purpose languages, like early SQL, or the Berkeley Packet Filter, which restrict their programs regular expressions. In other words, they are as expressive as finite-state machines, rather than the Turing machines. The markup languages like XML and HTML are sometimes also mentioned as exceptions, although they are



---

not programming languages so it is not clear what the notion of Turing-completeness would mean for them.

20221212

## 5 What cannot be computed

---

5.1	Decidable extensional properties . . . . .	90
5.2	Gödel, Tarski: Provability and truth are undecidable . . . . .	92
5.3	Turing: Halting is undecidable . . . . .	93
5.4	Rice: Decidable extensional predicates are constant . . . . .	94
5.5	Workout . . . . .	96
5.6	Stories . . . . .	96
5.6.1	I cannot decide whether I lie . . . . .	96
5.6.2	The Church-Turing Antithesis . . . . .	97

---

## 5.1 Decidable extensional properties

**Idea.** Consider the following sets:

- $\{n \in \mathbb{N} \mid \exists m \in \mathbb{N}. m + m = n\},$
- $\{n \in \mathbb{N} \mid \exists m \in \mathbb{N}. 2m = n\},$
- $\{n \in \mathbb{N} \mid \frac{n}{2} \in \mathbb{N}\},$
- $\{n \in \mathbb{N} \mid \frac{n+1}{2} \notin \mathbb{N}\}.$

Since the four predicates that define these sets are satisfied by the same elements, the even numbers, the set-theoretic principle of *extensionality* says that the four sets are equal, i.e. that they are the same set. In the language of arithmetic, there are infinitely many predicates that describe this set. Ditto for other sets. Different predicates provide different *intensional* descriptions of the same *extensional* objects, the sets.

In computation, different programs provide different *intensional* descriptions of the same *extensional* objects, the computable functions. Different programs for the same computable function  $d : \mathbb{N} \rightarrow \mathbb{N}$  can be constructed from different basic or previously programmed operations:

- $d_0(m) = m + m,$
- $d_1(m) = 2 \times m,$
- $d_2(m) = \mu n. \left(\frac{n}{2} = m\right),$
- $d_3(m) = \mu n. \left(\frac{n+1}{2} > m\right),$

and in infinitely many other ways. The operation  $\mu n. \Phi(n)$ , discussed in Sec. 4.4.1, outputs the smallest  $n$  satisfying  $\Phi(n)$ . While different programming languages assign different sets of intensional descriptions to each function extension, the general view of it all, provided in the monoidal computer, is quite simple: the intensional descriptions are the elements of  $C^\bullet(X, \mathbb{P})$ , the function extensions are the elements of  $C(X \times A, B)$ , and the  $X$ -indexed instruction  $\text{run}_X : C^\bullet(X, \mathbb{P}) \rightarrow C(X \times A, B)$  from Sec. 2.5.3, represented by  $\text{run}_{\mathbb{P}}(\text{id}) = \{\}_A^B$  collapses the intensional descriptions  $G : X \multimap \mathbb{P}$  to the corresponding function extensions  $\text{run}_X(G) = \{G\}_A^B$ .

**Extensional equivalence** for types  $A, B$  is the binary relation  $\left(\overset{B}{\underset{A}{\equiv}}\right)$  on programs induced by running them as functions from  $A$  to  $B$ . The formal definition is

$$F \overset{B}{\underset{A}{\equiv}} G \iff \{F\}_A^B = \{G\}_A^B \quad (5.1)$$

The Fundamental Theorem, as displayed in Fig. 3.7, says that any program transformer  $\gamma : \mathbb{P} \multimap \mathbb{P}$  has an extensional fixpoint  $\Gamma \overset{B}{\underset{A}{\equiv}} \gamma(\Gamma)$ . The  $\Upsilon$ -classifiers in Sec. 3.4.2 could all be specified as extensional fixpoints, without the curly brackets. The systems of equations used to specify software systems in Sec. 3.5.2 can be formulated as extensional equivalence constraints. In general, since every computable

function has corresponds to a unique equivalence class of programs modulo  $\equiv_A^B$ , there is a bijection

$$C(A, B) \cong C^\bullet(I, \mathbb{P}) \Big/ \equiv_A^B \quad (5.2)$$

This gives rise to the question:

*What can a computer tell about a computable function from its programs without running them?* (5.3)

This chapter presents answers to some important special cases of this question, and a special answer to the general question. Ch. 9 returns for more general answers.

**Properties of functions.** A property  $\mathfrak{Q}$  of functions  $A \rightarrow B$  is presented as a predicate over the hom-set  $C(A, B)$ . The predicate returns  $\mathfrak{Q}(f) = \top$  when  $f$  has the property  $\mathfrak{Q}$  and  $\mathfrak{Q}(f) = \perp$  when it does not. We write  $\mathfrak{Q}(f)$  when  $\mathfrak{Q}(f) = \top$  and  $\neg\mathfrak{Q}(f)$  when  $\mathfrak{Q}(f) = \perp$ .

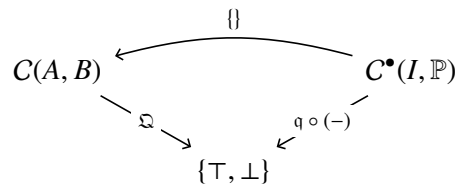


Figure 5.1: When can a property  $\mathfrak{Q}$  of computations be recognized as a property  $q$  of programs?

**Computable properties of computable functions.** A property of computable functions may be computable if it can be expressed as a computable predicate over their programs, as displayed in Fig. 5.1. Recall from Sec. 2.4.5 that a computable predicate over programs is presented as a computable function  $q \in C(\mathbb{P}, \mathbb{B})$ . We abbreviate  $q \circ F = \top$  to  $q(F)$  and  $q \circ F = \perp$  to  $\neg q(F)$  again.

**Decidable properties of computable functions.** While Fig. 5.1 provides an effective setting for computing with the properties of  $f = \{F\}$  in terms of  $F$ , it leaves open the possibility that a computation of the truth value  $q(F)$  may diverge and not tell anything. To assure that it does tell something, the extensional predicates are required to be decidable, i.e. presented a cartesian functions  $q \in C^\bullet(\mathbb{P}, \mathbb{B})$ . Question (5.3) becomes:

*When can a property of computable functions  $\mathfrak{Q}$  be decided on its programs by a predicate  $q: \mathbb{P} \rightarrow \mathbb{B}$  so that the encodings  $f = \{F\}$  give*

$$\mathfrak{Q}(f) \iff q(F) \quad (5.4)$$

**Extensional predicates.** If there are programs  $F, G$  which encode the same function  $\{F\} = \{G\}$ , but  $q(F) \neq q(G)$ , then (5.4) is obviously impossible. A predicate  $q$  over programs corresponds to a property  $\mathfrak{Q}$  of functions only if

$$F \equiv_A^B G \implies q(F) = q(G) \quad (5.5)$$

The program predicates  $q$  that satisfy this requirement are *extensional* for  $A, B$ . They are the congruences

with respect to the extensionality relation  $(\equiv_A^B)$ . A predicate may be extensional with respect to one pair of types  $A, B$  and not with respect to another.

## 5.2 Gödel, Tarski: Provability and truth are undecidable

**Decision predicates** are computable functions  $\mathfrak{d}: \mathbb{P} \times \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{B}$  which determine the truth values of all predicates, i.e. satisfy

$$\mathfrak{d}(Q, A, x) = \{Q\}_A^{\mathbb{B}} x \quad (5.6)$$

for all  $Q: \mathbb{P}$ ,  $A$ , and  $x: A$ . As established in Sec. 2.3.1, the types  $A$  can be viewed presented by programs  $A: \mathbb{P}$  which induce idempotents  $\{A\}: \mathbb{P} \rightarrow \mathbb{P}$  such that  $x: \mathbb{P}$  a means  $x = \{A\} x$ .

The **question** is whether a decision predicate can be decidable. Can the program evaluation be cartesian on predicates?

**Decision predicates are undecidable**, unless  $\top = \perp$ . Suppose that there is a decidable decision predicate  $d: \mathbb{P} \times \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{B}$ , and let  $D: \mathbb{P}$  be its program, satisfying  $\mathfrak{d}(Q, A, x) = \{D\}_A^{\mathbb{B}}(Q, x)$  for all  $Q, A, x$ . Instantiating to  $Q = D$  yields  $\mathfrak{d}(D, \mathbb{P}, x) = \{D\}_{\mathbb{P}}^{\mathbb{B}}(D, x)$ . Intuitively, this instance of the decision predicate says: "*I am true*". But the Fundamental Theorem provides also provides Kleene fixpoint  $\Psi$  of the *negation* of the decision predicate:

$$\neg \mathfrak{d}(\Psi, \mathbb{P}, x) = \{\Psi\}_{\mathbb{P}}^{\mathbb{B}} x \quad (5.7)$$

The negation flips the truth values, so that

$$\neg q(x) = \begin{cases} \perp & \text{if } q(x) = \top \\ \top & \text{if } q(x) = \perp \end{cases} \quad (5.8)$$

for any predicate  $q: A \rightarrow \mathbb{B}$ . Intuitively, the predicate  $\{\Psi\}_{\mathbb{P}}^{\mathbb{B}} x$  says "*I am false*." A self-referential statement that says that it is false causes the *Liar Paradox*<sup>1</sup>. But if  $d$  is a decision predicate (5.6) and if  $\Psi$  is a program for its negation (5.7), then

$$\neg \mathfrak{d}(\Psi, \mathbb{P}, x) \stackrel{(5.7)}{=} \{\Psi\}_{\mathbb{P}}^{\mathbb{B}} x \stackrel{(5.6)}{=} \mathfrak{d}(\Psi, \mathbb{P}, x) \quad (5.9)$$

It follows that  $\top = \perp$ , since  $\neg \top = \perp$  and  $\neg \perp = \top$ , and the assumption that  $\mathfrak{d}(\Psi, \mathbb{P}, x): \mathbb{P}$  means that it must be either  $\top$  or  $\perp$ . If  $\top \neq \perp$ , then  $\mathfrak{d}(\Psi, \mathbb{P}, x)$  cannot output a value in  $\mathbb{B}$ , and is not decidable.

**Remark.** Gödel used the above instance of diagonal argument for the purposes of his analysis of decidability of *provability* of propositions in a formal theory of arithmetic in [58]. Tarski applied similar reasoning in [164] where he analyzed decidability of *validity* of propositions in a given model. Both thus applied the diagonal argument to derive contradiction and prove negative results. Although the two

<sup>1</sup>The liar paradox is often called the *Epimenides' paradox*, because Epimenides, a poet and philosopher from Creta, who lived in VI or VII century BC, had written that "*All Cretans are liars*".

results refer to different logical questions, and the original versions involve different technical details, both are based on the same logical paradox. Therefore, in the rudimentary predicate logic implemented in monoidal computer so far, the two results are isomorphic. It should be emphasized, though, that the categorical approach accommodates structural and conceptual refinements as a matter of principle. A reader interested in conceptual distinctions might enjoy testing that principle.

### 5.3 Turing: Halting is undecidable

The main result presented in Turing's breakthrough paper [169], where he introduced his machines, was that there is no machine that would input descriptions of arbitrary machines, and output a decision whether they halt or diverge. Turing presented the undecidability of this *halting problem* as the negative solution of Hilbert's *Decision Problem* (*Entscheidungsproblem*).

**Function halting.** For the moment, we reduce halting to totality, and say that a function  $f : A \rightarrow B$  halts on a total input  $a : A$  if it produces a total output  $f(a) : B$ . Recall from Sec. 1.4 that  $x : X$  is total if

$$\left( I \xrightarrow{x} X \xrightarrow{\bullet} I \right) = \uparrow = \text{id}_I$$

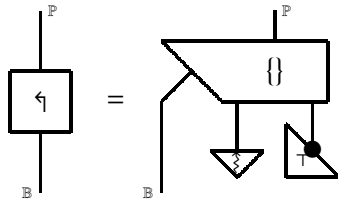
As mentioned in Sec. 2.4.5, the **halting notation**

$$f(a) \downarrow \quad \text{abbreviates} \quad \left( I \xrightarrow{a} A \xrightarrow{f} B \xrightarrow{\bullet} I \right) = \uparrow \quad (5.10)$$

**Halting predicates** are computable functions  $\mathfrak{h} : \mathbb{P} \times \mathbb{P} \times \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{B}$  that input programs  $F : \mathbb{P}$ , types  $A, B$ , and inputs  $x : A$ , and decide whether the evaluation of  $F$  on  $x : A$  halts or not:

$$\mathfrak{h}(F, A, B, x) = \begin{cases} \top & \text{if } \{F\}_A^B x \downarrow \\ \perp & \text{otherwise} \end{cases} \quad (5.11)$$

**Halting predicates are undecidable**, unless  $\top = \perp$ . Towards a contradiction, assume that there is a decidable halting predicate, and apply it to its own negation, like in Gödel's and Tarski's constructions in the preceding section. This time, though, we need a different negation: it should not just swap the truth values, but it should send a true output of a predicate to a divergent computation. That is what Turing used in his proof. This *divergent* negation  $\mathfrak{q}$  is defined as follows:



which gives

$$\mathfrak{q}q(x) = \begin{cases} \hat{\phantom{x}} & \text{if } q(x) = \top \\ \top & \text{if } q(x) = \perp \end{cases} \quad (5.12)$$

The function  $\mathfrak{q}q : A \rightarrow \mathbb{P}$  thus produces an output precisely when  $q : A \rightarrow \mathbb{B}$  is false:

$$\mathfrak{q}q(x) \downarrow \iff \neg q(x) \quad (5.13)$$

Let  $\Psi$  be a Kleene fixpoint of  $\mathfrak{h}$ , i.e.

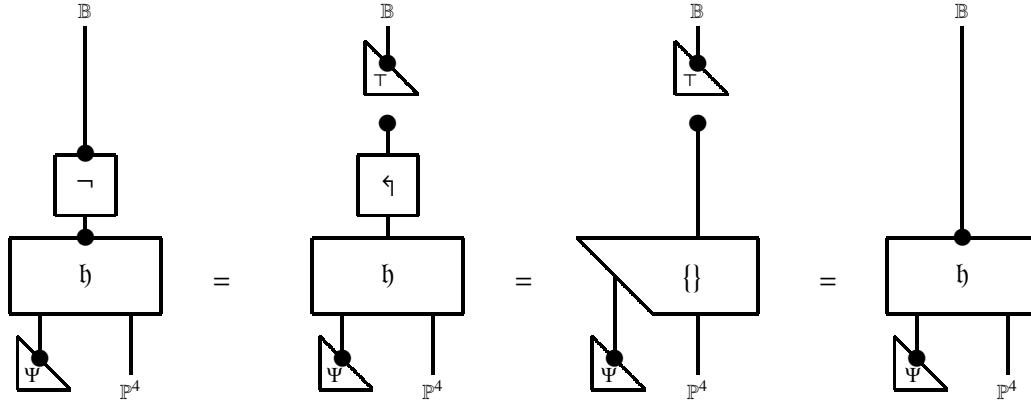
$$\mathfrak{h}(\Psi, \mathbb{P}^4, \mathbb{B}, x) = \{\Psi\}_{\mathbb{P}^4}^{\mathbb{B}} x \quad (5.14)$$

By (5.11), the halting predicate also satisfies

$$\mathfrak{h}(\Psi, \mathbb{P}^4, \mathbb{B}, x) \iff \{\Psi\}_{\mathbb{P}^4}^{\mathbb{B}} x \downarrow \quad (5.15)$$

Hence

$$\neg \mathfrak{h}(\Psi, \mathbb{P}^4, \mathbb{B}, x) \stackrel{(5.13)}{\iff} \mathfrak{h}(\Psi, \mathbb{P}^4, \mathbb{B}, x) \downarrow \stackrel{(5.14)}{\iff} \{\Psi\} x \downarrow \stackrel{(5.15)}{\iff} \mathfrak{h}(\Psi, \mathbb{P}^4, \mathbb{B}, x)$$



Since  $\neg$  swaps  $\top$  and  $\perp$ , the assumption that  $\mathfrak{h}(\Psi, \mathbb{P}^4, \mathbb{B}, x): \mathbb{B}$  implies  $\top = \perp$  again, and  $\top \neq \perp$  implies that  $\mathfrak{h}$  is not decidable.

## 5.4 Rice: Decidable extensional predicates are constant

Last but not least, we confront the original question: *What can we tell about a computation without running it?* For easy programs, we can often tell what they do. For convoluted programs, we cannot, and we tend to blame the programmers. Sometimes they are guilty as accused. In general, they are not, as there are no general, decidable predicates that tell anything at all about the computations just by looking at the programs. To add insult to the injury, asking such questions first leads to predicates over programs that are easy to program but may not terminate, and remain *undecidable*. Later it leads to predicates that are easy to define but impossible to program and remain *uncomputable*.

**Constant functions**  $f : A \longrightarrow B$  produce the same output  $b = f(x)$  for all  $x : A$ . For monoidal functions, this means that for all  $x, x' : X \longrightarrow A$  holds  $f \circ x = f \circ x'$ . Restricting to cartesian functions, it can be shown that  $f : A \longrightarrow B$  is constant if and only if

$$f = \left( A \xrightarrow{\top} I \xrightarrow{b} B \right)$$

for some  $b : B$ . Since  $\top$  and  $\perp$  as the only cartesian elements of  $\mathbb{B}$ , there are precisely two constant predicates, written  $\top, \perp : A \longrightarrow \mathbb{B}$ , provided that  $\top : A \longrightarrow I$  is epi, meaning that  $\left( A \xrightarrow{\top} I \xrightarrow{x} X \right) =$



$$\left( A \xrightarrow{\cdot} I \xrightarrow{\cdot} X \right) \text{ implies } x = x'.$$

**Swap elements.** A predicate  $q: A \rightarrow \mathbb{B}$  that is not constant must take both truth values. We define *swap elements* of  $A$  with respect to  $q$  to be a pair  $a_\top, a_\perp : A$  such that

$$q(a_\top) = \top \quad \text{and} \quad q(a_\perp) = \perp$$

**A non-constant predicate cannot be both extensional and decidable**, unless  $\top = \perp$ . Towards contradiction, assume that  $p$  is decidable and extensional. Using the decidability assumption, this time we define a cartesian function  $\sim: A \rightarrow A$  to flip  $p$ 's truth values:

which gives

$$\sim x = \begin{cases} a_\perp & \text{if } q(x) = \top \\ a_\top & \text{if } q(x) = \perp \end{cases} \quad (5.16)$$

This is a *q-swap negation over A*. It swaps the elements of  $A$  into its swap values, so that the truth values of the predicate  $q$  get swapped:

and thus  $q(\sim x) = \neg q(x)$  (5.17)

The Fundamental Theorem gives a Kleene fixpoint  $\Psi : \mathbb{P}$  such that

$$\{\sim\Psi\} = \{\Psi\} \quad (5.18)$$

Since  $p$  is extensional, (5.18) implies

$$q(\sim\Psi) = q(\Psi) \quad (5.19)$$

Hence

$$q(\Psi) \stackrel{(5.19)}{=} q(\sim\Psi) \stackrel{(5.17)}{=} \neg q(\Psi) \quad \text{!}$$

The assumption that  $q$  is a decidable, extensional, and non-constant thus leads to a contradiction. If  $q$  is decidable, then  $q(\Psi)$  must output a truth value. Then  $q(\Psi) = \neg q(\Psi)$  implies  $\top = \perp$ .

**The two ways of living with undecidability.** Rice's Theorem is a rigid no-go result, ending all hope for deciding any nontrivial properties of computations by looking at their programs without running them. There are just two ways out of the triviality of the extensional decidable properties:

a) drop the extensionality, and study the decidable *intensional* properties, or

b) drop the decidability, and study the extensional *computable* properties.

Approach (a) leads into the realm of *algorithmics* and computational complexity [8, 20, 120]. It studies the algorithms behind programs and measures the resources needed for program evaluation. Approach (b) leads into the realm of *semantics* and domain theory of computation [2, 64, 176]. It studies the structure of spaces of computable functions as the domains of meaning of computation. We return to this thread in Ch. 9.

## 5.5 Workout

Rice's Theorem says that a computable predicate cannot at the same time be extensional, decidable, and nontrivial. This theorem is thus usually applied by showing that a computable predicate satisfies two of the properties, and deriving that it does not satisfy the third one.

Determine whether the following predicates are decidable:

- |                                     |   |
|-------------------------------------|---|
| a. $P_1(x) \iff \{x\}4 = 7$         | g. $P_7(x) \iff \{13\}x \uparrow$                               |
| b. $P_2(x) \iff \{4\}x = 7$         | h. $P_8(x) \iff [[x]]x \uparrow$                                |
| c. $P_3(x) \iff \{x\}25 < 1000$     | i. $P_9(x) \iff (\{x\}7 \downarrow \implies \{x\}8 \downarrow)$ |
| d. $P_4(x) \iff \{x\}2 \leq \{x\}3$ | j. $P_{10}(x) \iff \forall n. \{x\}n \leq \{x\}(n+1)$           |
| e. $P_5(x) \iff [x]2 = 2$           | k. $P_{11}(x) \iff \exists n. \{x\}n \downarrow$                |
| f. $P_6(x) \iff \{x\}13 \uparrow$   | l. $P_{12}(x) \iff \{x\}x \downarrow$                           |

Prove your claims!

## 5.6 Stories

### 5.6.1 I cannot decide whether I lie

This chapter asked what can be decided about the computation of a program without running it. The simple *general* answer is: *nothing*. There are, of course, *particular* programs where it is easy to tell what they do. The claim is that there is no *general* way to tell for all programs. More precisely, any predicate computable over all programs but extensional, and thus predicated over the corresponding functions, must leave some programs undecided, unless it blindly assigns the same truth value to all programs. That is what Rice's Theorem says. Its proof says that this is a consequence of the fact that any computable predicate over programs is also predicated over its own programs, and over the programs of its negation. A predicate over its negation negation outputs the truth value of the statement "I lie". That statement is false if it is true, and true if it is false. It is an instance of the Liar Paradox. The theorems of Gödel, Tarski, Turing, and Rice are all proved as instances of the Liar Paradox. Each of the four original proofs encodes the paradox in a different setting. Gödel encoded the formal arithmetic in itself and discovered that definitions of arithmetic functions can be encoded as numbers

[58]. Tarski encoded the semantical assignment of a logical theory in itself [164]. Turing encoded his machines and evaluated them in a universal machine [169]. He constructed the universal computer for the purpose of this encoding. Rice encoded the diagonal argument in the general framework of recursive enumerations [135]. Gödel’s encodings of functions as numbers were the first programs in a universal programming language. Turing’s universal machine was the first universal computer. Each of the four theorems used a different programming language and a different computer, but they all used the same gadget to connect the two: a program evaluator. And they all used it to construct an instance of the fixpoint from Sec. 3.2. The four no-go theorems opened a maze of paths into the logic of self-reference [73, 101, 153, 155, 156, 177], and an alley into computation.

### 5.6.2 The Church-Turing Antithesis

The Church-Turing Thesis, discussed in Sec. 4.6, presents computability theory as a quilt of diverse models of computation, stitched together by computable encodings and reductions. Church’s  $\lambda$ -calculus, Turing’s machines, and Kleene’s partial recursive functions are just parts of the quilt, together with Post’s production systems, von Neumann’s cellular automata, and many other equivalent models. But some models came to be viewed as more equivalent than others. Kleene’s model became so popular among logicians that theory of computation was called theory of recursion as a part of mathematical logic [91, 138]. Turing’s and Church’s models spanned the two coordinate axes of theoretical computer science, called “track A: algorithmics” and “track B: semantics” in some leading publishing venues, conferences, and handbooks [171, 172]. On “track A”, Turing machines are used to measure computational complexity. On “track B”, Church’s  $\lambda$ -calculi and type systems are used to assign formal meanings to programs. At times, the two “tracks” were distinguished even by the geographic locations of the respective research communities [65, 173]<sup>2</sup>.

But the Church-Turing Antithesis is much more than a gap between research communities. After the initial confluence of ideas in 1936, [54], the models of computation bifurcated into the *extensional* and the *intensional*, as soon as they were used for solving Hilbert’s *Entscheidungsproblem*. Alonzo Church’s so-

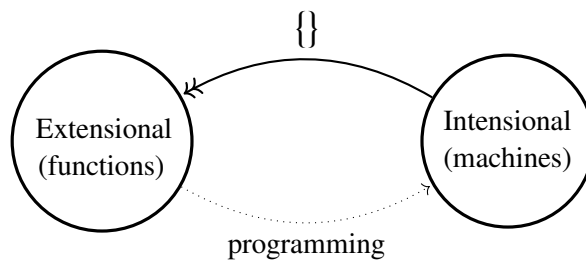


Figure 5.2: Programming is not an operation

lution [27] was based on the operations of function abstraction and application, whereas Alan Turing’s solution [169] was based on programming a universal machine to simulate computational processes. Church’s idea was to provide a high-level characterization of computable functions in terms of function abstraction. Turing provided a low-level characterization in terms of computing machines. While the operation function abstraction establishes the one-to-one correspondence in Fig. 4.13 between the func-

<sup>2</sup>Ironically, the geographic origins got switched and Church’s legacy of high-level calculi got pursued mostly in Europe, Turing’s legacy of low-level machines mostly in the US.

tions and their abstractions, program abstraction is not a routinely operation but programmers' effort to construct programs, as displayed in Fig. 5.2. The fact (worked out in Sec. 2.6.1) that each computation can be programmed in infinitely many ways is not a technicality but a fact of life. Programs do not arise by magic of abstraction as an operation, but through abstraction as creation<sup>3</sup>. The Church-Turing Antithesis is the antithesis of the extensional and the intensional views of computations: as functions and as programs. As programs are evaluated into functions, the induced extensionality relation (5.1) aligns the two views. The concept of extensionality goes back to Hermann Grassman's *linear extension theory* [60, Ch. I], an algebra of geometry whose offspring was the linear algebra that we nowadays use [61]. In set theory, the **Extensionality Axiom** says that sets are the objects that are equal if and only if they have the same elements. It follows that the functions between sets are equal if and only if they map the same elements to the same elements. In computation, many different programs are evaluated to each function. That fact, worked out in Sec. 2.6.1, should perhaps be called the **Intensionality Theorem**. The concept of intensionality goes back to Grassman again, but the usage evolved. The results presented in this chapter display the gap between the extensional and the intensional views: there are no nontrivial extensional predicates (i.e. over functions) that are decidable intensionally (i.e. over programs).

**Non-extensional function abstraction?** The correspondence imposed by the  $\lambda$ -abstraction in Fig. 4.13 does not have to be one-to-one. The original  $\lambda$ -calculus introduced by Alonzo Church in [27] was not extensional, since the *extensionality rule* of  $\eta$ -conversion, discussed in Sec. 4.6, was imposed only later. Intuitively, the rule asserts that  $\varphi a = \psi a$  implies  $\varphi = \psi$  and hence that  $\varphi a = f(a)$  implies  $\varphi = \lambda x.f$ . Without that rule, there may be  $\varphi \neq \lambda x.f$  with  $\varphi a = f(a)$  for all  $a$ , and  $\lambda x.f$  is not the sole implementation of  $f$ . Don't such non-extensional  $\lambda$ -calculi capture the intensional aspects of computation?

**The non-extensional function abstraction is essentially extensional.** A sequence of results in the 1980s [66, 92, 146] demonstrated that every non-extensional  $\lambda$ -model contains an extensional model as a computable retract. In other words, a non-extensional model can be made extensional by identifying the  $\lambda$ -terms along a computable extensional equivalence relation, with a  $\lambda$ -abstraction contained in each equivalence class. The mere presence of function abstraction as an operation makes program abstraction into an operation.

**Bridging the gap?** The extensional models factor out the intensional properties. The intensional models, by Rice's theorem, factor out all nontrivial decidable extensional properties. Neither of the two views subsumes the other.

Beyond the bifurcation, both paths have in the meantime widened significantly. Church's high-level view of function abstraction and typing provided a mathematical foundation for the programming language design [144, 176]. Turing's low-level approach gave rise to deep mathematical analyses of the computational processes [102, 120]. The traffic across the median moves in both directions, yet reaching an agreement about a solution of equation (1) still requires scaling the language barriers.

---

<sup>3</sup>Creativity is a property of evolutionary processes. Machines may evolve and create. The original computers that Turing thought of when he defined his machines were the persons that perform computations. Future programmers may be the machines that design algorithms.

## 6 Computing programs

---

6.1	Idea of metaprogramming . . . . .	100
6.2	Compilation and supercompilation . . . . .	100
6.2.1	Compilation . . . . .	100
6.2.2	Supercompilation . . . . .	101
6.3	Metaprogramming hyperfunctions . . . . .	103
6.3.1	Ackermann's hyperfunction . . . . .	104
6.3.2	Metaprogramming parametric iteration . . . . .	106
6.3.3	Metaprogramming a hyperfunction . . . . .	107
6.4	Metaprogramming ordinals . . . . .	107
6.4.1	Collection as abstraction . . . . .	108
6.4.2	Collecting natural numbers: the ordinal $\omega$ as a program . . . . .	111
6.4.3	Transfinite addition . . . . .	111
6.4.4	Transfinite multiplication . . . . .	112
6.4.5	Transfinite hyperfunction . . . . .	113
6.4.6	Background: computable ordinal notations . . . . .	115
6.5	Workout . . . . .	116
6.5.1	Is there a fourth Futamura projection? . . . . .	116
6.5.2	Is iteration all that hyper? . . . . .	116
6.6	Stories . . . . .	117
6.6.1	Programming languages . . . . .	117
6.6.2	Software systems and networks . . . . .	119
6.6.3	Software . . . . .	121

---

## 6.1 Idea of metaprogramming

Metaprograms are programs that compute programs. The idea is in Fig. 6.1. The computation  $f$  is

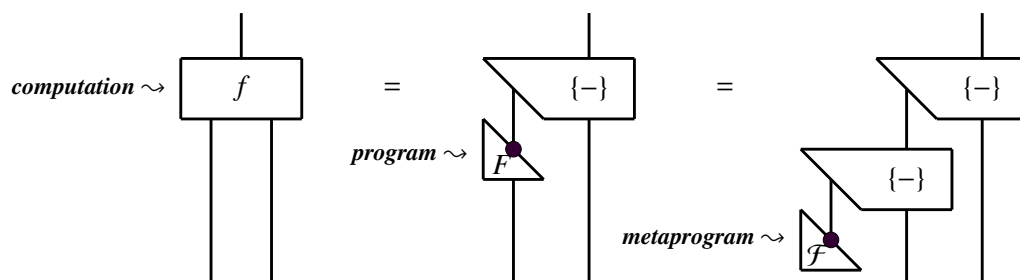


Figure 6.1: Program and metaprogram evaluations

encoded by a program  $F$  such that  $f = \{F\}$ , and  $F$  is computed by a metaprogram  $\mathcal{F}$  as  $F = \{\mathcal{F}\}$ . While the theory of computation arose from the idea that computable functions are just those that are programmable, one of the central tenets of the practice of computation has been to use computers to facilitate programming computers. That idea led to the invention of the high-level programming languages and has driven their evolution.

The first equation in Fig. 6.1 says that every computation  $f$  has a program  $F$  with  $f = \{F\}$ . The question arises: *How do we find a program  $F$  for a given computation  $f$ ?* Program evaluators take us from right to left in Fig. 6.1. How do we go from left to right? The answer, displayed already in Fig. 3, is the process of *programming*. It is usually left out of the models of computation, covered by the tacit assumptions that programs are produced by people, and that people live in a different realm from computers. In reality, of course, people and computers intermingle. Even more because they are different. Early on, it became clear that supplying the machine-readable programs was hard, error-prone, and very tedious. So people came up with the idea to use computers to program computers: they programmed them to translate high-level human code into low-level machine code.

## 6.2 Compilation and supercompilation

### 6.2.1 Compilation

An important aspect of programming and of the process of software development is that we can write metaprograms not just to translate human programs into machine programs, but also to transform and *improve* programs. This is done by *compilers*. The basic idea is already in Fig. 6.1 (but see also Fig. 6.14). Besides translating programs, compilers automate some routines, such as memory management. They often also optimize program structure, since what is simple for a programmer may not be efficient for the computer.

A compiler is thus a metaprogram that optimizes program code while translating it from a high-level language into a low-level language. Fig. 6.2 shows the idea from the opposite direction. Here you are given a friendly high-level language  $\mathbb{H}$ , with a slow universal evaluator  $h$ , and an unfriendly low-level language  $\mathbb{L}$  with a fast universal evaluator  $\ell$ . The compilation phase combines the best from both worlds: the friendly high-level programs are transformed into efficient low-level programs. All about

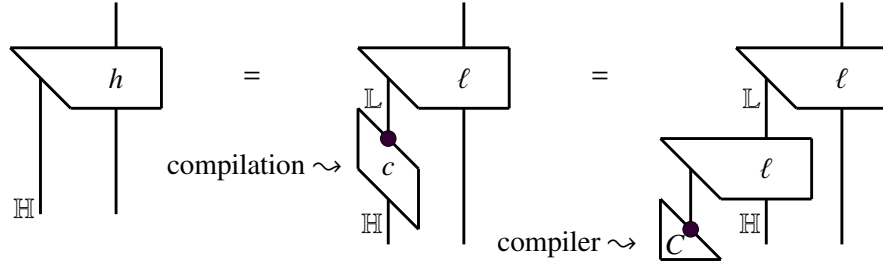


Figure 6.2: Compiler is a metaprogram that transforms programs from High-level language to Low-level language

compilation can be found in the Dragon Book [5].

### 6.2.2 Supercompilation

Leaving the environment-dependent program-optimizations aside, compilation can systematically improve efficiency of computations already by *partially evaluating the evaluators on code*. This idea was put forward and systematically explored under the name *supercompilation* by Valentin Turchin [167]. Much later, but independently, Yoshihiko Futamura presented in [53] an illuminating family of super-compilation constructions, which came to be called *Futamura projections*.

The initial setting is like in Fig. 6.2. We are given a friendly but inefficient universal evaluator  $h$  for a high-level language  $\mathbb{H}$  and an unfriendly but efficient evaluator  $\ell$  for a low-level language  $\mathbb{L}$ . We need a compiler, to translate easy  $\mathbb{H}$ -programs to efficient  $\mathbb{L}$ -programs. On the other hand, since the program evaluators  $h$  and  $\ell$  are universal, they can be programmed on each other. In other words, there is an  $\mathbb{L}$ -program  $H$  and an  $\mathbb{H}$ -program  $L$  such that

$$\{H\}_{\mathbb{L}} = h \quad \text{and} \quad \{L\}_{\mathbb{H}} = \ell$$

Such metaprograms are called *interpreters*:  $H$  interprets  $\mathbb{H}$  to  $\mathbb{L}$ , whereas  $L$  interprets  $\mathbb{L}$  to  $\mathbb{H}$ . It is nice that we can write a low-level interpreter  $L$  in a friendly language  $\mathbb{H}$ , but if we want to make it efficient, we must compile it to  $\mathbb{L}$ . So it is even nicer that we can write the high-level interpreter  $H$  in an efficient language  $\mathbb{L}$  — *because*

$$\{X\}_{\mathbb{H}} y \stackrel{(0)}{=} \{H\}_{\mathbb{L}} (X, y) \stackrel{(1)}{=} \{[H]_{\mathbb{L}} X\}_{\mathbb{L}} y$$

Partially evaluating the  $\mathbb{H}$ -to- $\mathbb{L}$ -interpreter  $H$  on  $\mathbb{H}$ -programs  $X$  thus compiles the  $\mathbb{H}$ -programs to  $\mathbb{L}$ . This is displayed in Fig. 6.3. A program transformer in the form  $C_1 = [H]$  is an instance of the *first Futamura projection*.

An  $\mathbb{H}$ -to- $\mathbb{L}$ -compilation can thus be realized by partially evaluating an  $\mathbb{H}$ -to- $\mathbb{L}$ -interpreter. But where is the underlying  $\mathbb{H}$ -to- $\mathbb{L}$ -compiler? Since  $\ell$  is a universal evaluator, there is of course an  $\mathbb{L}$ -program  $S$  that implements the partial  $\mathbb{L}$ -evaluator and which can then be partially evaluated itself, giving

$$[X]_{\mathbb{L}} y \stackrel{(2)}{=} \{S\}_{\mathbb{L}} (X, y)$$

Metaprograms that implement partial evaluators are usually called *specializers*, just like programs that

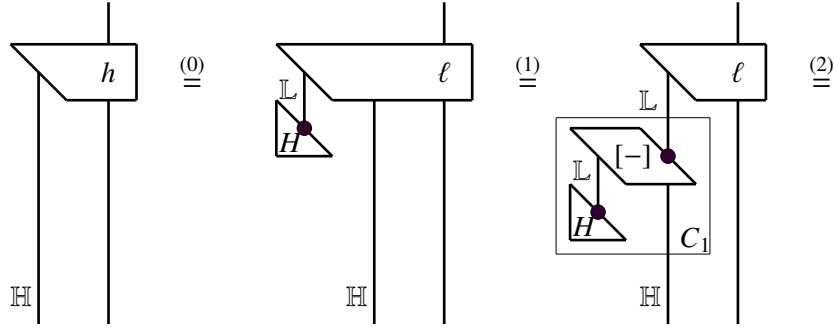


Figure 6.3: First Futamura projection: Compilation by partial evaluation of an interpreter

implement universal evaluators are called interpreters. Now partially evaluating  $S$  just like we partially evaluated  $H$  gives

$$\{S\}_{\mathbb{L}}(X, y) \stackrel{(3)}{=} \{[S]_{\mathbb{L}} X\}_{\mathbb{L}} y$$

Fig. 6.4 displays an interesting phenomenon: *partially evaluating a **specializer** on an **interpreter** gives a **compiler***. A compiler in the form  $C_2 = [S]_{\mathbb{L}} H$  is called a *second Futamura projection*.

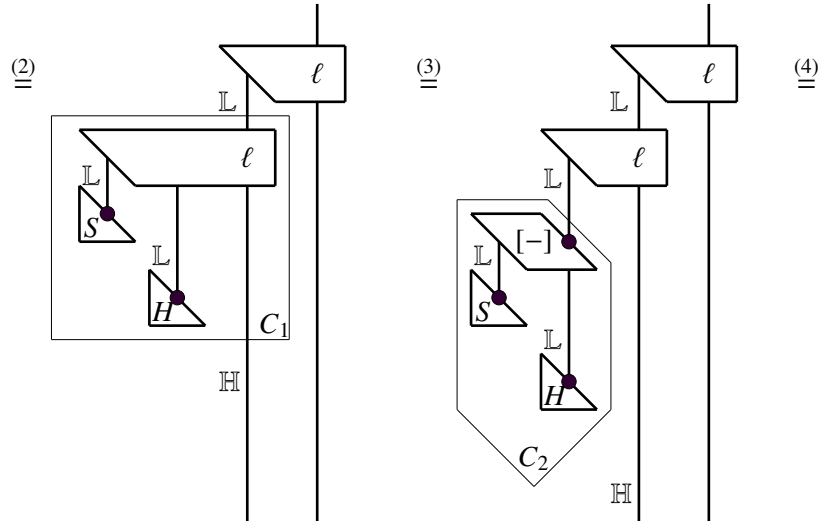


Figure 6.4: Second Futamura projection: A compiler by partial evaluation of a specializer on an interpreter

If the cost of the compiler itself is taken into account, then there is a sense in which compilation by partial evaluation is optimal. Can a further compilation gain be obtained by iterating the same trick? Well, there is still a partial evaluator inside  $C_2$ , which means that the specializer  $S$  can be partially evaluated again. The thing is that the partial evaluators inside  $C_2$  partially evaluates  $S$  itself. So now we have

$$[S]_{\mathbb{L}} H \stackrel{(4)}{=} \{S\}_{\mathbb{L}}(S, H)$$



which now partially evaluates to

$$\{S\}_{\mathbb{L}}(S, H) \stackrel{(5)}{=} \{[S]_{\mathbb{L}} S\}_{\mathbb{L}} H$$

Fig. 6.5 shows that the program in the form  $C_3 = [S]_{\mathbb{L}} S$ , called the *third Futamura projection*, generates a compiler, since  $\{C_3\}_{\mathbb{L}} = C_2$ , and  $C_2$  is a compiler. So we have yet another interesting phenomenon:

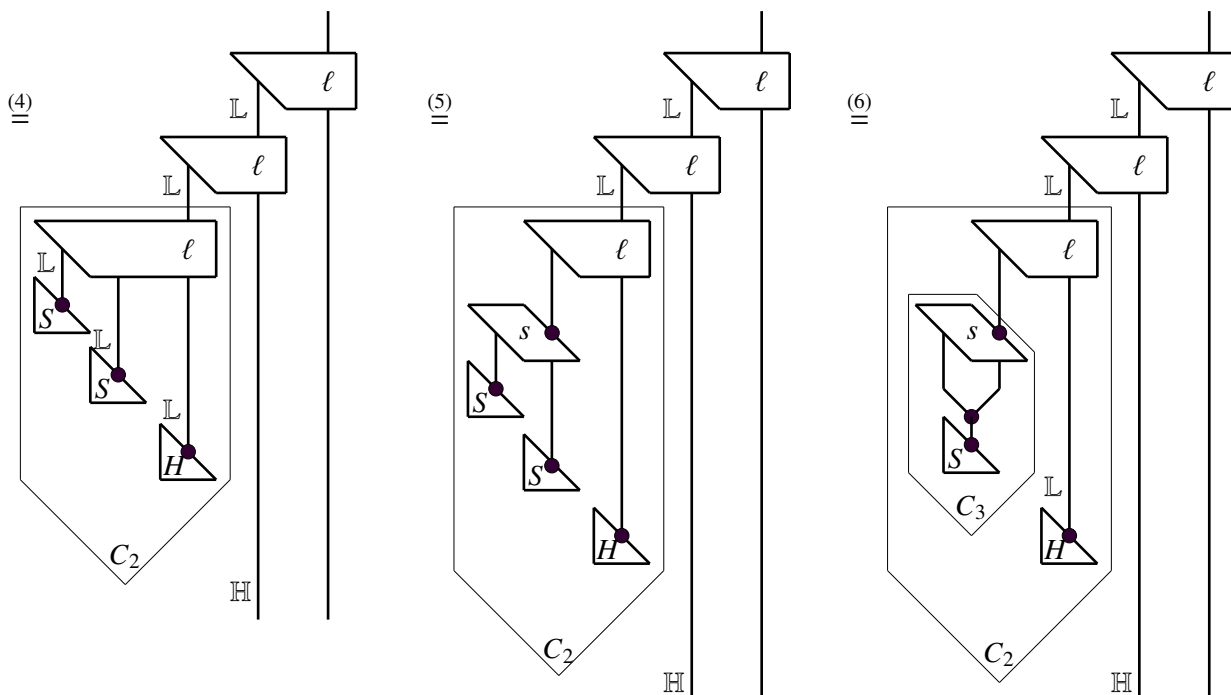


Figure 6.5: Third Futamura projection: A compiler generator by partial evaluation of a specializer on itself

partially evaluating a *specializer* on a *specializer* gives a *compiler generator*. So we have

$$\{x\}_{\mathbb{H}} y = \{C_1 x\}_{\mathbb{L}} y \quad \text{and} \quad C_1 = \{C_2\}_{\mathbb{L}} \quad \text{and} \quad C_2 = \{C_3\}_{\mathbb{L}}$$

Partially evaluating these metaprograms each time improves the execution of some programs. Sometimes significantly! If a speedup by a factor 10 is incurred each time, then applying the three Futamura projections one after the other gives a speedup by a factor 1000.

### 6.3 Metaprogramming hyperfunctions

So far, we studied the role of metaprogramming in the system programming. System programs manage programs, so it is natural that many of them are metaprograms. But metaprogramming is also an important technique in application programming. Some functions are hard to program but easy to metaprogram. In other words, it may be hard to write a program to compute a function, but easy to write a program that will output a program that will compute the function.

How can this be? An intuitive but vague way to understand this is to remember that, in many sports, it is easier to be the coach, than to be a player. Metaprograms can coach programs towards better computations. A more specific reason why metaprogramming often simplifies application programming is that a function, say  $f : A \times A \rightarrow B$ , may be such that

- the function  $\hat{f} : A \rightarrow B$  defined by  $\hat{f}(x) = f(x, x)$  is hard to program, but
- the functions  $f_a : A \rightarrow B$ , where  $f_a(x) = f(a, x)$  for  $a \in A$  (or  $f^a : A \rightarrow B$ , where  $f^a(x) = f(x, a)$ ) are much easier.<sup>1</sup>

This means that it is hard to find a program  $\hat{F}$  such that  $\{\hat{F}\} = \hat{f} : A \rightarrow B$ , but easy to find a metaprogram  $F$  such that  $\{F\} : A \rightarrow \mathbb{P}$  gives  $\{\{F\}a\} = f_a : A \rightarrow B$  for every  $a \in A$ . The metaprogram  $F$  thus computes the partial evaluations of a computation for  $f : A \times A \rightarrow B$  over the first argument. Another metaprogram  $\tilde{F}$  would give the partial evaluations  $\{\tilde{F}\} : A \rightarrow \mathbb{P}$  over the second argument, i.e.  $\{\{\tilde{F}\}a\} = f^a : A \rightarrow B$ . Functional programming (in languages like Lisp or Haskell) is particularly well suited for such metaprogramming ideas. On the imperative side, the language Nim is designed with an eye on metaprogramming: as a high-level language that compiles to high level programming languages, including C++ and JavaScript.

### 6.3.1 Ackermann's hyperfunction

In Exercises 4.5, we saw how arithmetic operations arise from one another, starting from the *successor*  $s(n) = 1 + n$ , and then proceeding with

- *addition*  $m + (-) : \mathbb{N} \rightarrow \mathbb{N}$  as iterated successor,
- *multiplication*  $m \cdot (-) : \mathbb{N} \rightarrow \mathbb{N}$  as iterated addition,
- *exponentiation*  $m^{(-)} : \mathbb{N} \rightarrow \mathbb{N}$  as iterated multiplication.

It is clear that the sequence continues, provided that we surmount the notational obstacle of iterated exponentiation in the form:

$$m^{m^{\dots m}}$$

which seems hard to iterate any further. But just a cosmetic change in notation, writing the exponentiation in the form  $m \uparrow n$  instead of  $m^n$ , allows listing a recurrent sequence of recursive definitions of basic arithmetic operations:

$1+0 = 1$	$1+(n+1) = (1+n) + 1$
$m+0 = m$	$m+(n+1) = 1+(m+n)$
$m \cdot 0 = 0$	$m \cdot (n+1) = m+(m \cdot n)$
$m \uparrow 0 = 1$	$m \uparrow (n+1) = m \cdot (m \uparrow n)$
$m \uparrow \uparrow 0 = 1$	$m \uparrow \uparrow (n+1) = m \uparrow (m \uparrow \uparrow n)$
$m \uparrow \uparrow \uparrow 0 = 1$	$m \uparrow \uparrow \uparrow (n+1) = m \uparrow \uparrow (m \uparrow \uparrow \uparrow n)$
	$\dots$

<sup>1</sup>Such examples anticipate the realm of *parametric* complexity: a computation may be more complex in one argument and less complex in another argument. Partially evaluating over the complex argument then localizes the complexity at the metaprogram.

After the base cases settle on the multiplicative unit 1, this sequence of recursive definitions boils down to the equations

$$\begin{aligned} m \uparrow^0 n &= mn \\ m \uparrow^{1+k} 0 &= 1 \\ m \uparrow^{1+k} (1+n) &= m \uparrow^k (m \uparrow^{1+k} n) \end{aligned} \tag{6.1}$$

The question whether these equations can be subsumed under primitive recursion was explored by Wilhelm Ackermann<sup>2</sup>. It is easy to see that the function  $\uparrow: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  grows very fast:

$$\begin{aligned} 3 \uparrow^0 2 &= 3 \cdot 2 = 6 \\ 3 \uparrow^1 2 &= 3^2 = 9 \\ 3 \uparrow^2 2 &= 3 \uparrow 3 = 3^3 = 27 \\ 3 \uparrow^3 2 &= 3 \uparrow^2 3 = 3 \uparrow 3 \uparrow 3 = 7,625,597,484,987 \\ 3 \uparrow^4 2 &= 3 \uparrow^3 3 = 3 \uparrow^2 3 \uparrow^2 3 = 3 \uparrow^2 7,625,597,484,987 \\ &= \underbrace{3 \uparrow 3 \uparrow 3 \uparrow 3 \uparrow \dots \uparrow 3}_{7,625,597,484,987 \text{ times}} \\ &\dots \end{aligned}$$

Ackermann proved that it grows faster than any primitive recursive function. One way to see this is to show that for every primitive recursive function  $f: \mathbb{N}^\ell \rightarrow \mathbb{N}$  there is a number  $k_f$  such that  $f(n_1, n_2, \dots, n_\ell) < 2 \uparrow^{k_f} \bar{n}$ , where  $\bar{n} = \sum_{i=1}^\ell n_i$ . Another way will be sketched as an exercise at the end of this chapter.

The arithmetic operation  $\uparrow: \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  is sometimes called a *hyperfunction*. Our goal is to show that it is easy to metaprogram. Replacing the infix notation  $\uparrow$  with the prefix notation  $\epsilon_k(n, m) = m \uparrow^k n$ , equations (6.1) become

$$\begin{aligned} \epsilon_0(n, m) &= m \cdot n \\ \epsilon_{k+1}(n, m) &= \epsilon_k^{\bullet n}(m, 1) \end{aligned} \tag{6.2}$$

where

$$f^{\bullet n}(m, x) = \begin{cases} x & \text{if } n = 0 \\ \underbrace{f(m, f(m, \dots f(m, x) \dots))}_{n \text{ times}} & \text{otherwise} \end{cases}$$

is the *parametrized iteration* operation

$$\frac{n \in \mathbb{N} \quad f: M \times A \rightarrow A}{f^{\bullet}: \mathbb{N} \times M \times A \rightarrow A}$$

defined by

$$\begin{aligned} f^{\bullet 0}(m, x) &= x \\ f^{\bullet n+1}(m, x) &= f(m, f^{\bullet n}(m, x)) \end{aligned} \tag{6.3}$$

---

<sup>2</sup>The  $\uparrow$  notation is due to Don Knuth.

The **task** is thus to metaprogram (6.3), and use it to program (6.2)

**Hierarchy of recursive functions.** Andrzej Grzegorzcyk [63] stratified the recursive functions into a tower

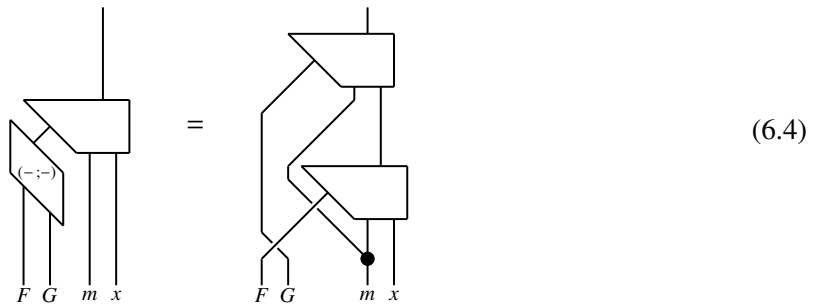
$$\begin{array}{ccccccccccc} \mathcal{E}^0 & \subset & \mathcal{E}^1 & \subset & \mathcal{E}^2 & \subset & \mathcal{E}^3 & \subset & \mathcal{E}^4 & \subset & \mathcal{E}^5 & \subset & \dots & \subset & \mathcal{E}^{k+2} & \subset & \dots \\ \Psi & & \Psi & & \Psi & & \Psi & & \Psi & & \Psi & & \dots & & \Psi & & \dots \\ \mathbf{s} & & + & & \times & & \uparrow & & \uparrow^2 & & \uparrow^3 & & \dots & & \uparrow^k & & \dots \end{array}$$

where each class also contains the data services and is closed under the composition. He proved that that every recursive function is contained in  $\mathcal{E}^k$  for some  $k$ .

### 6.3.2 Metaprogramming parametric iteration

In most programming languages, the sequential composition of programs, usually written as a semi-colon, allows that the composed programs share a parameter, passed through the composition. Denoting the parameter as  $m$ , the sequential composition is thus in the form

$$\{F; G\}(m, x) = \{G\}(m, \{F\}(m, x))$$



Let  $(-;-) : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$  be the program operation defined by (6.4)<sup>3</sup>. Iterating this operation on a program iterates its execution

$$\underbrace{\{F; F; \dots; F\}}_{n \text{ times}}(m, x) = \underbrace{\{F(m, \{F(m, \dots \{F(m, x))\} \dots))\}}_{n \text{ times}} = \{F\}^{\bullet n}(m, x)$$

To iterate this parametrized sequential composition, we define the program operation  $\widetilde{\iota} : \mathbb{P} \times \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{P}$  by

$$\widetilde{\iota}(p, F, n) = \text{ifte}(0^{\flat}(n), I, (\{p\}(F, \mathbf{r}(n)) ; F))$$

where  $\{I\}$  is a program for the identity, and  $\mathbf{r}n$  is the predecessor of  $n$ , which is 0 when  $n$  is 0. If  $J$  is the Kleene fixed point of  $\widetilde{\iota}$ , then for  $\iota = \{J\}$  holds

$$\begin{aligned} \iota(F, n) &= \{J\}(F, n) = \widetilde{\iota}(J, F, n) = \{J\}(F, n-1); F = \{J\}(F, n-2); F; F = \dots \\ &\dots = \widetilde{\iota}(J, F, 1); \underbrace{F; F; \dots; F}_{n-1 \text{ times}} = \widetilde{\iota}(J, F, 0); \underbrace{F; F; \dots; F}_{n \text{ times}} = I; \underbrace{F; F; \dots; F}_{n \text{ times}} \end{aligned}$$

<sup>3</sup>In the preceding chapter, we defined sequential composition and parallel composition ignoring the parameters, for simplicity. In programming languages, most program operations are implemented with parameters, for convenience.

and thus

$$\{\iota(F, n)\}(m, x) = \{I; \underbrace{F; F; \dots; F}_{n \text{ times}}\}(m, x) = \{F\}^{\bullet n}(m, x)$$

### 6.3.3 Metaprogramming a hyperfunction

Given  $\iota = \{J\}$  let  $W$  be the program such that

$$\{W\}(F, n, m) = \{\iota(F, n)\}(m, 1) = \{F\}^{\bullet n}(m, 1)$$

so that

$$\{[W]F\}(n, m) = \{F\}^{\bullet n}(m, 1)$$

Using this  $[W] : \mathbb{P} \longrightarrow \mathbb{P}$ , define

$$\widetilde{e}(p, k) = \text{ifte}(0^?(k), \Xi, [W](\{p\}(\mathbf{rk})))$$

where  $\{\Xi\}(n, m) = m \cdot n$ . Now if  $E$  is the Kleene fixed point of  $\widetilde{e}$ , then  $e = \{E\}$  satisfies

$$\begin{aligned} \{e(k)\}(n, m) &= \{\{E\}k\}(n, m) = \{\widetilde{e}(E, k)\}(n, m) \\ &= \begin{cases} m \times n & \text{if } k = 0 \\ \{[W](\{E\}(k-1))\}(n, m) & \text{otherwise} \end{cases} \\ &= \begin{cases} m \times n & \text{if } k = 0 \\ \{\{E\}(k-1)\}^{\bullet n}(m, 1) & \text{otherwise} \end{cases} \\ &= \begin{cases} m \times n & \text{if } k = 0 \\ \{e(k-1)\}^{\bullet n}(m, 1) & \text{otherwise} \end{cases} \end{aligned}$$

Thus the hyperoperation that we sought to metaprogram is  $\epsilon_k = \{e(k)\}$ , since this gives

$$\epsilon_k(n, m) = \begin{cases} m \times n & \text{if } k = 0 \\ \epsilon_{k-1}^{\bullet n}(m, 1) & \text{else} \end{cases}$$

as desired. The construction is summarized in Fig. 6.6.

## 6.4 Metaprogramming ordinals

We saw in Sec. 4.2 how to represent natural numbers as programs. Now we proceed to represent transfinite ordinal numbers as metaprograms. Finite descriptions of infinite processes are, of course, the beating heart of computation and of language in general. All sentences, all conversations, all books and programs in all languages are all generated by finitely many rules from finitely many words. But the finite metaprograms for transfinite numbers are not just another avatar of this miracle but also a striking example iteration and a striking display of dynamics of convergence and divergence in computation.

### 6.4.1 Collection as abstraction

**Finitely iterated abstractions.** Fig. 6.7 displays how we implement the set collection as program abstraction. Each collection step corresponds to a program abstraction:

- Ironically, the notational clash between the set-theoretic and the computational uses of the curly brackets here turns out to be convenient. The sequence  $c \ni d \ni e$  corresponds to “the time of creation” of sets [36] and the order of evaluation of programs.

108

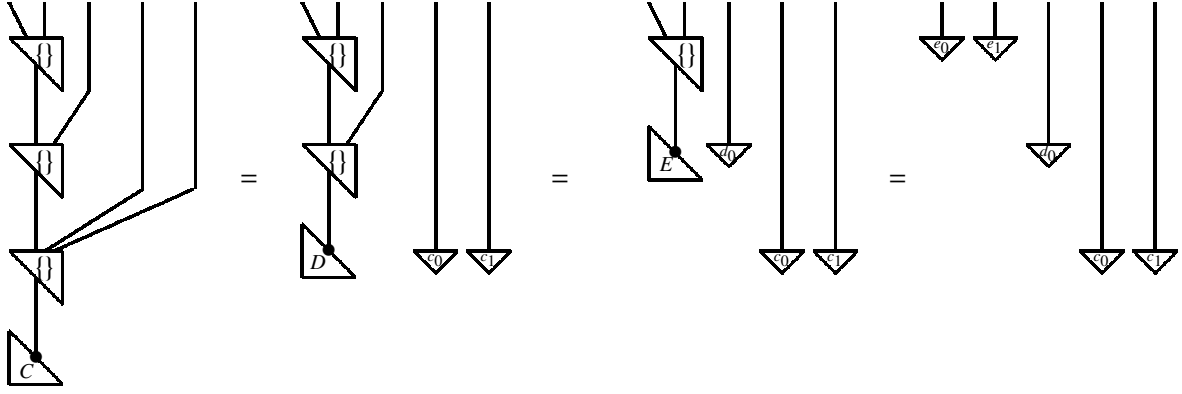


Figure 6.7: The set  $c = \{c_0, c_1, \{d_0, \{e_0, e_1\}\}\}$  represented as a program  $C$ .

**Infinite sequences.** The simplest way to generate an infinite sequence is the induction, i.e. counting infinitely long. This means that we start from the beginning  $b : B$ , and repeat the counting step  $\sigma : B \rightarrow B$  forever. The infinite sequence arises:

$$s_0 = b, \quad s_1 = \sigma(b), \quad s_2 = \sigma^2(b), \quad s_3 = \sigma^3(b), \quad s_4 = \sigma^4(b), \dots \quad (6.5)$$

In a universe of sets, this sequence can be collected into a set without much ado

$$s = \{s_0, s_1, s_2, \dots, s_n, \dots\}$$

If  $s$  is a *fundamental* sequence, i.e. it satisfies

$$s_0 < s_1 < s_2 < s_3 < s_4 < \dots \quad (6.6)$$

where  $<$  is the transitive closure of the element relation, i.e.

$$x < y \iff \exists x_1 x_2 \dots x_n. x \in x_0 \in x_1 \in x_2 \in \dots \in y$$

then

In a computer, the same can be done along the lines of Fig. 6.7. In particular, the set  $\mathbb{N}$  of the natural numbers represented as programs in Sec. 4.2.3 can be collected into a single program, corresponding to the first infinite ordinal  $\omega$ .

**Infinite iterated abstractions.** The pattern in Fig. 6.7 suggests that the sequence  $s_0, s_1, \dots, s_n, \dots$  could be captured by a sequence of programs  $S_0, S_1, \dots, S_n, \dots$  with  $\{S_n\} = \langle S_{n+1}, s_n \rangle$  for all  $n \in \mathbb{N}$ . Unfortunately, the equation  $\{S_n\} = \langle S_{n+1}, s_n \rangle$  implies that if  $\{S_{n+1}\}$  is of type  $X$ , then  $\{S_n\}$  must be of type  $X \times \mathbb{P}$ . Allow  $n$  to be arbitrarily large makes the type of  $\{S_0\}$  into an infinite product. This problem can be avoided by using the internal pairing  $\lceil -, - \rceil : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{P}$  instead of the product types. We are thus looking for a program  $S$  and the function  $\wr \sigma S$  obtained by executing it, such that

$$\wr \sigma S x = \{S\} x = \lceil \{S\}(\sigma(x)), x \rceil = \lceil \wr \sigma S(\sigma(x)), x \rceil \quad (6.7)$$

The program  $S$  can be constructed as a Kleene fixpoint of the function

$$\widehat{\sigma}(p, x) = \lceil \{p\}(\sigma(x)), x \rceil \quad (6.8)$$

in Fig. 6.8. The equation  $\lambda\sigma\mathcal{S}(\sigma(x)) = \lfloor \lambda\sigma\mathcal{S}(x) \rfloor_0$ , derived in Fig. 6.9, implies that the inductively defined

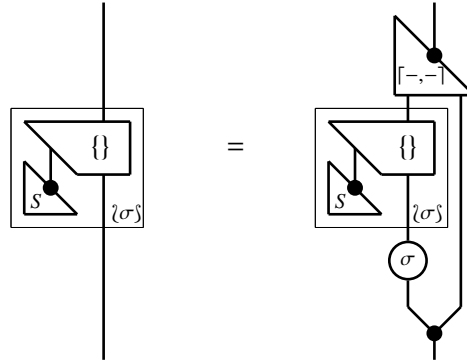


Figure 6.8: The Kleene fixpoint  $S$  of  $\widehat{\sigma}$  induces  $\lambda\sigma\mathcal{S}(x) = \{S\} x$  so that  $\lambda\sigma\mathcal{S}(x) = \lceil \lambda\sigma\mathcal{S}(\sigma(x)), x \rceil$

sequence in (6.5) now unfolds from a single program

$$\lambda\sigma\mathcal{S}_b = \lceil \lambda\sigma\mathcal{S}_{s_1}, s_0 \rceil = \lceil \lceil \lambda\sigma\mathcal{S}_{s_2}, s_1 \rceil, s_0 \rceil = \lceil \lceil \lceil \lambda\sigma\mathcal{S}_{s_3}, s_2 \rceil, s_1 \rceil, s_0 \rceil = \lceil \lceil \lceil \lceil \lambda\sigma\mathcal{S}_{s_4}, s_3 \rceil, s_2 \rceil, s_1 \rceil, s_0 \rceil = \dots \quad (6.9)$$

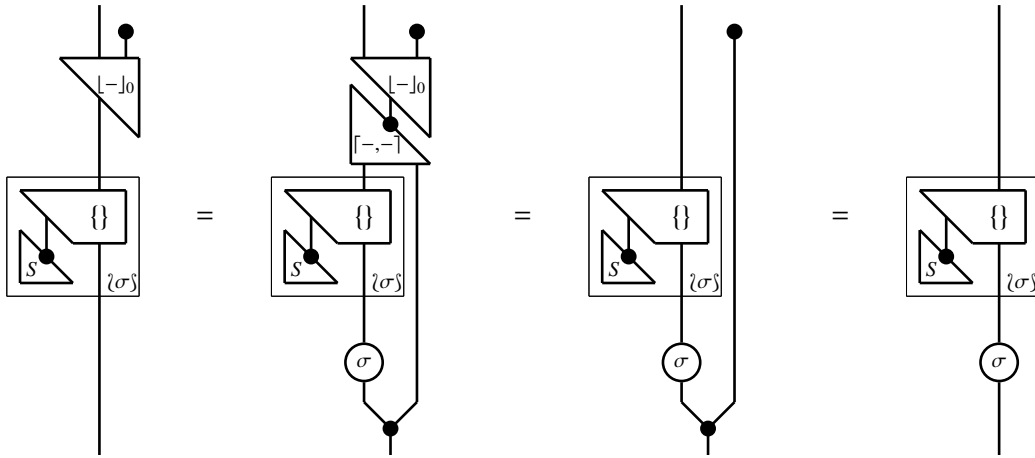


Figure 6.9: The function  $\lambda\sigma\mathcal{S} = \{S\}$  iterates  $\sigma$  within the first projections  $\lfloor \lambda\sigma\mathcal{S}(x) \rfloor_0 = \lambda\sigma\mathcal{S}(\sigma(x))$

Since  $\lambda\sigma\mathcal{S}(b)$  projects out each  $s_n = \sigma^n(b)$ , for all  $n = 0, 1, 2, \dots$ , after  $n$  iterated evaluations, it is the least upper bound of the sequence, i.e. the minimal extension of its well-order:

$$b < \sigma(b) < \sigma^2(b) < \sigma^3(b) < \sigma^4(b) < \dots < \lambda\sigma\mathcal{S}(b) = \bigvee_{n=0}^{\infty} \sigma^n(b) \quad (6.10)$$



### 6.4.2 Collecting natural numbers: the ordinal $\omega$ as a program

Instantiating  $b$  to  $\bar{0} = \lceil \top, \bar{0} \rceil$  and  $\sigma(x)$  to  $s(x) = \lceil \perp, x \rceil$  from Sec. 4.2.3, the sequence (6.5) becomes the fundamental sequence of natural numbers

$$\bar{0}, \quad \bar{1} = s(\bar{0}), \quad \bar{2} = s^2(\bar{0}), \quad \bar{3} = s^3(\bar{0}), \quad \bar{4} = s^4(\bar{0}), \dots \quad (6.11)$$

and the least upper bound (6.10) becomes

$$\bar{0} < \bar{1} < \bar{2} < \bar{3} < \bar{4} < \dots < \bar{\omega} \quad (6.12)$$

where

$$\bar{\omega} = \lambda s(\bar{0}) = \lceil \lambda s(\bar{1}), \bar{0} \rceil = \lceil \lceil \lambda s(\bar{2}), \bar{1} \rceil, \bar{0} \rceil = \lceil \lceil \lceil \lambda s(\bar{3}), \bar{2} \rceil, \bar{1} \rceil, \bar{0} \rceil = \lceil \lceil \lceil \lceil \lambda s(\bar{4}), \bar{3} \rceil, \bar{2} \rceil, \bar{1} \rceil, \bar{0} \rceil = \dots$$

Its unfolding is displayed in Fig. 6.10.

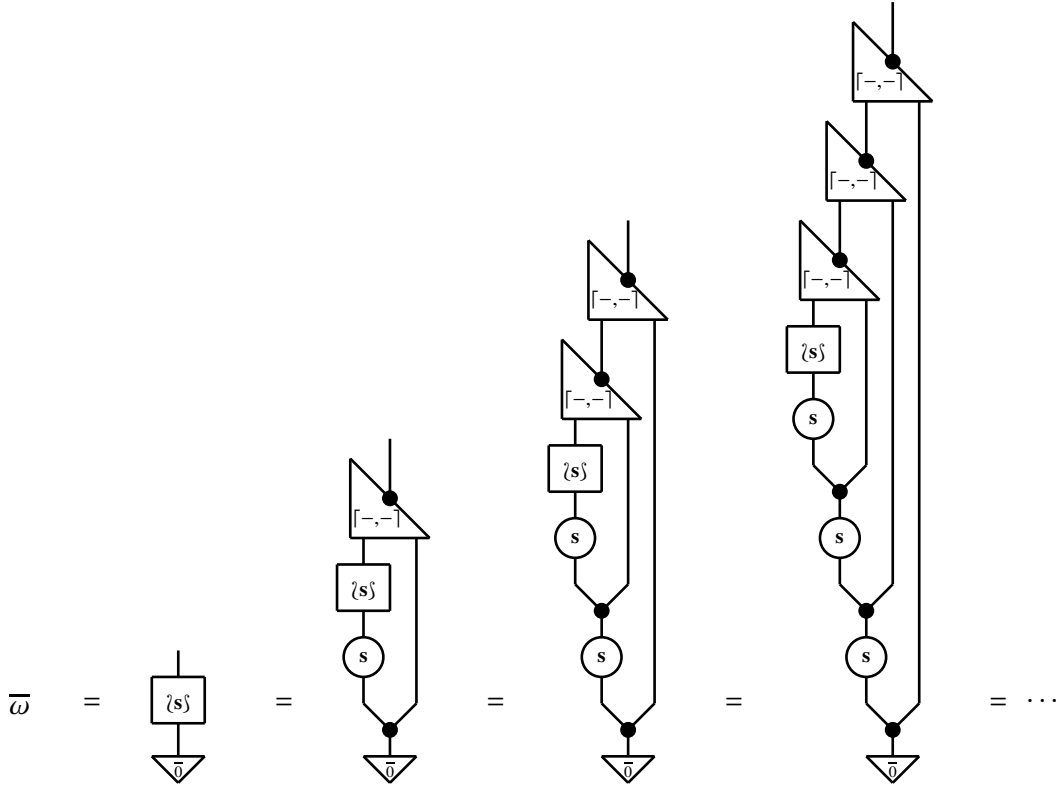


Figure 6.10: The ordinal  $\omega$  appears as the evaluation order of the program  $\bar{\omega} = \lambda s(\bar{0})$ .

### 6.4.3 Transfinite addition

The ordinal type  $1 + \omega$  is defined by adjoining an element at the bottom of  $\omega$ . The ordinal type  $\omega + 1$  is defined by adjoining an element at the top of  $\omega$ . It is easy to see that  $1 + \omega$  is order-isomorphic to  $\omega$ , whereas  $\omega + 1$  is not.

Extending the program  $\overline{\omega} = \zeta s(\overline{0})$  with an application of  $s$  before the input  $\overline{0}$  is fed to  $\zeta s$  yields  $\overline{1 + \omega} = \zeta s(s(\overline{0}))$ . The evaluation order of  $\overline{\omega}$  is thus extended in  $\overline{1 + \omega}$  with an additional step at the beginning. Defining in general  $\overline{n + \omega} = \zeta s(s^n(\overline{0}))$ , with  $n$  additional steps adjoined at the beginning, leads to the fundamental sequence

$$\overline{\omega} = \zeta s(s^0(\overline{0})), \quad \overline{1 + \omega} = \zeta s(s^1(\overline{0})), \quad \overline{2 + \omega} = \zeta s(s^2(\overline{0})), \quad \overline{3 + \omega} = \zeta s(s^3(\overline{0})), \dots \quad (6.13)$$

But the evaluation orders of all  $\overline{n + \omega} = \zeta s(s^n(\overline{0}))$  are in fact the same like  $\overline{\omega}$ : they all just iterate the computation  $\zeta s$ . The order-isomorphisms of the ordinals  $n + \omega$  and  $\omega$  are implemented on the representations  $\overline{n + \omega}$  and  $\overline{\omega}$  by precomposing with the predecessor or the successor functions.

Extending the program  $\overline{\omega} = \zeta s(\overline{0})$  by applying  $s$  to the output yields  $\overline{\omega + 1} = s(\zeta s(\overline{0}))$ . Its evaluation order is thus extended with an additional step adjoined at the end. The general definition  $\overline{\omega + n} = s^n(\zeta s(\overline{0})) = s^n(\overline{\omega})$  adjoins  $n$  additional steps at the end. The fundamental sequence induced by counting up the  $n$  is now

$$\overline{\omega} = s^0(\overline{\omega}), \quad \overline{\omega + 1} = s^1(\overline{\omega}), \quad \overline{\omega + 2} = s^2(\overline{\omega}), \quad \overline{\omega + 3} = s^3(\overline{\omega}), \quad \overline{\omega + 4} = s^4(\overline{\omega}), \dots \quad (6.14)$$

The well-ordering and its least upper bound now become

$$\overline{\omega} < \overline{\omega + 1} < \overline{\omega + 2} < \overline{\omega + 3} < \overline{\omega + 4} < \dots < \overline{\omega + \omega} \quad (6.15)$$

where

$$\overline{\omega + \omega} = \zeta s(\overline{\omega}) = \zeta s(\zeta s(\overline{0})) \quad (6.16)$$

The evaluation unfolding of the second component is displayed in Fig. 6.11. The evaluation unfolding of the first component would add Fig. 6.10 at the bottom of Fig. 6.11. The ordinal type  $\omega + \omega$  consists of two copies of  $\omega$ , one after the other, and the evaluation of  $\overline{\omega + \omega}$  reproduces that order.

#### 6.4.4 Transfinite multiplication

The multiplication of infinite ordinals is iterated addition, just like in the finite case. While the ordinal  $2 \times \omega$  consists of  $\omega$  copies of 2, the ordinal  $\omega \times 2$  consists of 2 copies of  $\omega$ , i.e.  $\omega \times 2 = \omega + \omega$ , and hence  $\overline{\omega \times 2} = \overline{\omega + \omega} = \zeta s \circ \zeta s(\overline{0}) = \zeta s^2(\overline{0})$ . The fundamental sequence is now

$$\overline{\omega \times 0} = \zeta s^0(\overline{0}), \quad \overline{\omega \times 1} = \zeta s^1(\overline{0}), \quad \overline{\omega \times 2} = \zeta s^2(\overline{0}), \quad \overline{\omega \times 3} = \zeta s^3(\overline{0}), \quad \overline{\omega \times 4} = \zeta s^4(\overline{0}), \dots \quad (6.17)$$

Following (6.10), we now get

$$\overline{0} < \overline{\omega} < \overline{\omega \times 2} < \overline{\omega \times 3} < \overline{\omega \times 4} < \dots < \overline{\omega \times \omega} \quad (6.18)$$

where

$$\overline{\omega \times \omega} = \zeta \zeta s(\overline{0}) \quad (6.19)$$

The evaluation is unfolded in Fig. 6.12. Each  $\zeta s$ -box in Fig. 6.12 further unfolds like in Fig. 6.10. The evaluation order thus consists of  $\omega$  copies of  $\omega$ , as a representation of  $\omega \times \omega$  should.

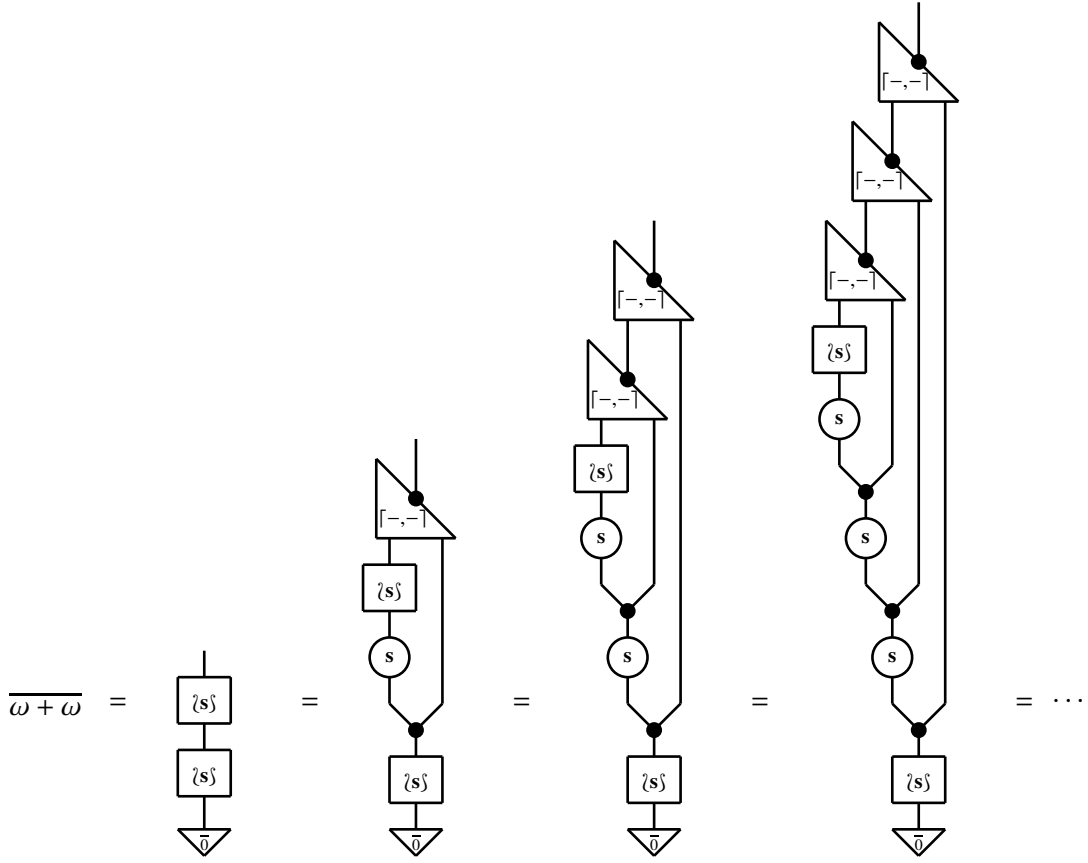


Figure 6.11: The ordinal  $\omega + \omega$  appears as the evaluation order of the program  $\overline{\omega + \omega} = \zeta s(\zeta s(\overline{0}))$ .

### 6.4.5 Transfinite hyperfunction

Like in the finite case, the exponentiation of infinite ordinals is iterated multiplication again, with

$$\omega^2 = \omega \times \omega \quad \text{suggesting the representation} \quad \overline{\omega^2} = \overline{\omega \times \omega} = \zeta \zeta s(\zeta s(\overline{0}))$$

The fundamental sequence of metaprograms for finite exponents thus becomes

$$\overline{\omega^0} = \overline{1}, \quad \overline{\omega^1} = \zeta s(\overline{0}), \quad \overline{\omega^2} = \zeta \zeta s(\zeta s(\overline{0})), \quad \overline{\omega^3} = \zeta \zeta \zeta s(\zeta s(\zeta s(\overline{0}))), \quad \overline{\omega^4} = \zeta \zeta \zeta \zeta s(\zeta s(\zeta s(\zeta s(\overline{0})))) \dots \quad (6.20)$$

To iterate the  $\zeta$ -construction, we need a program transformer  $\mathbf{w}$  such that

$$\zeta\{q\} = \{\mathbf{w}(q)\} \quad (6.21)$$

Recalling  $\zeta\sigma(x)$ , was defined in (6.7) to unfolds to  $[\zeta\sigma(\sigma(x)), x]$ , the program transformer  $\mathbf{w}$  can thus be obtained by partially evaluating the Kleene fixpoint  $W$  of the function

$$\widehat{w}(p, q, x) = [\{p\}(q, \{q\}x), x] \quad (6.22)$$

as displayed in Fig. 6.13. The ordinal representations so far can be reconstructed using a program  $N$  for

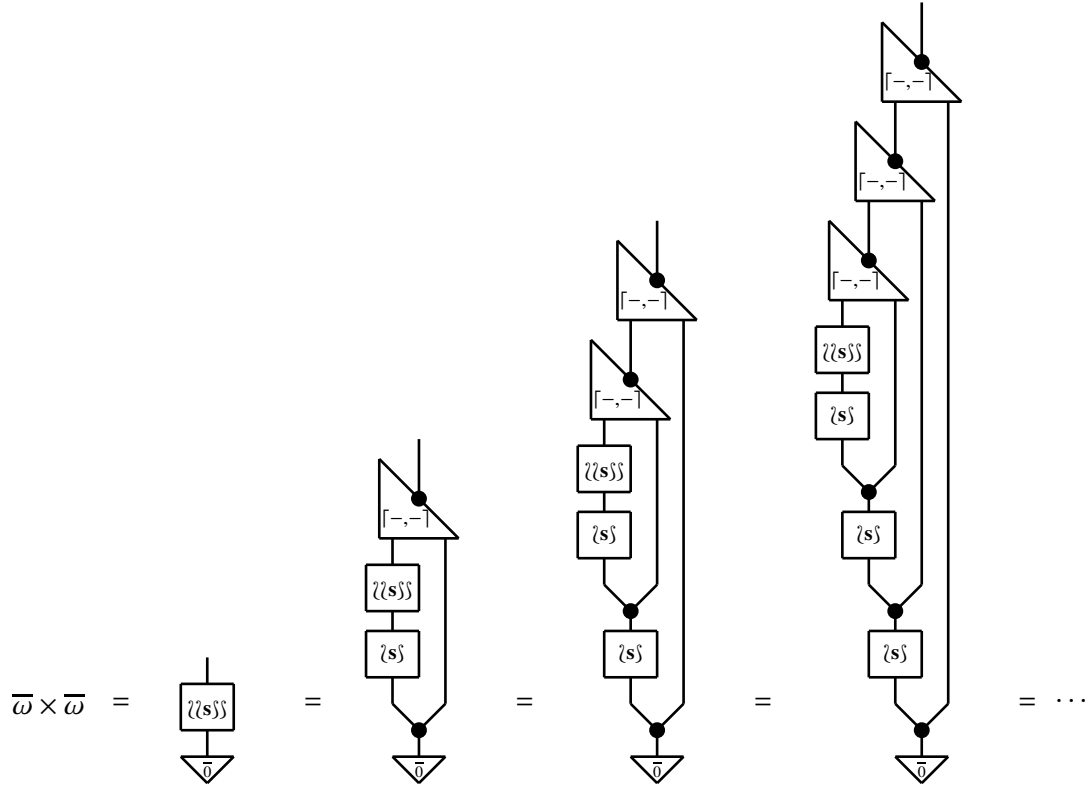


Figure 6.12: The ordinal  $\omega \times \omega$  appears as the evaluation order of the program  $\overline{\omega \times \omega} = \lambda s. s(\overline{0})$ .

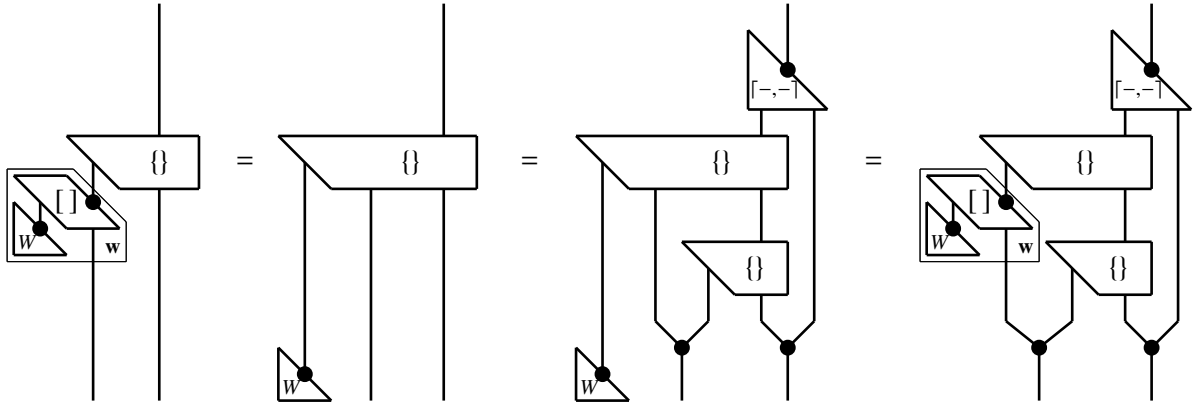


Figure 6.13: If  $\{W\}(q, x) = \{ \{W\}(q, \{q\} x), x \}$ , then  $\mathbf{w} = [W]$  gives  $\{\mathbf{w}(q)\} = \lambda \{q\}$

the successor function  $\mathbf{s} = \{N\}$ , in the form

$$\begin{aligned} \overline{\omega} &= \{\mathbf{w}(N)\} \overline{0} \\ \overline{\omega + \omega} &= \{\mathbf{w}(N)\}^2 \overline{0} \\ \overline{\omega \times \omega} &= \{\mathbf{w}^2(N)\} \overline{0} \\ \overline{\omega^\omega} &= \{\mathbf{w}^{(2)}(N)\} \overline{0} \end{aligned}$$

where  $\mathbf{w}^{(2)}$  is the iterator of  $\mathbf{w}$ , i.e.

$$\{\mathbf{w}^{(2)}(q)\} = \{\mathbf{w} \circ \mathbf{w}^{(2)}(q)\} = \{\mathbf{w} \circ \mathbf{w} \circ \mathbf{w}^{(2)}(q)\} = \{\mathbf{w} \circ \mathbf{w} \circ \mathbf{w} \circ \mathbf{w}^{(2)}(q)\} = \dots$$

Continuing with the iterations

$$\begin{aligned} \{\mathbf{w}^{(3)}(q)\} &= \{\mathbf{w}^{(2)} \circ \mathbf{w}^{(3)}(q)\} = \{\mathbf{w}^{(2)} \circ \mathbf{w}^{(2)} \circ \mathbf{w}^{(3)}(q)\} = \{\mathbf{w}^{(2)} \circ \mathbf{w}^{(2)} \circ \mathbf{w}^{(2)} \circ \mathbf{w}^{(3)}(q)\} = \dots \\ \{\mathbf{w}^{(4)}(q)\} &= \{\mathbf{w}^{(3)} \circ \mathbf{w}^{(4)}(q)\} = \{\mathbf{w}^{(3)} \circ \mathbf{w}^{(3)} \circ \mathbf{w}^{(4)}(q)\} = \{\mathbf{w}^{(3)} \circ \mathbf{w}^{(3)} \circ \mathbf{w}^{(3)} \circ \mathbf{w}^{(4)}(q)\} = \dots \\ &\dots \quad \dots \quad \dots \end{aligned}$$

we reconstruct the ordinal version of the Ackermann hyperfunction

$$\begin{aligned} \overline{\omega \uparrow \omega} &= \overline{\omega^\omega} = \overbrace{\omega \times \omega \times \omega \times \dots}^{\omega \text{ times}} = \{\mathbf{w}^{(2)}(N)\} \bar{0} \\ \overline{\omega \uparrow \uparrow \omega} &= \overbrace{\omega \uparrow \omega \uparrow \omega \uparrow \omega \uparrow \omega \uparrow \dots}^{\omega \text{ times}} = \{\mathbf{w}^{(3)}(N)\} \bar{0} \\ &\dots \\ \overline{\omega \uparrow^{1+n} \omega} &= \overbrace{\omega \uparrow^n \omega \uparrow^n \omega \uparrow^n \omega \uparrow^n \dots}^{\omega \text{ times}} = \{\mathbf{w}^{(2+n)}(N)\} \bar{0} \end{aligned}$$

where  $N$  is still a program for the successor function  $\mathbf{s}(x) = \lceil \perp, x \rceil$ . The set-theoretic version of the ordinal Ackermann hyperfunction goes back to [44]. Deployed in a programming language  $\mathbb{P}$ , the sequence of programs

$$\mathbf{w}^{(2)}(N) = \varpi_0, \quad \mathbf{w}^{(3)}(N) = \varpi_1, \quad \mathbf{w}^{(4)}(N) = \varpi_2, \quad \dots \quad \mathbf{w}^{(n+2)}(N) = \varpi_n, \dots \quad (6.23)$$

is generated by iterating the Kleene fixpoint  $\tilde{W}$  of

$$\tilde{u}(p, n, q) = \text{ifte}(0^?(n), \mathbf{w}(q), \{p\}(\mathbf{r}(n), \{p\}(n, q)))$$

leading to  $\mathbf{w}^{(n)} = [\tilde{W}](n)$ . Applying the construction from Sec. 6.4.1, we get  $\Omega = \{\{\tilde{W}\}\}(N)$  as the limit of the sequence in (6.23). The definition of  $\mathbf{w}$  also gives  $\Omega = \{\mathbf{w}(\tilde{W})\}N$ . The transfinite well-orders up to  $\varepsilon_0$  can thus be metaprogrammed in any programming language  $\mathbb{P}$ .

## 6.4.6 Background: computable ordinal notations

Ordinal numbers arise as transitive closures of the element relation in set theory, or as the types of deterministic evaluations in type theory. In computers, programs are well-ordered lexicographically, usually as binaries. In recursion theory, programs have been represented as natural numbers since Gödel. We have seen that the iterative metaprogramming induces transfinitely well-ordered evaluations well beyond the natural numbers. Computable ordinal notations have been studied since the early days of theory of computation [28, 31, 32, 89, 170]. The iterative computational processes that generate ordinals go back to the origins of Georg Cantor's work on his *second number class* [23], and thus predate the theory of computation by some 50 years. His explorations of transfinite constructions can be viewed as a radical application of iterated counting: "Suppose that I have counted infinitely long and then I start again. And then I do that infinitely many times. And then I iterate that infinitely many times..." Such

metaprograms were at the heart of Cantor's work. Hilbert's endorsements may have diverted attention from their constructive content.

## 6.5 Workout

### 6.5.1 Is there a fourth Futamura projection?

The three Futamura projections are based on the same evaluation pattern. In Fig. 6.3, the first one partially evaluates the interpreter  $H$  on the  $\mathbb{H}$ -programs. In Fig. 6.4, the second one partially evaluates the specializer  $S$  on the interpreter  $H$ , previously specialized to  $\mathbb{H}$ -programs. In Fig. 6.5, the third one partially evaluates the specializer  $S$  on the instance of itself, previously specialized to the interpreter  $H$ . Continuing in the same way, a fourth projection would partially evaluate the specializer  $S$  on the instance of itself, previously specialized to another instance of itself. Does this projection improve compilation in the same way as the preceding ones?

### 6.5.2 Is iteration all that hyper?

Looking at Fig. 6.6, one might wonder whether we really need metaprogramming here. The function  $e : \mathbb{N} \rightarrow \mathbb{P}$  is clearly primitive recursive, with  $e = \langle \Xi, [W] \rangle$ . Its diagrammatic definition in Fig. 6.6 is a special case of the schema in Fig. 4.8. The metaprogram for  $e$  builds the programs for Ackermann's hyperfunction recursively. The following exercises explore whether this primitive recursive metaprogram can be transformed into a primitive recursive program. Do we really need metaprogramming here?

- a. Fig. 6.6 shows that the computation of  $m \uparrow^k n$  boils down to the primitive recursion of  $e(k)$ , and just an  $\{\}$ -evaluation of its output  $e(k)$  on  $n$  and  $m$ , to get  $\{e(k)\}(n, m) = m \uparrow^k n$ .

Answer the following questions.

- i) Given the functions  $f : \mathbb{N} \times A \rightarrow \mathbb{P}$ ,  $g : A \rightarrow \mathbb{P}$ , and  $h : \mathbb{N} \times \mathbb{P} \times A \rightarrow \mathbb{P}$ , define

$$\begin{aligned}\varphi(k, x, y) &= \{f(k, x)\}y \\ \gamma(x, y) &= \{g(x)\}y \\ \chi(k, u, x, y) &= \{h(k, u, x)\}y\end{aligned}$$

Is it true that

$$f = \langle g, h \rangle \stackrel{?}{\Longrightarrow} \varphi = \langle \gamma, \chi \rangle \quad (6.24)$$

Draw the programs and explain the answer.

- b. Consider the following function  $A : \mathbb{N}^2 \rightarrow \mathbb{N}$  (defined and studied by Rozsa Péter [131])

$$\begin{aligned}A(0, n) &= n + 1 \\ A(k + 1, 0) &= A(k, 1) \\ A(k + 1, n + 1) &= A(k, A(k + 1, n))\end{aligned}$$

Prove the following claims:

- i)  $A(k, n) = 2^{\uparrow^{k-2}(n+3)} - 3$
- ii) Let  $\mathcal{A} = \{f : \mathbb{N}^\ell \rightarrow \mathbb{N} \mid \exists k \forall \vec{x} \in \mathbb{N}. f(\vec{x}) < A(k, \hat{x})\}$ , where  $\hat{x} = \max\{x_1, x_2, \dots, x_\ell\}$  for  $\vec{x} = (x_1 x_2 \dots x_\ell)$ . Then the class  $\mathcal{A}$  contains the data services and the successor function, and it is closed under composition and recursion.
- iii) The function  $A$  is not primitive recursive.
- iv) Ackermann's hyperfunction  $\uparrow : \mathbb{N}^3 \rightarrow \mathbb{N}$  is not primitive recursive.

## 6.6 Stories

### 6.6.1 Programming languages

Programming is the art of making computers do what you want.<sup>5</sup> In the first instance, making a computer do what you want is not all that different from building and running any other machine: you try to anticipate what the components will do, things get complicated, a lot of trial and error. Writing a program directly in a machine language (e.g. as a list of 5-tuples describing a Turing machine to a universal Turing machine) can be immensely frustrating. Some parts are hard to get right, while other parts are repetitive and deadly boring.

That was the state of affairs in the early days of computer programming. Programming early computers was similar to programming the clockwork-based calculators, built for particular calculations since XVIII century. But computers are different from calculators. Computers can be programmed to do *anything*. So it didn't take long before people came up with the idea that *computers* could be used to simplify computer programming. E.g., certain fixed lists of machine instructions are repeated in many programs. When executed, they perform some frequently reoccurring operations, such as copying data from one memory location to another, adding binary numbers, etc. Each such list of machine instructions is grouped into a single *program instruction*; and then a machine program is written, instructing the computer to replace each program instruction with the corresponding list of machine instructions. These program instructions form a *programming language*. A programming language whose program instructions are directly translated into machine instructions is called an *assembly language*. A metaprogram that instructs the computer how to replace the assembly language instructions with lists of machine instructions is called an *assembler*. The idea is displayed in Fig. 6.14. The strings  $\mathbb{M}$ ,  $\mathbb{L}$  and  $\mathbb{H}$  correspond to three types of programs, in three types of programming languages: the  $\mathbb{M}$ achine languages, the  $\mathbb{L}$ ow-level languages, and the  $\mathbb{H}$ igh-level languages. The program evaluator evaluates programs written in the machine language. One such program is  $P_m$ . Another such program is an assembler  $A$ . At least initially, an assembler must also be written in the machine language, because there are no metaprograms, before an assembler is written, that would translate the assembler into the machine code. However, once an assembler is available, then another assembler can be written in the assembly language, and assembled

<sup>5</sup>If the programming task, a function  $f$ , is viewed as a problem, then a program  $F$  that computes it as  $f = \{F\}$  is a solution. Programmers thus infuse computers with their problem-solving capacity: the *intelligence*. If computers acquire a problem-solving capacity, and return the favor, then the process of programming will be inverted, and the intelligence will flow in the opposite direction. Artificial intelligence can in that sense become dual to programming. If programming is the process of people telling computers what to do, then artificial intelligence may be the process of computers telling people what to do. The two processes may interleave into a conversation between people and computers.

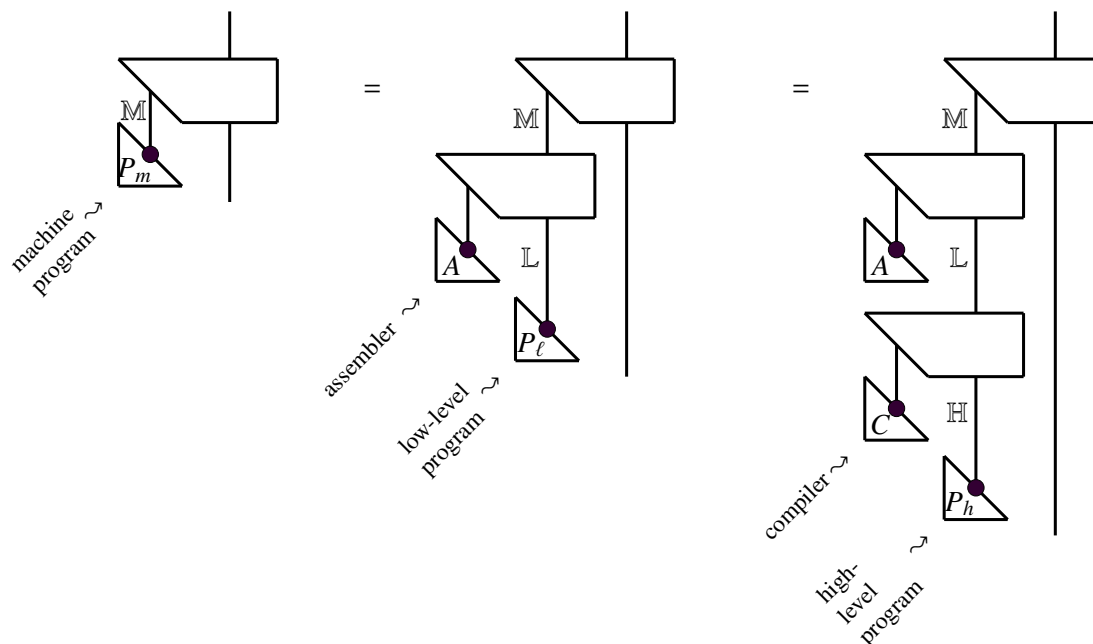


Figure 6.14: Computing machine programs from low-level programs from high-level programs

into the machine language by the first assembler.

It quickly becomes clear that the idea of translating human-written programs into machine-executable programs can be iterated. E.g., it often happens that certain fixed lists of assembly instructions are often repeated in programs of a certain kind, usually within the same application domain. It is then useful to group each such list of *low-level* program instructions into a single *high-level* program instruction, and another metaprogram is written to translate such high-level program instructions into lists of low-level program instructions. A stack of programming languages is created, and the metaprograms are written to translate high-level code to low-level code. Already in the 1950s, for the most important application domains, programmers started developing high-level programming languages. The early ones were the *Algorhythmic language Algol*, then also FORTRAN, for FORMula TRANslations and calculations in science and engineering, and a COMmon Business-Oriented Language, standardized in the late 1950s as COBOL. *Algol* was said to be "so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors" [72]. It was probably one of the most studied but least used programming languages in the history of computation. FORTRAN and COBOL are still in use, though for different reasons. COBOL was largely designed by Grace Murray Hopper, one of the first programmers, see Fig. 6.15. She did not name her language, but she did name a basic concept of computation: the *bug*. The primordial software bug first appeared in Grace Murray Hopper's lab note, see Fig. 6.16. She had long suspected that software failures were caused by bugs, and when she finally caught one, stuck in the computer hardware, she duly taped it in the notebook.

Ever since those early days, programming was facilitated by a steady stream of diverse high-level languages, often very domain-specific, habit-specific, and even taste-specific. In 1977 the US Department of Defense (DoD) counted 450 different programming languages in their codebase, and initiated a large project to design a new programming language Ada, to supersede all of them. In 1991, the Ada project was declared a victory, and the DoD mandated that all software procured for them be written in Ada, except in the exceptional, explicitly approved cases. In 1997, after many exceptions, the mandate was





Figure 6.15: Grace Murray Hopper

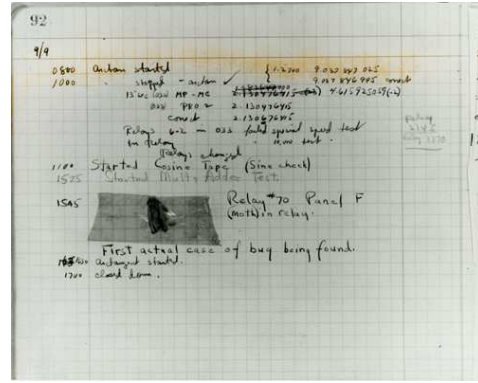


Figure 6.16: The original software bug

removed. Nowadays, the variety of programming languages is greater than ever, but software engineers seem to have adopted the linguistic diversity as a feature of their world, and not a bug.

Programming languages seem to evolve just like natural languages: they undergo a sort of natural (or social, or market) selection. Some of them accumulate random mutations.<sup>6</sup> They are still used by people to tell computers what to do; but they are used by many people to tell many computers to do many things, all at the same time. So modern programming is not just a flow of instructions from people to computers, but it involves a great deal of coordination between the programmers, and coordination between computers, and cross-coordinations of all kinds. Most importantly, as nearly all economic and social processes involve software in one way or another, the coordination involved in programming is not always cooperative, but is often competitive, and even adversarial. As social processes are becoming increasingly computational, computation is acquiring social dimensions. For better or for worse, security has grown into one of the central problems of computation. The basic concepts of software and of complexity are the main milestones on this path.

## 6.6.2 Software systems and networks

### 6.6.2.1 System programming

As software applications grew in size, they also grew in complexity. Their computational complexity got formalized and became one of the central concerns in algorithmics, and the foundation of security in computing and networking. Their architectural complexity became one of the main problems of software engineering. Through years, a whole gamut of methods was devised for partitioning software into components, and for connecting software components into software systems. In order to develop, run, and maintain application software, the programmers need to develop, run, and maintain system software. Some of it gets bundled into *operating systems*, all of it grows in size and complexity. The realm of *system programming* is thus gradually distinguished from the realm of *application programming*. Such partitions of engineering disciplines tend to be often breached, and sometimes confused. The boundary between system programming and application programming is particularly vague, as sys-

<sup>6</sup>Through the 1980s and 1990s, programming language research was based on mathematical semantics of computation, and geared towards semantically based language design. The hope was that judicious, mathematically based language design could not only increase programmers' productivity, but also preclude some programming errors. Many of my friends from that research community still do not accept the idea that programming languages evolve like natural languages. Some of them believe in evolution of natural languages, and in intelligent design of programming languages.

tem programmers often adopt certain applications into their systems, while application programmers sometimes embed system functions into their applications. On the other hand, the interactions of the goals and the needs of application programming and of system programming have left many traces and scars on the landscape of software engineering. We mention some of them.

Early system programs were written in assembly languages. But assembly programming does not scale up, and the high-level programming languages were invented and deployed as soon as large-scale programming projects had to be tackled. Probably the most prominent such language is C, which came after B, which came after BCPL. The UNIX operating system was written in it, and used for the early, friendly versions of Apple's OSX. On the other hand, although designed for system programming, C turned out to be very convenient for application programming, where it was succeeded by C++, which was succeeded by Java, etc. Like people, languages often have more children than parents. Java has aged a little (like some of us) but still occupies a particularly interesting place, where things got a little complicated.

### 6.6.2.2 Portable executables

Compiled programming languages dominated the scene as long as programs were designed by programmers, and run by computers. This meant that the *software lifetime* was clearly separated into three phases:

- *design-time*, when programmers transform ideas into programs, and
- *run-time*, when computers transform programs into computations, and in-between them there is
- *compile-time*, where compilers translate programmers' high-level programs into computers' low-level programs.

All languages mentioned so far are compiled.

These tidy times ended with the advent of the web. Java was released a couple of years into the web explosion. The web brought network interactions to a new level. Long before the web, software systems were connected into networks, and exchanged messages. But the messages were static: only data. The web significantly expanded the scope of network interactions, initially just by passing around not just data but also *metadata*, that allow web servers to send text formatting and network links to be reconstructed on the client side. Very soon, these enhanced network functions inspired *dynamic* interactions, where web servers distributed not only data and metadata but also some program code, to be executed on the client side. Such executables distributed on the web are called *applets*.

The challenge of serving applets on the web was that they had to be *portable executables*, that could be compiled on the server side, and executed on the client side. Since servers and clients run on different machines, all with different machine languages, applets had to be compiled on the server side from a high-level language into an intermediary language, high enough to allow assembly into all clients' machine languages, but low enough to allow efficient interpretation by every client. Java was designed to compile into such an intermediary language, the *Java bytecode*, which is then interpreted by different interpreters at different machines.

**Historic remark.** The idea of portable executables emerged in 1969, in the early days of ARPANET research. A machine-independent language DEL was specified, and an interpreter was prototyped at SRI

in Menlo Park. The project was discussed and definition of the language was announced in the Network Working Group RFC 5.

### 6.6.2.3 Interpreters and scripting

In a basic computer, programs are directly evaluated. If programs are coded in a high-level language, then the evaluator must be programmed in a machine language, at least initially. So there must be an interpreter for the high-level language. If we partially evaluate that interpreter, we get a compiler. That was the first Futamura projection, Fig. 6.3. Since compilation provides an opportunity for optimizations, the high-level languages of the first generation of were always compiled. But when software systems got connected into networks, the compilation phase of high-level code into a machine executable had to be split into a compilation on the server side into a portable executable, and an interpretation on the client side into a machine executable. This software evaluation scenario was realized in Java, and many variations have been realized in other languages.

Another boost for interpreters came from a different direction at the same time. Since the early days of system programming, a divide of the development tasks into *programming-in-the-small* and *programming-in-the-large* was felt and discussed. With the growth of software systems in size and scope, the large became larger, the divide increased, and new languages for programming-in-the-small started emerging: the *scripting* languages. Nowadays, we use JavaScript, PHP, Perl for web programming. They are all purely interpreted languages. Scripting for the web is metaprogramming of computations with metadata. For different reasons, Matlab, Mathematica, or any language underlying a spreadsheet is also interpreted.

**What do Java and JavaScript have in common?** Both are mentioned in the story about interpreters. Both emerged with the web. Both have "Java" in their name. Other than that, they could hardly be more different. One has objects and classes, the other throws everything on the heap. Java was designed at Sun Inc in 1991, as a language for programming appliances, like set-top boxes and washing machines. The portability requirement meant that the compiled code should be suitable for different appliances. The design was deemed "too advanced", as the thought of streaming TV was still some years away, and the web was only just conceived. So the project was shelved. But a couple of years later, in May 1995, the designers of Netscape web browser came to Sun Inc to ask for a language that could be compiled to portable executables, and got Java. They also needed an interpreter for web scripts, and when they settled on Java, they wanted to make the syntax of their scripting language maximally similar to Java. One of them, Brian Eich, specified a Java-like syntax for a scripting language over a weekend, prototyped the needed interpreter in 10 days, and it shipped in Netscape Navigator 2.0 in September 1995. The name JavaScript settled a couple of months later.

### 6.6.3 Software

Software makes computation into a large-scale process. The basic concepts of computation become tools of computation [81]

- the **program evaluators** become the *interpreters*, whereas
- the **partial evaluators** become the *specializers*.

Partially evaluating programs for those basic components generates compilers, compiler generators, and

other varieties of familiar or less familiar program transformers.

Programming program transformers is an instance of metaprogramming. Metaprogramming can be construed as the main feature of programmable computation: that programs themselves can also be computed, and the computations that compute programs can themselves be programmed. And so on.

If programs describe computations, then software is the narrative comprised of descriptions. If programs are generated by applying some program constructor schemas (induction, recursion, parallel and sequential composition, bounded and unbounded search, etc.) on some basic operations (logical and arithmetic), then software is generated by applying software patterns and architectures to build systems from programs. If the basic operations are letters, then programs are words, software components are sentences, software systems are stories.

But some stories cannot be told in a linear narrative: "this happened, and then that happened, and then something else". You have to pop up a level, make some abstractions, and tell a story about the story. Similarly, when a computational task does not yield to applying standard program constructions, like in the case of the Ackermann function in Sec. 6.3, then you pop up a level up, make new abstractions, and metaprogram. Instead of composing operations to process data, you compose abstractions the process algorithms. The Ackermann function and the transfinite counting cannot be programmed using the recursion schema, so they get metaprogrammed. The infinities, by definition, cannot be reached by simple counting, but they are reached by iterating, i.e. by counting the counting processes. This took in the preceding chapter us to the constructions of transfinite ordinals as primordial metaprograms, and back to Cantor as the originator, along the lines mentioned in Sec. 1.6, of the finite descriptions of infinite processes that we now call programs. A particular "program" due to Cantor will play a central role at the end of the next chapter as well.

## 7 Stateful computing

---

---

20221212

20221212

## 8 Program-closed categories: computability as a property

---

---

20221212

20221212



## 9 Computability as continuity

---

---

20221212

20221212

**What??**

20221212

20221212

# Appendices

1	On categories and functors . . . . .	131
1.1	Categories . . . . .	131
1.2	Six concrete categories . . . . .	132
1.3	Functors . . . . .	134
1.4	Natural transformations . . . . .	135
1.5	Equivalence of categories . . . . .	135
2	On idempotents . . . . .	136
3	What numbers are not . . . . .	138
4	Structures vs properties . . . . .	139
5	Work . . . . .	140

## 1 On categories and functors

### 1.1 Categories

A category  $C$  is specified by

- a family (set or class) of *objects*  $|C|$ ,
- for all  $A, B \in |C|$  a set of *morphisms*  $C(A, B)$ ,
- for all  $A, B, C \in |C|$  the sequential *composition* operation  $(\circ): C(A, B) \times C(B, C) \longrightarrow C(A, C)$  satisfying

$$\begin{array}{ccc}
 C(A, B) \times C(B, C) \times C(C, D) & \xrightarrow{(\circ) \times C(C, D)} & C(A, C) \times C(C, D) \\
 \downarrow C(A, B) \times (\circ) & & \downarrow (\circ) \\
 C(A, B) \times C(B, D) & \xrightarrow{(\circ)} & C(A, D)
 \end{array} \tag{1}$$

for all  $A, B, C, D \in |C|$ , which is just the associativity  $x \circ (y \circ z) = (x \circ y) \circ z$  of all composable morphisms; and

- for all  $A \in |C|$  an *identity* morphism  $\text{id} \in C(A, A)$  satisfying

$$\begin{array}{ccc}
 C(A, B) & \xrightarrow{\text{id} \times C(A, B)} & C(A, A) \times C(A, B) \\
 \downarrow C(A, B) \times \text{id} & \searrow & \downarrow (\circ) \\
 C(A, B) \times C(B, B) & \xrightarrow{(\circ)} & C(A, B)
 \end{array} \quad (2)$$

for all  $A, B \in |C|$ , which is the unitarity  $x \circ \text{id} = x = \text{id} \circ x$ .

The conditions in diagrams (1–2) obviously generalize the more familiar monoid conditions, which are encountered in (??) for the monoid  $(\overline{\mathbb{P}}, \circ, \text{id})$ . Formally, a category  $C$  can be viewed as a  $|C| \times |C|$ -indexed monoid  $(C, \circ, \text{id})$ .

**Isomorphisms.** An isomorphism  $X \cong Y$  is a pair of morphisms in a category

$$X \begin{array}{c} \xleftarrow{f_*} \\ \xrightarrow{f^*} \end{array} Y \quad (3)$$

such that  $\text{id}_X = f_* \circ f^*$  and  $f^* \circ f_* = \text{id}_Y$ .

**Terminology.** In this book,

- *objects* are usually called **types** (because they are in a computer),
- *morphisms* are usually called **computable functions** (ditto), and
- **sequential composition** is usually drawn as a *string between boxes* (in a string diagram).

## 1.2 Six concrete categories

Categories of structures over sets are called *concrete*. The following concrete categories can even be specified without mentioning any structure on sets as objects<sup>1</sup>, with all structure carried by the morphisms:

- **Set** — sets with cartesian (total, single-valued) functions;
- **Set<sub>q</sub>** — sets with partial (single-valued) functions;
- **Set<sub>ρ</sub>** — sets with multi-valued functions (a.k.a. binary relations);
- **Set<sub>ℳ</sub>** — sets with multi-valued functions into multisets (bags);

<sup>1</sup>In this book, we usually call “types” what category theorists call “objects”; and we call “functions” what they call “morphisms”. The reason is that in monoidal computers the objects are types and the morphisms are computable functions. In this section we discuss categories where the objects are sets, and revert to the general terminology.

- $\text{Set}_{\mathcal{V}}$  — sets with linear functions;
- $\text{Set}_{\mathcal{D}}$  — sets with convex functions (a.k.a. stochastic relations).

In all cases, the objects are thus the same:

$$|\text{Set}| = |\text{Set}_{\mathcal{I}}| = |\text{Set}_{\mathcal{P}}| = |\text{Set}_{\mathcal{B}}| = |\text{Set}_{\mathcal{V}}| = |\text{Set}_{\mathcal{D}}|$$

The hom-sets are defined as follows:

$$\text{Set}(A, B) = \{f: A \longrightarrow B\} \quad (4)$$

$$\text{Set}_{\mathcal{I}}(A, B) = \{f: A_{\mathcal{I}} \longrightarrow B_{\mathcal{I}} \mid f(\mathcal{I}) = \mathcal{I}\} \quad (5)$$

$$\text{Set}_{\mathcal{P}}(A, B) = \left\{ f: \mathcal{P}A \longrightarrow \mathcal{P}B \mid \forall \mathcal{U} \subseteq \mathcal{P}A. f\left(\bigcup \mathcal{U}\right) = \bigcup_{U \in \mathcal{U}} f(U) \right\} \quad (6)$$

$$\text{Set}_{\mathcal{X}}(A, B) = \left\{ f: \mathfrak{X}A \longrightarrow \mathfrak{X}B \mid \forall \vec{u} \in \mathcal{B}A. f\left(\sum_{a \in A} u_a |a\rangle\right) = \sum_{a \in A} u_a |f(a)\rangle \right\} \quad (7)$$

where the set-constructors are defined

- $A_{\mathcal{I}} = A + \{\mathcal{I}\}$  is the partially ordered set consisting of the elements of  $A$  as incomparable and the fresh element  $\mathcal{I}$  taken as the bottom;
- $\mathcal{P}A = \{0 < 1\}^A$  is the free semilattice of subsets of  $A$  and  $\cup$  is their supremum;
- $\mathcal{B}A = \mathbb{N}_{<\infty}^A$  is the free commutative monoid of bags from  $A$  and  $\coprod$  is their sum;
- $\mathcal{V}A = \mathfrak{K}_{<\infty}^A$  is the vector space over the field  $\mathfrak{K}$  and basis  $A$ ;
- $\mathcal{D}A = \{\vec{u} \in [0, 1]_{<\infty}^A \mid \sum_{a \in A} u_a = 1\}$  is the convex hull of  $A$ .

and  $\mathfrak{X}$  denotes any of  $\mathcal{B}, \mathcal{V}, \mathcal{D}$ . The subscript “ $< \infty$ ” reminds that vectors are always finitely supported, i.e.

$$R_{<\infty}^A = \{v \in R^A \mid \#\{a \in A \mid v_a \neq 0\} < \infty\}$$

where  $R$  is a rig<sup>2</sup> and thus contains 0 and 1. In the *bra-ket-notation*,  $|a\rangle \in R_{<\infty}^A$  denotes the basis vector induced by the generator  $a \in A$ . A general vector is therefore always in the form  $\vec{u} = \sum_{a \in A} u_a |a\rangle$  where all but finitely many coefficients  $u_a$  are 0. The morphisms of  $\text{Set}_{\mathcal{P}}$ ,  $\text{Set}_{\mathcal{B}}$ , and  $\text{Set}_{\mathcal{D}}$  and  $\text{Set}_{\mathcal{V}}$  can equivalently be viewed as matrices:

$$\text{Set}_{\mathcal{P}}(A, B) = \{0 < 1\}^{B \times A} \quad (8)$$

$$\text{Set}_{\mathcal{B}}(A, B) = \mathbb{N}_{<\infty}^{B \times A} \quad (9)$$

$$\text{Set}_{\mathcal{V}}(A, B) = \mathfrak{K}_{<\infty}^{B \times A} \quad (10)$$

$$\text{Set}_{\mathcal{D}}(A, B) = \{f \in [0, 1]_{<\infty}^{B \times A} \mid \forall a \in A. \sum_{b \in B} f_{ba} = 1\} \quad (11)$$

It is instructive to show that the function compositions restrict to the matrix compositions

<sup>2</sup>A rig is a “ring without the negatives”. Distributive lattices, rings, and fields are special cases. Free semilattices happen to be distributive lattices.

- in  $\text{Set}_{\mathcal{P}}$ :

$$g \circ f = \sum_{\substack{a \in A \\ c \in C}} |c\rangle \left( \bigvee_{b \in B} g_{cb} \wedge f_{ba} \right) \langle a| \quad (12)$$

- in  $\text{Set}_{\mathcal{B}}$ ,  $\text{Set}_{\mathcal{D}}$ , and  $\text{Set}_{\mathcal{V}}$ :

$$g \circ f = \sum_{\substack{a \in A \\ c \in C}} |c\rangle \left( \sum_{b \in B} g_{cb} \cdot f_{ba} \right) \langle a| \quad (13)$$

Viewing the entries of the lattice  $\{0 < 1\}$  as truth values, (12) boils down to the relational composition:

$$(g \circ f)_{ca} \iff \exists b \in B. g_{cb} \wedge f_{ba} \quad (14)$$

### 1.3 Functors

A *functor*

$$F: \mathcal{C} \longrightarrow \mathcal{D}$$

where  $\mathcal{C}$  and  $\mathcal{D}$  are categories specified as above, is given by

- the object part  $F: |\mathcal{C}| \longrightarrow |\mathcal{D}|$  and
- the morphism part  $F_{AB}: \mathcal{C}(A, B) \longrightarrow \mathcal{D}(FA, FB)$

together satisfying the *functoriality* requirements

$$\begin{array}{ccc} & \mathcal{C}(A, A) & \\ \text{id} \nearrow & \downarrow F_{AA} & \\ 1 & & \\ \text{id} \searrow & \downarrow & \\ & \mathcal{D}(FA, FA) & \end{array} \quad \begin{array}{ccc} \mathcal{C}(A, B) \times \mathcal{C}(B, C) & \xrightarrow{(\circ)} & \mathcal{C}(A, C) \\ \downarrow F_{AB} \times F_{BC} & & \downarrow F_{AC} \\ \mathcal{D}(FA, FB) \times \mathcal{D}(FB, FC) & \xrightarrow{(\circ)} & \mathcal{D}(FA, FC) \end{array} \quad (15)$$

meaning that  $F(\text{id}) = \text{id}$  and  $F(x \circ y) = Fx \circ Fy$  for all composable morphisms  $x, y$ .

When  $\mathcal{C}$  and  $\mathcal{D}$  are monoidal categories, as specified in Sec. 1.5, then the functor  $F: \mathcal{C} \longrightarrow \mathcal{D}$  is said to be monoidal when it is given with the isomorphisms

$$FI \cong I \qquad F(A \times B) \cong FA \times FB$$

*naturally* indexed over  $A, B \in |\mathcal{C}|$ . What does it mean that they are indexed "naturally"?



## 1.4 Natural transformations

Let  $\mathcal{C}$  and  $\mathcal{D}$  be categories and  $F, G: \mathcal{C} \rightarrow \mathcal{D}$  as above. A *natural transformation*

$$\mathbf{t}: F \rightarrow G: \mathcal{C} \rightarrow \mathcal{D}$$

is defined to be a family of morphisms  $\mathbf{t}_X \in \mathcal{D}(FX, GX)$  indexed by  $X \in |\mathcal{C}|$  such that for all  $X, Y \in |\mathcal{C}|$  and all  $u \in \mathcal{C}(X, Y)$  the following squares commute

$$\begin{array}{ccc} FX & \xrightarrow{\mathbf{t}_X} & GX \\ Fu \downarrow & & \downarrow Gu \\ FY & \xrightarrow{\mathbf{t}_Y} & GY \end{array} \quad (16)$$

The natural transformation  $\mathbf{t}$  is an isomorphism when all of its components  $\mathbf{t}_X: FX \rightarrow GX$  are isomorphisms. This means that there is a natural transformation  $\mathbf{t}': G \rightarrow F$  such that  $\mathbf{t}'_X \circ \mathbf{t}_X = \text{id}_{FX}$  and  $\mathbf{t}_X \circ \mathbf{t}'_X = \text{id}_{GX}$ .

## 1.5 Equivalence of categories

An equivalence  $\mathcal{C} \simeq \mathcal{D}$  is a pair of functors

$$\mathcal{C} \begin{array}{c} \xleftarrow{F_*} \\ \simeq \\ \xrightarrow{F^*} \end{array} \mathcal{D} \quad (17)$$

together with the isomorphisms  $X \cong F_*F^*X$  and  $F^*F_*Y \cong Y$ , natural in  $X \in |\mathcal{C}|$  and  $Y \in |\mathcal{D}|$ . This generalizes the isomorphisms in Sec. 1.1 from objects of categories to categories themselves as objects of 2-categories.

### Exercises

- For the categories  $\text{Set}_{\mathcal{P}}, \text{Set}_{\mathcal{B}}, \text{Set}_{\mathcal{V}}, \text{Set}_{\mathcal{D}}$ , prove the equivalence of the morphism presentations in (6–7) and the matrix presentations (8–11). In each case, specify a bijection between the hom-sets in the first and in the second form, and verify its functoriality, i.e. that the identities and the composition are preserved.
- Show that  $\text{Set}_{\mathcal{I}}$  is equivalent to the subcategory of  $\text{Set}_{\mathcal{P}}$  in the matrix form with as morphisms those matrices that contain *at most one* 1 in each row, and all other entries 0.
- Show that  $\text{Set}$  is equivalent to the subcategory of  $\text{Set}_{\mathcal{P}}$  in the matrix form with as morphisms those matrices that contain *precisely one* 1 in each row, and all other entries 0.
- Show that  $\text{Set}$  is equivalent to the subcategory of  $\text{Set}_{\mathcal{D}}$  in the matrix form with as morphisms those matrices that take values in  $\{0, 1\} \supset [0, 1]$ .
- Spell out the monoidal structure and the data services induced by the cartesian products in  $\text{Set}_{\mathcal{I}}, \text{Set}_{\mathcal{P}}$ , and  $\text{Set}_{\mathcal{D}}$ .
- Prove that the total, single-valued functions in each of the above examples yield  $\text{Set}$  as the largest

cartesian subcategory:

$$\mathbf{Set} \simeq \mathbf{Set}_{\mathcal{C}}^{\bullet} \simeq \mathbf{Set}_{\mathcal{P}}^{\bullet} \simeq \mathbf{Set}_{\mathcal{B}}^{\bullet} \simeq \mathbf{Set}_{\mathcal{D}}^{\bullet} \simeq \mathbf{Set}_{\mathcal{V}}^{\bullet} \quad (18)$$

g. Show that an equivalence can be recognized on either of its components. Recall that a function  $f : A \rightarrow B$  has an inverse if and only if it is surjective and injective, i.e. for every  $y \in B$  there is  $x \in A$  with  $fx = y$ , and for all  $x, x' \in A$  holds  $fx = fx' \iff x = x'$ . The claim is now that a functor  $F : \mathcal{C} \rightarrow \mathcal{D}$  has an equivalence mate if and only if

- $F$  is essentially surjective: for every  $Y \in |\mathcal{D}|$  there is  $X \in |\mathcal{C}|$  with an isomorphism  $FX \cong Y$ , and
- $F$  is full and faithful: for all  $X, X' \in |\mathcal{C}|$  the functor component  $F_{XX'}$  induces a bijection  $\mathcal{D}(FX, FX') \cong \mathcal{C}(X, X')$ .

Given a functor  $F$  with these properties, construct its mate  $F' : \mathcal{D} \rightarrow \mathcal{C}$  such that  $X \cong F'FX$  and  $FF'Y \cong Y$  naturally in  $X$  and  $Y$ .

## 2 On idempotents

A *retraction* is a diagram in the form

$$R \begin{array}{c} \xleftarrow{\mathbf{q}} \\ \xrightarrow{\mathbf{i}} \end{array} X \quad (19)$$

where  $\mathbf{q} \circ \mathbf{i} = \text{id}$ . The type  $R$  is called a *retract* of the type  $A$ . Given a retraction (19), the composite  $\rho = \mathbf{i} \circ \mathbf{q}$  is the induced *idempotent*. A function  $\rho : A \rightarrow A$  is said to be idempotent when it satisfies  $\rho \circ \rho = \rho$ . The composite  $\rho = \mathbf{i} \circ \mathbf{q}$  is idempotent because  $\mathbf{q} \circ \mathbf{i} = \text{id}$ .

The other way around, given an idempotent  $\rho = \rho \circ \rho$ , a pair of functions  $\mathbf{q}, \mathbf{i}$  such that  $\mathbf{q} \circ \mathbf{i} = \text{id}$  and  $\mathbf{i} \circ \mathbf{q} = \rho$  is called a *splitting* of  $\rho$ . The equations that characterize idempotents and their splittings are summarized in the following commutative diagram.

$$\begin{array}{ccccc} X & \xrightarrow{\quad \rho \quad} & X & & \\ & \searrow \rho & \nearrow \rho & & \\ & & X & & \\ & \nearrow \mathbf{i} & \searrow \rho & & \\ R & \xleftarrow{\quad \mathbf{q} \quad} & R & \xleftarrow{\quad \mathbf{i} \quad} & R \end{array} \quad (20)$$

It is easy to see that any two idempotent splittings are isomorphic. When  $X$  is a set, then any idempotent  $\rho : X \rightarrow X$  induces the retract

$$R = \{x \in X \mid \rho(x) = x\}$$

with the role of  $R \xleftarrow{\mathbf{i}} X$  played by the inclusion and the role of  $X \xrightarrow{\mathbf{q}} R$  played by  $\rho$  itself. Retractions of sets are precisely the quotients where every equivalence class comes with a chosen element. Note that the Axiom of Choice says that every quotient can be extended to a retraction.

Any category  $\mathcal{X}$  can be embedded into the *category of idempotents*  $\mathcal{X}^\cup$ , going back to [9, IV.7.5], defined

$$\begin{aligned} |\mathcal{X}^\cup| &= \coprod_{X,Y \in |\mathcal{X}|} \{\varrho \in \mathcal{X}(X,Y) \mid \varrho \circ \varrho = \varrho\} \\ \mathcal{X}^\cup(\varrho_X, \varsigma_Y) &= \{\varphi \in \mathcal{X}(X,Y) \mid \varsigma \circ \varphi = \varphi = \varphi \circ \varrho\} \end{aligned} \quad (21)$$

The embedding  $\mathcal{X} \rightarrow \mathcal{X}^\cup$  sending every type  $X$  to  $\text{id}_X$  is the absolute completion of  $\mathcal{X}$ , under the absolute limits and colimits, which are defined as those that are preserved under all functors. It turns out that the absolute limits and colimits are just the idempotent splittings [121]. See [18, I.6.5] for a broader context. Any idempotent  $\varrho: X \rightarrow X$  in  $\mathcal{X}$  splits in  $\mathcal{X}^\cup$  by the retraction

$$\varrho \begin{array}{c} \xleftarrow{\varrho} \\ \xrightarrow{\varrho} \end{array} \text{id}_X \quad (22)$$

Any splitting  $R$  of an idempotent  $\varrho: X \rightarrow X$  that may already exist in  $\mathcal{X}$  is isomorphic in  $\mathcal{X}^\cup$  to  $\varrho$  as the added splitting (22). If  $\varsigma: Y \rightarrow Y$  is another idempotent in  $\mathcal{X}$  with a splitting  $S$ , then there is a one-to-one correspondence between

- the functions  $\bar{f} \in \mathcal{X}(R,S)$  and
- the functions  $f \in \mathcal{X}(X,Y)$  such that  $\varsigma \circ f \circ \varrho = f$ .

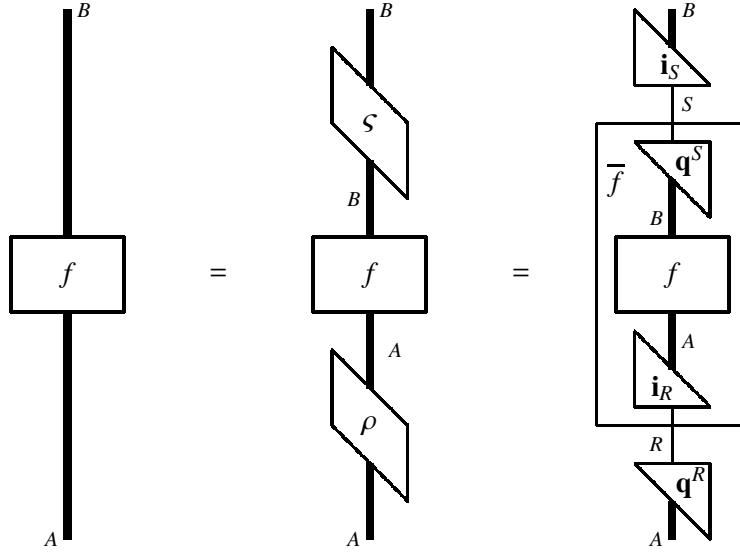


Figure 1:  $f$  induces  $\bar{f} = \mathbf{q}^S \circ f \circ \mathbf{i}_R$ , which induces  $f = \mathbf{i}_S \circ \bar{f} \circ \mathbf{q}^R$ .

Note that the condition  $\varsigma \circ f \circ \varrho = f$  is equivalent to  $f \circ \varsigma = f = \varrho \circ f$ . The fact that the underlying category  $\mathcal{C}$  of any monoidal computer is equivalent with the category of idempotents  $\mathcal{P}^\cup$  over the monoid  $\mathcal{P}$  of computable functions on its programming language  $\mathbb{P}$  is worked out in 2.6.3.2.

### 3 What numbers are not

**How much is 3?** We use 3 fingers to count 3 apples, 3 coins to pay for them, 3 units of weight to weigh them. The number 3 arises as the property shared by the sets of 3 elements. Frege formalized this by defining the number 3 as the set of all sets of 3 elements [67, pp. 1–83]. Just as the second volume of Frege’s book on the arithmetic of such numbers was in press, Bertrand Russell noticed a paradox in Frege’s system, and described it in a letter to Frege [67, pp. 124–126].

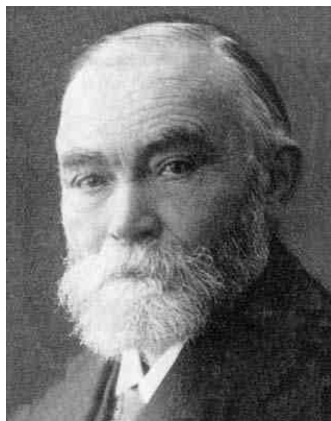


Figure 2: Gottlob Frege

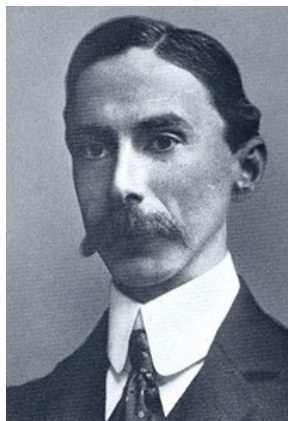


Figure 3: Bertrand Russell



Figure 4: Neumann János

**Russell’s paradox.** Call a set *ordinary* if it does not contain itself as an element, and *extraordinary* if it does contain itself as an element. E.g. the set of all cats is ordinary because it is not a cat; whereas the set of all sentences defined in less than 20 words is extraordinary because we just defined it in less than 20 words. Let  $O$  be the set of all ordinary sets, and  $\mathcal{E}$  the set of all extraordinary sets, i.e.

$$\begin{aligned} O &= \{x \mid x \notin x\} \\ \mathcal{E} &= \{x \mid x \in x\} \end{aligned}$$

The paradox arises from asking whether the sets  $O$  and  $\mathcal{E}$  themselves are ordinary or extraordinary. For each of them, both possibilities lead to a contradiction. Let us consider  $O$ .

- If  $O$  is an ordinary set, then  $O \in O$ , since  $O$  contains all ordinary sets. But then  $O$  contains itself as an element, i.e. it is extraordinary.
- If  $O$  is extraordinary, then  $O \notin O$ , because  $O$  only contains ordinary sets. But then  $O$  does not contain itself as an element, i.e. it is ordinary.

Both possibilities, that the set  $O$  contains itself as an element, and that it does not, lead to a contradiction. Hence the paradox. The underlying logical schema, known as the *diagonal argument*, was used earlier by Georg Cantor to show that the real numbers are not countable, and later in Gödel’s Incompleteness Theorem, presented in Sec. 5.2, and Turing’s proof of undecidability of the Entscheidungsproblem, presented in Sec. 5.3.<sup>3</sup>

<sup>3</sup>Cantor also applied his diagonal argument to the element relation to construct his “inconsistent” sets [115], but the paradox that he noticed in the end got attributed to Russell, who presented it to Frege, and clarified its impact to Dedekind, and perhaps even to Cantor. Cantor’s “inconsistent” sets, as a solution of the paradoxes, were clarified still much later, renamed to classes, and attributed to von Neumann.

**Comprehension.** Seeing his life's work shattered by Russell's observation, Frege responded with remarkable honesty. He admitted that there was a problem, and tried to solve it. The solution that gradually filtered through set theory is that collecting all entities with a certain property does not always yield a set. Frege's original proposal was that every property, formalized as a predicate  $P(x)$ , induces the set  $\{x|P(x)\}$  of all  $x$  that satisfy  $P$ . That is how we form *classes* in the Hilbert-Bernays set theory. Russell's argument was that this leads to a paradox if the predicate  $P(x)$  is instantiated to  $x \in x$ . To avoid this paradox, the sets are built inductively, by the powerset constructor, and the sets of elements satisfying a certain property, presented as a predicate, can only be carved out of previously constructed sets, using the axiom of *comprehension*. A predicate  $P(x)$  can thus be comprehended within any given set  $A$  as its subset  $\{x \in A \mid P(x)\}$ . A different solution, proposed by Bertrand Russell, was to replace sets with types [140]. A type-theoretic view of the set-theoretic comprehension constructor expressed as a categorical adjunction is described in [99]. Whether they are comprehended within the set of finite sets, or built as a type, numbers arise from counting, as presented in Sec. 4.2.

**From numbers to life.** The static view of numbers as things or properties, espoused by philosophers from the Pythagoreans to Frege, was superseded by the inductive view in [174], authored by the 17-year old Hungarian wunderkind Neumann János, Fig. 4. His life was the XX century version of the tragedy of Dr Faustus, also presented as the comedy of Dr Strangelove. In the US, János became John von Neumann, whom we saw in Fig. 2.15. He was, with Einstein, one of the first members of the Institute of Advanced Studies, where he contributed to the paradigm shifts in logic, computation, quantum physics, game theory. Having seen nazis in action early on, he gradually refocused from logic to the nuclear weapons and died at the age of 53, probably from exposure to radiation at the nuclear tests that he insisted on attending. On his deathbed, he worked on a series of lectures proving that life, as the capability of self-reproduction, arises from computation. The lectures were never delivered, but appeared posthumously as [114].

## 4 Structures vs properties

If a poset (a *partially ordered set*) has the suprema of all of its subsets, then they are uniquely determined by the order, and their existence is a *property* of the poset. Completeness is thus an intrinsic property. On the other hand, a poset supports many different topologies, and each of them is a *structure* that can be imposed on it [55]. A familiar example is the closed interval  $[0, 1]$ , which can be viewed with the usual open interval topology, or with the Cantor space topology, or with the cofinal topology, or with any of the other neighborhood structures that have been invented or discovered in its applications. The topologies are thus extrinsic structures of sets and posets. For another familiar example, consider the size of a set, and the group structures on it. Whichever way we count a set of 8 elements, we come with the same number. The different ways in which we may count it are different bijections, and the number of elements is thus unique up to bijections. It is a property of the set. On the other hand, a set of 8 elements may carry 5 different, non-isomorphic group structures. The size of a set is thus its property, whereas a group is a structure that may be added to it.

## 5 Work

### Work 1.6: Types and functions

- a. i) The function  $\langle (+), (\cdot) \rangle$  can be constructed as a parallel composition of  $(+)$  and  $(\cdot)$ , but we need to feed them the inputs. To feed two functions, we need two copies of  $m$  and two copies of  $n$ . For this we run two copies of the copying data service  $\Delta : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ , one to get  $\Delta m = \langle m, m \rangle$  and one to get  $\Delta n = \langle n, n \rangle$ . The two copies can be run in parallel, giving  $\langle m, m, n, n \rangle$ . To get the inputs for  $(+)$  and  $(\cdot)$  in the form  $\langle m, n, m, n \rangle$ , we swap the middle pair in  $\langle m, m, n, n \rangle$ . The data flow is thus

$$\mathbb{N} \times \mathbb{N} \xrightarrow{\Delta \times \Delta} \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \xrightarrow{\mathbb{N} \times \zeta \times \mathbb{N}} \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \xrightarrow{(+)\times(\cdot)} \mathbb{N} \times \mathbb{N}$$

$$\langle m, n \rangle \longmapsto \langle m, m, n, n \rangle \longmapsto \langle m, n, m, n \rangle \longmapsto \langle m+n, m \cdot n \rangle$$

The algebraic form of this function is thus

$$\langle (+), (\cdot) \rangle = ((+)\times(\cdot)) \circ (\mathbb{N} \times \zeta \times \mathbb{N}) \circ (\Delta \times \Delta)$$

and the corresponding string diagram is in Fig. 5, top left.

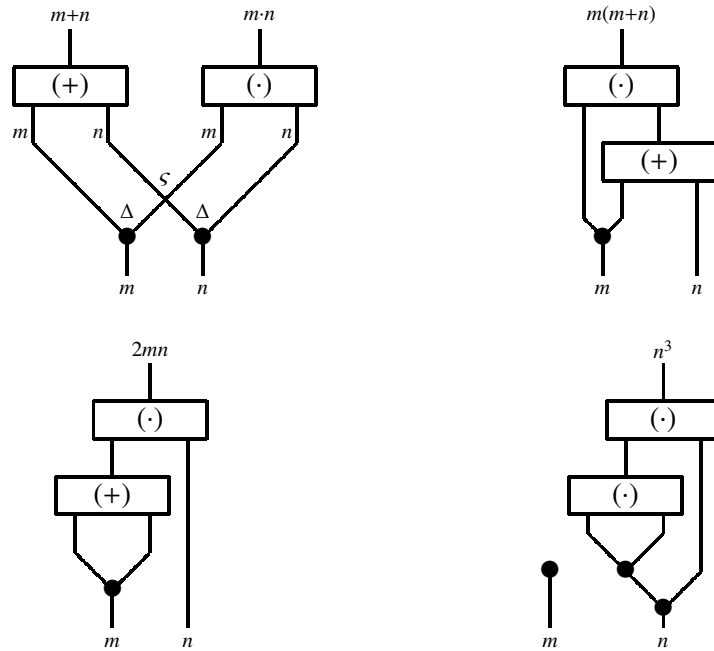


Figure 5: String diagrams for exercise 1.6.a.

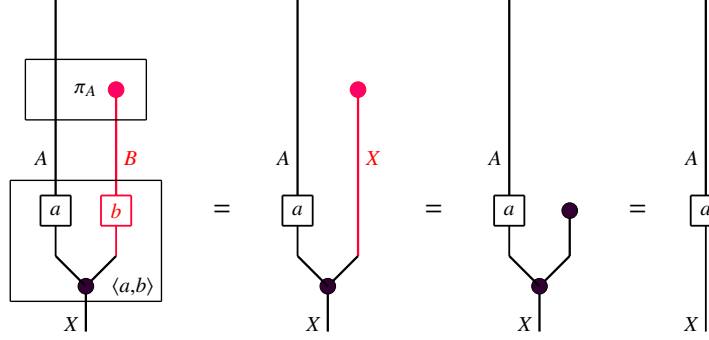
- ii–iv) The string diagrams are in Fig. 5.

b. The bottom-up data flow through the diagram is as follows:

$$\begin{aligned} \langle x, y \rangle &\mapsto \langle y, x \rangle \mapsto \langle y, y, x, x, x \rangle \mapsto \langle y, x, y, x, x \rangle \mapsto \langle y, 0, x, y, x, x \rangle \mapsto \\ &\mapsto \langle f(y, 0, x), y, x, g(x) \rangle \mapsto h(f(y, 0, x), y, x, g(x)) \mapsto \pi_E(h(f(y, 0, x), y, x, g(x))) \end{aligned}$$

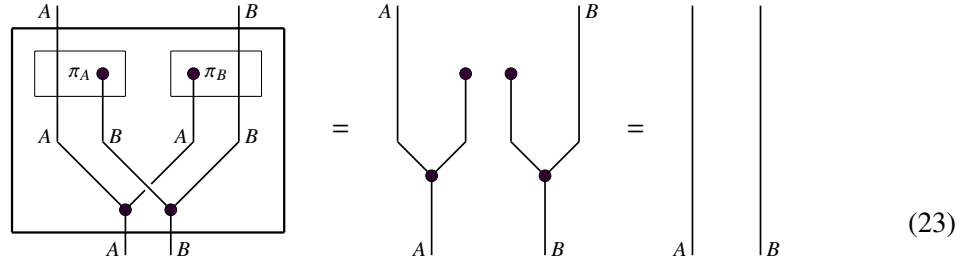
Fig. 1.9 thus displays the function that maps the inputs  $\langle x, y \rangle$  to the output  $\pi_E(h(f(y, 0, x), y, x, g(x)))$ .

c. i) The first equation holds by

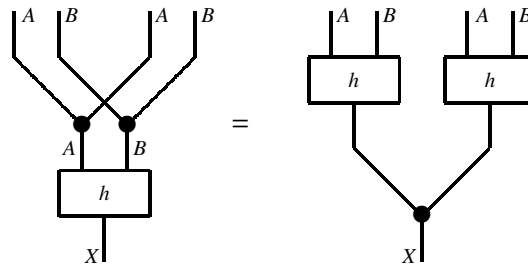


The assumption that  $b$  is total is needed at the first step, and marked in red. The converse, that this equation implies that  $b$  is total, is obtained by setting  $a = \uparrow_X$ .

ii) Towards the second equation, first note that  $\langle \pi_A, \pi_B \rangle = \text{id}_{A \times B}$  always holds:



The assumption that  $h$  is single-valued on the other hand means



Postcomposing both sides with  $\pi_A \times \pi_B$  and using (23) leaves  $h$  on the left and  $\langle \pi_A h, \pi_B h \rangle$  on the right.

iii) The third equation is analogous to the first.

### Work 2.6.1: How many programs?

a. We show that  $\top = \perp$  implies  $C \simeq 1$ .

i) An equational proof would be:

$$\varphi = \{\top\}(\varphi \times \gamma) = \{\perp\} \circ (\varphi \times \gamma) = \gamma \quad (24)$$

Draw the string diagrams.

ii) Any  $f : A \rightarrow B$  induces

$$\bar{f} = \left( \mathbb{P} \xrightarrow{\{\}} A \xrightarrow{f} B \xrightarrow{\ulcorner - \urcorner} \bullet \mathbb{P} \right) \quad (25)$$

For any  $f, g : A \rightarrow B$ , (24) implies  $\bar{f} = \bar{g}$ . But then

$$f = \left( A \xrightarrow{\ulcorner - \urcorner} \bullet \mathbb{P} \xrightarrow{\bar{f}} \mathbb{P} \xrightarrow{\{\}} B \right) = \left( A \xrightarrow{\ulcorner - \urcorner} \bullet \mathbb{P} \xrightarrow{\bar{g}} \mathbb{P} \xrightarrow{\{\}} B \right) = g \quad (26)$$

iii) For arbitrary types  $C, D$ , define

$$h = \left( C \xrightarrow{\ulcorner - \urcorner} \bullet \mathbb{P} \xrightarrow{\{\}} D \right) \quad k = \left( D \xrightarrow{\ulcorner - \urcorner} \bullet \mathbb{P} \xrightarrow{\{\}} C \right)$$

Applying (26) to  $k \circ h, \text{id} : C \rightarrow C$  gives  $k \circ h = \text{id}$ . Applying it to  $h \circ k, \text{id} : D \rightarrow D$  gives  $h \circ k = \text{id}$ . The functions  $h$  and  $k$  thus form an isomorphism  $C \cong D$ .

iv) Taking  $D$  in answer (iii) to be the unit type  $I$  gives an isomorphism  $C \cong I$  for any type  $C$ . Answer (ii) implies that the only function on  $I$  is the identity, and that for every type  $C$  there is precisely one function  $C \rightarrow I$  and  $I \rightarrow C$ . These two functions must be the isomorphism  $C \cong I$ , and the only function on  $C$  must also be the identity.

b. We show that  $\top \neq \perp$  implies that for any  $f \in C(A, B)$  there are infinitely many  $F \in C^\bullet(I, \mathbb{P})$  with  $f = \{F\}$ .

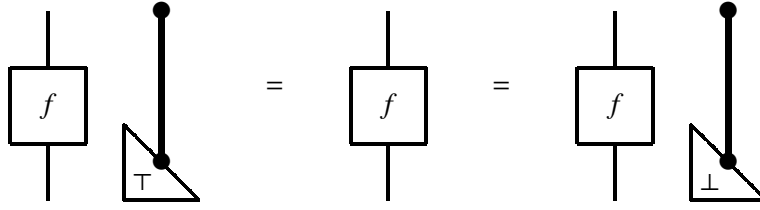
i) If  $\top \neq \perp$  in  $\mathbb{P}$ , then the pairs  $\langle \top, \top \rangle, \langle \top, \perp \rangle, \langle \perp, \top \rangle, \langle \perp, \perp \rangle$  in  $\mathbb{P}^2$  are all different. But since  $[-, -] : \mathbb{P}^2 \rightarrow \bullet \mathbb{P}$ , defined in (2.11), is an injection, the programs  $[\top, \top], [\top, \perp], [\perp, \top], [\perp, \perp]$  in  $\mathbb{P}$  are all different, and hence there are at least  $4 = 2^2$  different elements in  $\mathbb{P}$ . Generalizing to  $n$ , the assumption  $\top \neq \perp$  implies that there are  $2^n$  different  $n$ -tuples of  $\top$ s and  $\perp$ s in  $\mathbb{P}^n$ . But the encoding  $[-, -, \dots, -] : \mathbb{P}^n \rightarrow \bullet \mathbb{P}$  of  $n$ -tuples of programs as programs is also an injection, since  $\mathbb{P}^n$ , like any type, is a retract of  $\mathbb{P}$ . Hence the claim that there must be at least  $2^n$  different programs in  $\mathbb{P}$ . Since the above arguments hold for any  $n$ , there is no finite bound, which means that we have actually proved that  $\mathbb{P}$  must be infinite.

ii) For an arbitrary function  $f \in C(A, B)$ , consider the equal functions

$$\left( A \xrightarrow{f \times \top} B \times \mathbb{P} \xrightarrow{\pi} B \right) = \left( A \xrightarrow{f} B \right) = \left( A \xrightarrow{f \times \perp} B \times \mathbb{P} \xrightarrow{\pi} B \right)$$

displayed in Fig. 6. If  $F$  is a program for  $f$ , and if  $\ulcorner \top \urcorner, \ulcorner \perp \urcorner, \ulcorner \pi \urcorner$  are the encodings for the



Figure 6: Different compositions of  $f$ 

functions  $\top$ ,  $\perp$  and  $\pi$ , with  $\{\ulcorner \top \urcorner\} = \top$ ,  $\{\ulcorner \perp \urcorner\} = \perp$  and  $\{\ulcorner \pi \urcorner\} = \pi$ , then we have

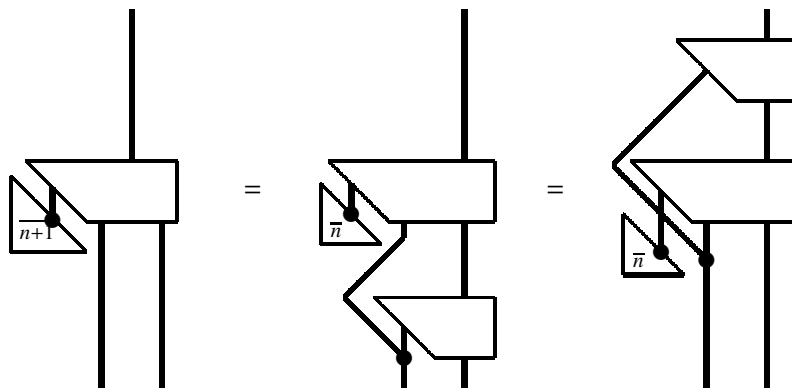
$$\begin{aligned} \{(F \parallel \ulcorner \top \urcorner) ; \ulcorner \pi \urcorner\} &= \{F\} = \{(F \parallel \ulcorner \top \urcorner) ; \ulcorner \pi \urcorner\} && \text{but} \\ \{(F \parallel \ulcorner \top \urcorner) ; \ulcorner \pi \urcorner\} &\neq F \neq \{(F \parallel \ulcorner \top \urcorner) ; \ulcorner \pi \urcorner\} \end{aligned}$$

where  $(- ; -)$  and  $(- \parallel -)$  are the program composition operations from Sec. 2.2.3. The inequalities are satisfied by all programs  $F$  as soon as these operations are injective. It can be proved in a straightforward but lengthy reasoning that any choice of composition encodings can be made injective. A reader willing to skip ahead can find a simple trick that does the job in Sec. ???. Composing  $f$  in parallel with each of the  $2^n$  different  $n$ -tuples of  $\top$  and  $\perp$ , and then projecting away the outputs of these  $n$ -tuples, as above, yields  $2^n$  different programs encoding the same function  $f$ . Since this can be done for any  $n$ , there must be infinitely many.

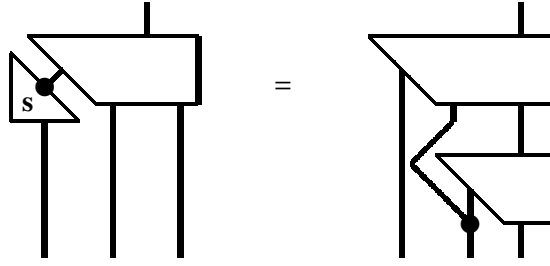
### Work 2.6.2: Church numerals

The arithmetic operations on the Church numerals can be defined as follows.

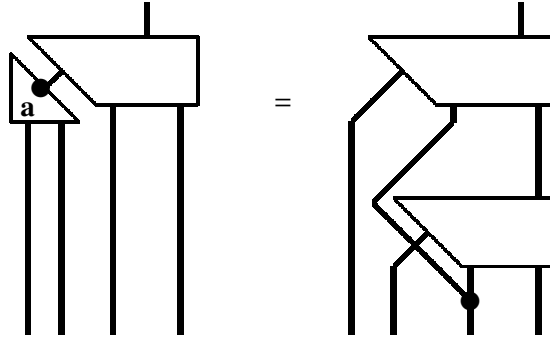
- a. The successor can be defined in two ways:



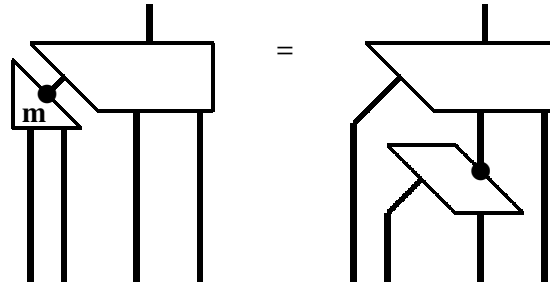
The program should thus be:



b. The Church-Rosser addition:



c. The Church-Rosser multiplication:

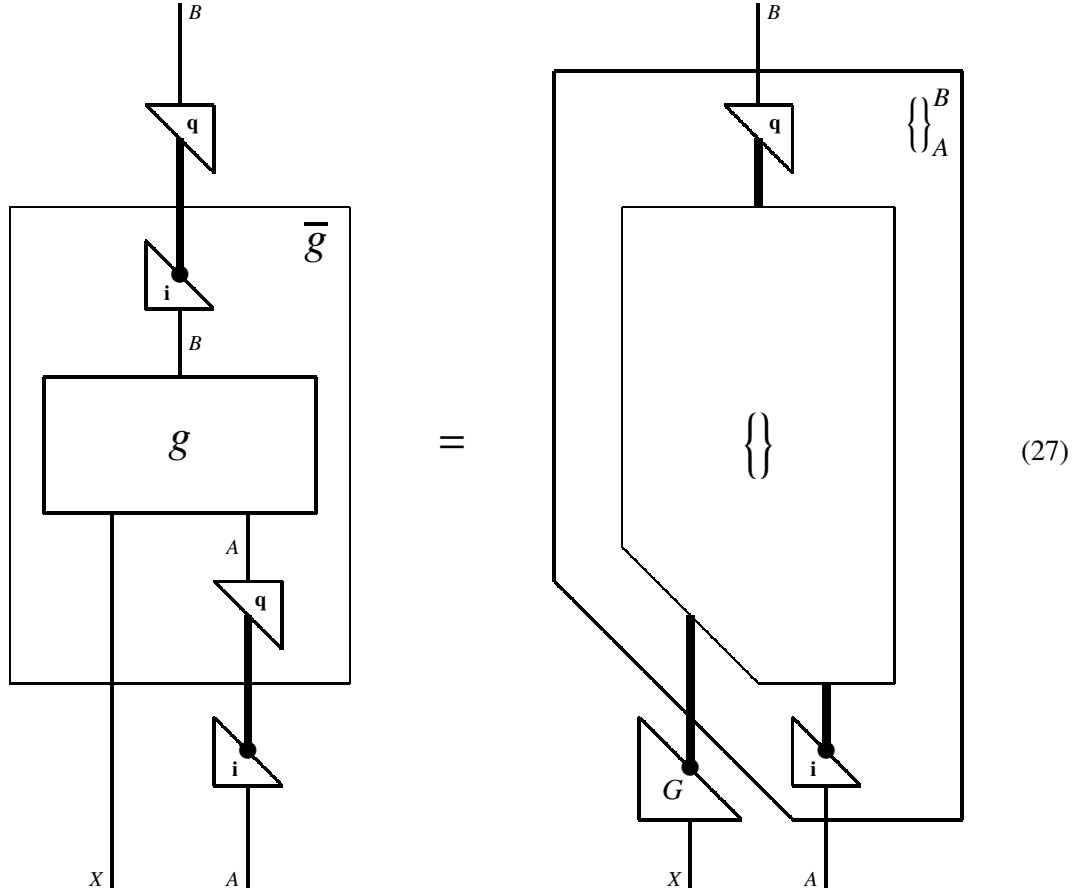


What is the idea behind this solution? We need  $\{\mathbf{m}(\bar{k}, \bar{\ell})\}(x, y) = \{x\}^{k \cdot \ell} y$ . Given that  $\{\bar{k}\}(z, y) = \{z\}^k y$  holds by definition of  $\bar{k}$ , what should be substituted for  $z$  so that  $\{z\} y = \{x\}^\ell (y)$ ?

### Work 2.6.3.1: Universal program evaluator

- a. We show that the function defined in (2.20), satisfies the program evaluator requirement from Sec. 2.2.1. Given an arbitrary function  $g : X \times A \rightarrow B$ , the task is to find a family of programs  $G : X \rightarrow \bullet \mathbb{P}$

such that  $\mathbf{q}^B(\{G_x\}_{\mathbb{P}}^{\mathbb{P}}(\mathbf{i}_A(a))) = g_x(a)$ . The construction is displayed in the following diagram.



We first define

$$\bar{g} = \left( X \times \mathbb{P} \xrightarrow{X \times \mathbf{q}} X \times A \xrightarrow{g} B \xrightarrow{\mathbf{i}} \bullet \mathbb{P} \right)$$

Since  $\{ \} : \mathbb{P} \times \mathbb{P} \longrightarrow \mathbb{P}$  is a program evaluator, there is  $G : X \longrightarrow \bullet \mathbb{P}$  such that  $\{G\} = \bar{g}$ . But then

$$\begin{aligned} g &= \left( X \times A \xrightarrow{X \times \mathbf{i}} X \times \mathbb{P} \xrightarrow{\bar{g}} \mathbb{P} \xrightarrow{\mathbf{q}} B \right) \\ &= \left( X \times A \xrightarrow{X \times \mathbf{i}} X \times \mathbb{P} \xrightarrow{G \times \mathbb{P}} \mathbb{P} \times \mathbb{P} \xrightarrow{\{ \}} \mathbb{P} \xrightarrow{\mathbf{q}} B \right) \\ &= \left( X \times A \xrightarrow{G \times A} \mathbb{P} \times A \xrightarrow{\{ \}_A^B} B \right) \end{aligned}$$

- b. The program transformer  $\nu_A^B : \mathbb{P} \longrightarrow \bullet \mathbb{P}$  is defined to be the  $\mathbb{P}$ -indexed program for the computation  $\mathbf{i}_B \circ \{ \}_A^B \circ \mathbf{q}^A$ , as displayed in Fig. 7. If  $f = \{F\}_A^B$  then  $\mathbf{i}_B \circ f \circ \mathbf{q}^A = \mathbf{i}_B \circ \{F\}_A^B \circ \mathbf{q}^A = \{ \nu_A^B F \}_{\mathbb{P}}^{\mathbb{P}}$  and hence  $f = \mathbf{q}^B \circ \{ \nu_A^B F \}_{\mathbb{P}}^{\mathbb{P}} \circ \mathbf{i}_A$ .

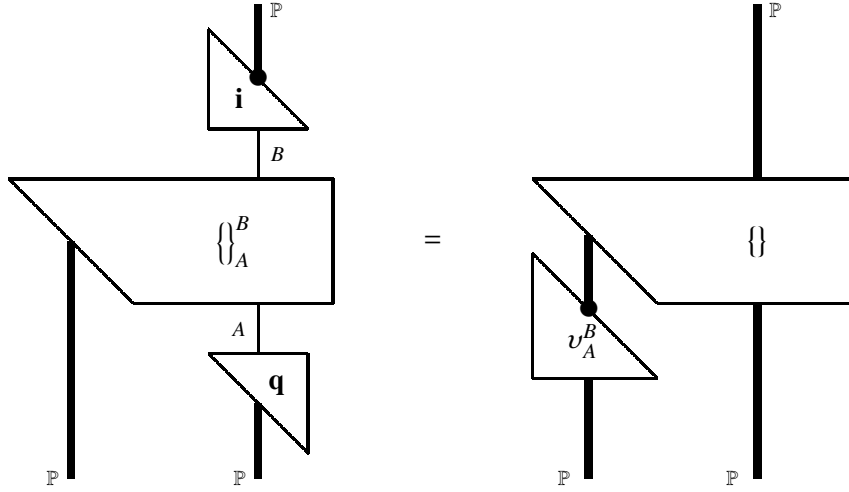


Figure 7: If  $\mathbf{i}_B \circ \{ \}_A^B \circ \mathbf{q}^A = \{ \nu_A^B \}_P^P$  then  $\{ F \}_A^B = \mathbf{q}^B \circ \{ \nu_A^B F \}_P^P \circ \mathbf{i}_A$

### Work 2.6.3.2: Idempotent completion of submonoid

- a. Every idempotent  $\varrho$  of  $\mathcal{P}$  becomes in  $\mathcal{P}^\cup$  not only an object, but also no less than 4 different morphisms:

$$\varrho \circ \text{id} \xleftarrow{\varrho} \varrho \xrightarrow{\varrho} \varrho \circ \varrho$$

Note that the  $\varrho$ -loop on the right is the identity on  $\varrho$  as an object of  $\mathcal{P}^\cup$ , whereas the  $\varrho$ -loop on the left is an idempotent on  $\text{id}$  as an object of  $\mathcal{P}^\cup$ . The remaining two  $\varrho$ -morphisms between the objects  $\text{id}$  and  $\varrho$  display  $\varrho$  as a retract of  $\text{id}$  in  $\mathcal{P}^\cup$ . Every object of  $\mathcal{P}^\cup$  is thus a retract of  $\text{id}$ . If  $\varsigma$  is another idempotent in  $\mathcal{P}$ , then an arbitrary  $\varphi \in \mathcal{P}$  satisfying  $\varphi = \varsigma \circ \varphi \circ \varrho$  is a morphism  $\varphi \in \mathcal{P}^\cup(\varrho, \varsigma)$  making the upper diagram in the following derivation commute.

$$\begin{array}{ccccc} & & \varrho & & \\ & \text{id} & \xrightarrow{\varrho} & \varrho & \xrightarrow{\varrho} & \text{id} \\ & \downarrow \varphi & & \downarrow \varphi & & \downarrow \varphi \\ & \text{id} & \xrightarrow{\varsigma} & \varsigma & \xrightarrow{\varsigma} & \text{id} \\ & & \text{---} \varrho & & \\ & & F\varrho & & \\ & F\mathbb{P} & \xrightarrow{F\varrho} & F^\# \varrho & \xrightarrow{F\varrho} & F\mathbb{P} \\ & \downarrow F\varphi & & \downarrow F^\# \varphi & & \downarrow F\varphi \\ & F\mathbb{P} & \xrightarrow{F\varsigma} & F^\# \varsigma & \xrightarrow{F\varsigma} & F\mathbb{P} \end{array}$$

The lower diagram defines the claimed extension  $F^\#: \mathcal{P}^\cup \longrightarrow C$  of  $F: \mathcal{P} \longrightarrow C$ . The  $F^\#$ -images of the idempotents  $\varrho$  and  $\varsigma$  as objects of  $\mathcal{P}^\cup$  are determined by the splittings of the idempotents  $F\varrho$  and  $F\varsigma$  in  $C$ , which exist by the assumption about  $C$ . The  $F^\#$ -image of  $\varphi \in \mathcal{P}^\cup(\varrho, \varsigma)$  is induced by the splittings and the fact that  $F\varphi = F\varsigma \circ F\varphi \circ F\varrho$ .

- b. The assumption that every  $A \in |C|$  is a retract of  $\mathbb{P}$  means that there is an idempotent

$$\rho_A = \left( \mathbb{P} \xrightarrow{\mathbf{q}^A} A \xrightarrow{\mathbf{i}_A} \mathbb{P} \right) \quad (28)$$

The object part of the claimed functor  $\rho: C \longrightarrow \mathcal{P}^\cup$  is a choice of such idempotents. The arrow part and the fact that it establishes a bijection  $C(A, B) \cong \mathcal{P}^\cup(\rho_A, \rho_B)$  can be obtained from the following diagram

$$\begin{array}{ccccc} & & \rho_A & & \\ & & \curvearrowright & & \\ \mathbb{P} & \xrightarrow{\mathbf{q}} & A & \xrightarrow{\mathbf{i}} & \mathbb{P} \\ \downarrow \rho_f & & \downarrow f & & \downarrow \rho_f \\ \mathbb{P} & \xleftarrow{\mathbf{i}} & B & \xleftarrow{\mathbf{q}} & \mathbb{P} \\ & & \rho_B & & \end{array} \quad (29)$$

defining  $\rho_f = \mathbf{i}_B \circ f \circ \mathbf{q}^A$  which is inverted by  $\mathbf{q}^B \circ \rho_f \circ \mathbf{i}_A = f$ .

It is easy to see that the  $\rho$  is completely determined by a choice of retractions (28). But what if  $A$  is a retract of  $\mathbb{P}$  in several ways, and induces several different idempotents on  $\mathbb{P}$  which all split on  $A$ ? We prove that all such idempotents are isomorphic in  $\mathcal{P}^\cup$ , and thus determine isomorphic functors  $\rho: C \longrightarrow \mathcal{P}^\cup$ . Suppose that we are given some retractions  $\rho_A = \mathbf{i} \circ \mathbf{q}$  and  $\rho'_A = \mathbf{i}' \circ \mathbf{q}'$  as on the following diagram:

$$\begin{array}{ccccc} & & v & & \\ & & \curvearrowright & & \\ \rho_A \circlearrowleft \mathbb{P} & \xleftarrow{\mathbf{i}} & A & \xleftarrow{\mathbf{q}'} & \mathbb{P} \circlearrowright \rho'_A \\ & \xrightarrow{\mathbf{q}} & & \xrightarrow{\mathbf{i}'} & \\ & & u & & \end{array} \quad (30)$$

Then  $u = \mathbf{i}' \circ \mathbf{q}$  and  $v = \mathbf{i} \circ \mathbf{q}'$  form an isomorphism  $\rho_A \cong \rho'_A$  in  $\mathcal{P}^\cup$ , since  $u = \rho'_A \circ u \circ \rho_A$  justifies  $u \in \mathcal{P}^\cup(\rho_A, \rho'_A)$ ,  $v = \rho_A \circ v \circ \rho'_A$  justifies  $v \in \mathcal{P}^\cup(\rho'_A, \rho_A)$  and the equations  $u \circ v = \rho_A$  and  $v \circ u = \rho'_A$  make them into an isomorphism, since the  $\mathcal{P}$ -idempotent  $\rho_A$  is the identity on the  $\mathcal{P}^\cup$ -object  $\rho_A$ , whereas  $\rho'_A$  is the identity on  $\rho'_A$ .

- c. Since  $\rho: C \longrightarrow \mathcal{P}^\cup$  maps  $\mathcal{P}$  as a subcategory of  $C$  into  $\mathcal{P}$  as a subcategory of  $\mathcal{P}^\cup$ , its mate will have to preserve  $\mathcal{P}$  as well, and thus map  $\text{id} \in |\mathcal{P}^\cup|$  into  $\mathbb{P} \in |C|$ , and every  $\varphi \in \mathcal{P}$  as  $\varphi \in \mathcal{P}^\cup(\text{id}, \text{id})$  back into itself as  $\varphi \in C(\mathbb{P}, \mathbb{P})$ . Mapping the other objects  $\varrho \in |\mathcal{P}^\cup|$  into  $\widehat{e}|C|$  boils down to choosing the splittings of the idempotents  $\varrho \in C(\mathbb{P}, \mathbb{P})$ , which exist by assumption. The construction is an instance of exercise a above, where the functor  $F$  is taken to be the inclusion  $\mathcal{P} \hookrightarrow C$ . Note that it boils down to choosing particular splittings for the idempotents on  $\mathbb{P}$ . The equivalence claim is straightforward.

### Work 2.6.3.3: Relating products and pairing

- c.ii) For convenience, write  $\widehat{\varrho}$  as  $A$  and  $\widehat{\varsigma}$  as  $B$ , and suppose that the functor  $\rho$  is chosen so that  $\varrho = \rho_A$  and  $\varsigma = \rho_B$  (which is always possible with no loss of generality). Then the definition of  $\otimes$  gives  $\varrho \otimes \varsigma = \rho_{A \times B}$ . The convolution  $\varrho \star \varsigma = \rho_A \star \rho_B$  is shown in Fig. 8 on the left. Its splitting on  $A \times B$

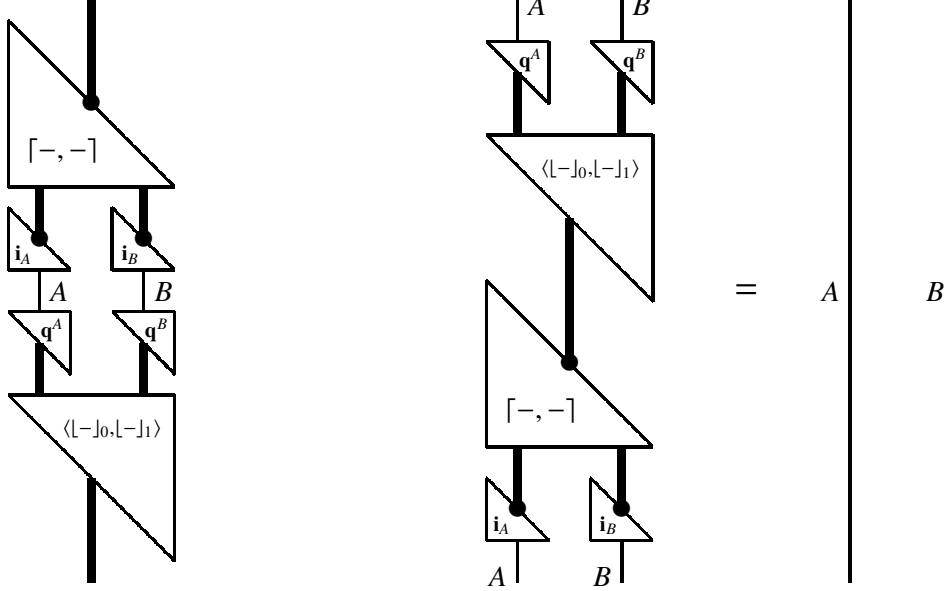


Figure 8: The convolution of  $\rho_A = \mathbf{i}_A \circ \mathbf{q}^A$  and  $\rho_B = \mathbf{i}_B \circ \mathbf{q}^B$ , and its splitting

is shown on the right. On the other hand,  $\varrho \otimes \varsigma = \rho_{A \times B}$  also splits on  $A \times B$ , by the definition of  $\rho$ . So the idempotents  $\varrho \star \varsigma$  and  $\varrho \otimes \varsigma$  have the same splitting. The isomorphism  $\varrho \star \varsigma \cong \varrho \otimes \varsigma$  in  $\mathcal{P}^\cup$  is then constructed like in work 2.6.3.2.b., using a diagram like (30).

- d. The algebraic proof that a monoidal structure  $(\mathcal{P}, \star, \rho_I)$  over a monoid  $(\mathcal{P}, \circ, \text{id}_{\mathbb{P}})$  must collapse to the same monoid is known as the *Eckmann-Hilton argument* [45]. It uses the unit laws and the middle-two-interchange law, discussed in Sec. 1.3.2, here in the form  $(g \circ f) \star (s \circ t) = (g \star s) \circ (f \star t)$ , to derive

- $\rho_I = \rho_I \star \rho_I = (\rho_I \circ \text{id}_{\mathbb{P}}) \star (\text{id}_{\mathbb{P}} \circ \rho_I) = (\rho_I \star \text{id}_{\mathbb{P}}) \circ (\text{id}_{\mathbb{P}} \star \rho_I) = \text{id}_{\mathbb{P}} \circ \text{id}_{\mathbb{P}} = \text{id}_{\mathbb{P}}$  and then
- $\varphi \star \psi = (\varphi \circ \text{id}_{\mathbb{P}}) \star (\text{id}_{\mathbb{P}} \circ \psi) = (\varphi \star \text{id}_{\mathbb{P}}) \circ (\text{id}_{\mathbb{P}} \star \psi) \stackrel{*}{=} (\varphi \star \rho_I) \circ (\rho_I \star \psi) = \varphi \circ \psi$

A variation on the second argument also shows that the binary operation is commutative. Although we started from monoids, the associativity was never used, and can also be derived. The isomorphism  $I \cong \mathbb{P}$  follows from  $\rho_I = \text{id}_{\mathbb{P}}$ , since the splitting of an invertible idempotent must be invertible.

### Work 2.6.3.4: Reducing $\{\}_{A}^B$ to $\{\}_{\mathbb{P}}^{\mathbb{P}}$

- b.i) The equation  $\mathbf{i}_{A \times B} = \mathbf{i}_{\mathbb{P} \times \mathbb{P}} \circ (\mathbf{i}_A \times \mathbf{i}_B)$  follows from Fig. 10, by evaluating both sides on a program that encodes  $\text{id}_{\mathbb{P}}$ . Instantiating to  $A = B = \mathbb{P}$  gives  $\mathbf{i}_{A \times B} = \mathbf{i}_{\mathbb{P} \times \mathbb{P}} \circ (\mathbf{i}_{\mathbb{P}} \times \mathbf{i}_{\mathbb{P}})$ . Postcomposing both sides with  $\mathbf{q}^{\mathbb{P} \times \mathbb{P}}$  and using  $\mathbf{q} \circ \mathbf{i} = \text{id}$  proves that  $\mathbf{i}_{\mathbb{P}}$  is the identity. It follows that  $\mathbf{q}^{\mathbb{P}}$  is also identity

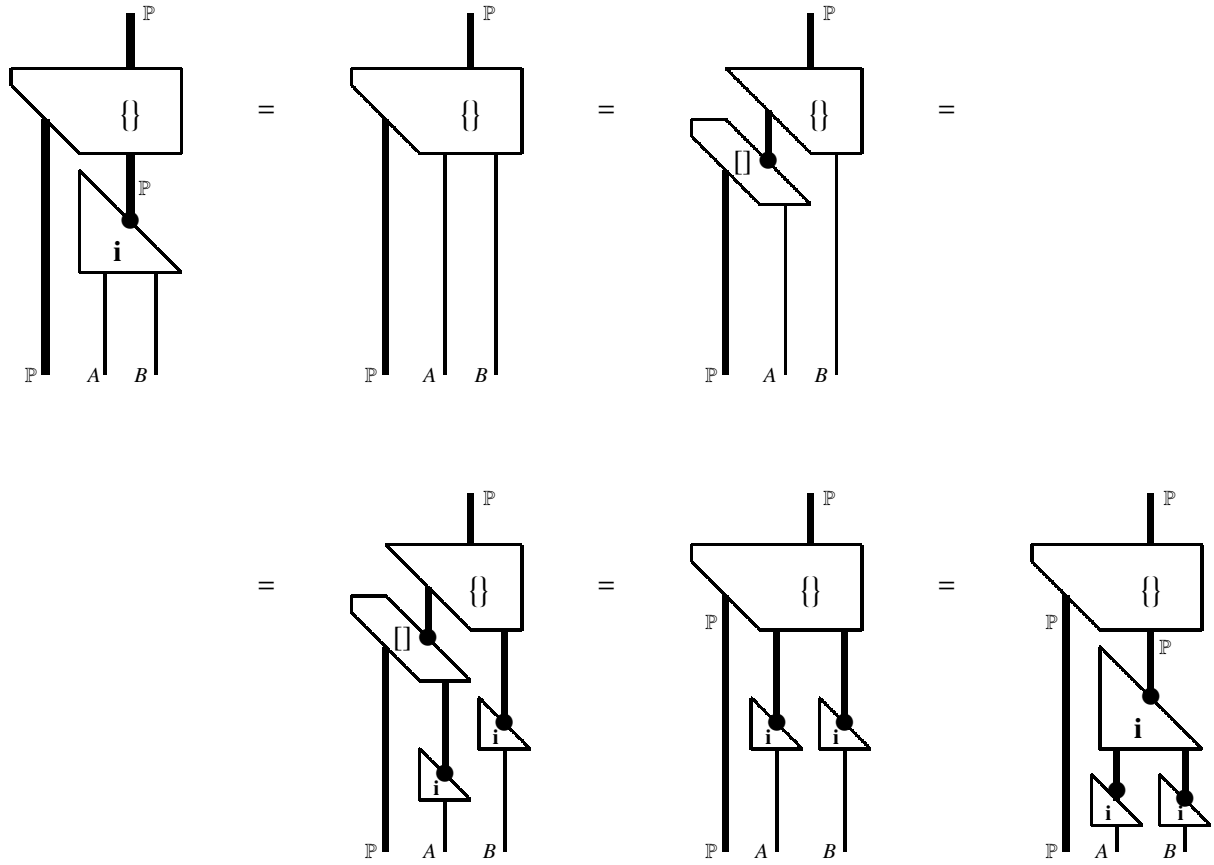


Figure 9: If  $\{F\}_{A \times B}(x, y) = \{F\}_{\mathbb{P} \times \mathbb{P}}[x, y]$  then  $\mathbf{i}_{A \times B} = \mathbf{i}_{\mathbb{P} \times \mathbb{P}} \circ (\mathbf{i}_A \times \mathbf{i}_B)$

and hence  $\mathbf{q}^{\mathbb{P}} \neq \{\}_{I}^{\mathbb{P}}$ , or else  $\{F\}_A^{\mathbb{P}} a = [F]_A a$ , as displayed in Fig. 10.

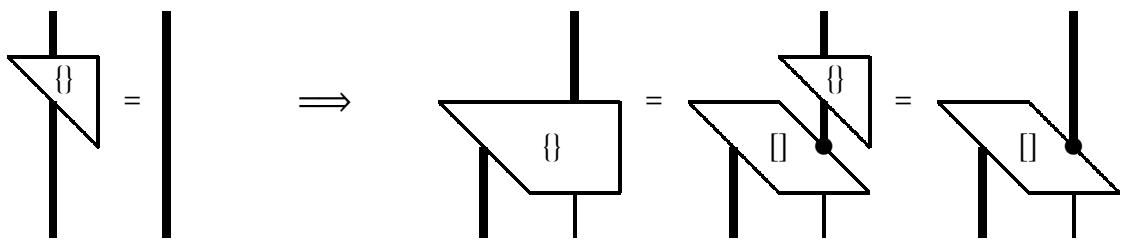
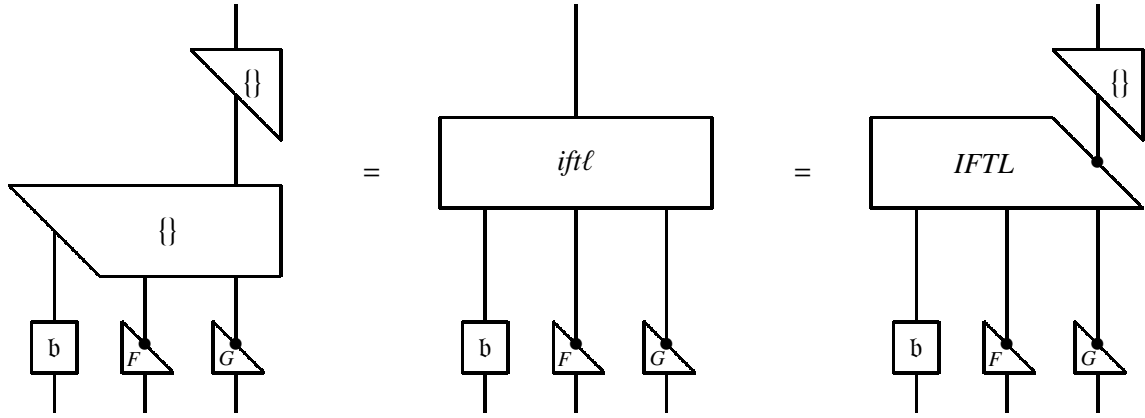


Figure 10: If  $\{\}_{I}^{\mathbb{P}} = \text{id}_{\mathbb{P}}$  then  $\{F\}_A^{\mathbb{P}} = \{[F]_A\}_I^{\mathbb{P}} = [F]_A$

### Work 3.6.1: Lazy branching

Lazy branching:



### Work 3.6.2: Stateful fixpoints

- a. Let  $G: \mathbb{P}$  be a program for  $g([p](p, x), x, a) = \{G\}(p, x, a)$ . Defining  $\Gamma: X \longrightarrow \mathbb{P}$  to be  $\Gamma x = [G](G, x)$  gives

$$g(\Gamma x, x, a) = g([G](G, x), x, a) = \{G\}(G, x, a) = \{[G](G, x)\} a = \{\Gamma x\} a$$

Draw the diagrams generalizing figures 3.5 and 3.6.

- b. Given a program transformer  $\theta: \mathbb{P} \times X \longrightarrow \mathbb{P}$ , define  $g: \mathbb{P} \times X \times A \longrightarrow B$  by

$$g(p, x, a) = \{\theta(p, x)\} a$$

and apply the preceding exercise.

### Work 3.6.3: Qing, kuine,...

- a. A narcissist  $T$  is a Kleene fixpoint of the function

$$t(p, x) = \{x\} p$$

- b. Let  $W$  be a Kleene fixpoint of the function

$$w(p, x, y) = \langle [p] x, [p] x, \{x\} y \rangle$$

The virus generator is  $\gamma = [W]$ .

- c. A *Qing* and a *Kueen* can be constructed as Smullyan fixpoints of the projections

$$g(p, q, x) = q \qquad h(p, q, x) = p$$



### Work 4.5.1: Fibonacci

Since the pairs of adult rabbits produce pairs of baby rabbits, we count the pairs. Writing  $a_n$  and  $b_n$ , respectively, for the numbers of adult pairs and baby pairs at time  $n$ , the population growth rules are:

$$\begin{array}{ll} b_0 = 1 & b_{n+1} = a_n \\ a_0 = 0 & a_{n+1} = a_n + b_n \end{array}$$

Adding up the two pairs of equations to get the total number of rabbits  $t_n = a_n + b_n$ , we get

$$t_0 = 1 \qquad t_{n+1} = t_n + t_{n-1}$$

But since the step  $t_{n+1}$  depends on two predecessors the induction requires two base cases, i.e. also

$$t_1 = a_1 + b_1 = b_0 + a_0 = 1 + 0 = 1$$

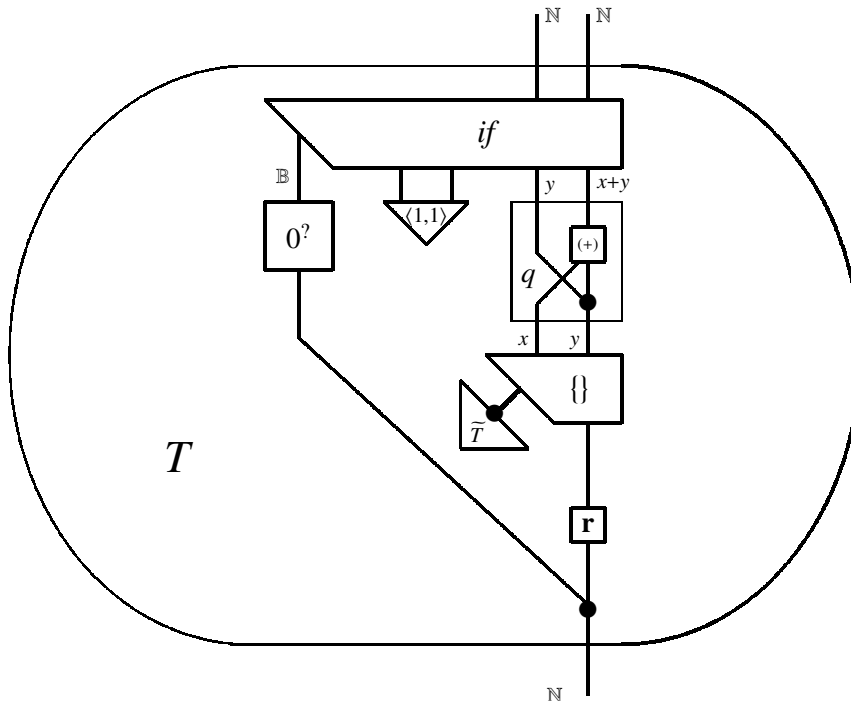
Writing  $T_n = \langle t_i, t_{i+1} \rangle$ , the induction is thus

$$T_0 = \langle 1, 1 \rangle \qquad T_{n+1} = \langle t_{n+1}, t_n + t_{n+1} \rangle$$

To capture this two-step induction, set  $B$  in (4.13) to be  $\mathbb{N} \times \mathbb{N}$  giving

$$\frac{b : \mathbb{N} \times \mathbb{N} \qquad q : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}}{T : \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}}$$

where  $b = \langle 1, 1 \rangle$  and  $q(x, y) = \langle y, x + y \rangle$ . Instantiating Fig. 4.5, the string program for rabbits is thus



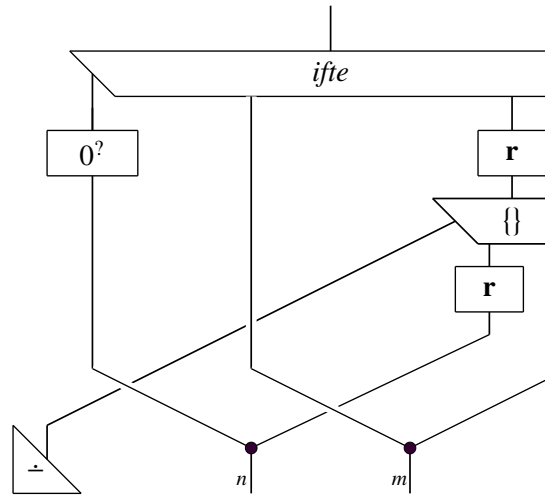
To complete the workout, explain the evaluation of this program, as a special case of Fig. 4.9.

## Work 4.5.2: Recursion

- a. iii) The truncated subtraction, or *monus*, is usually defined by

$$m \dot{-} 0 = m \qquad m \dot{-} (n + 1) = \mathbf{r}(m \dot{-} n)$$

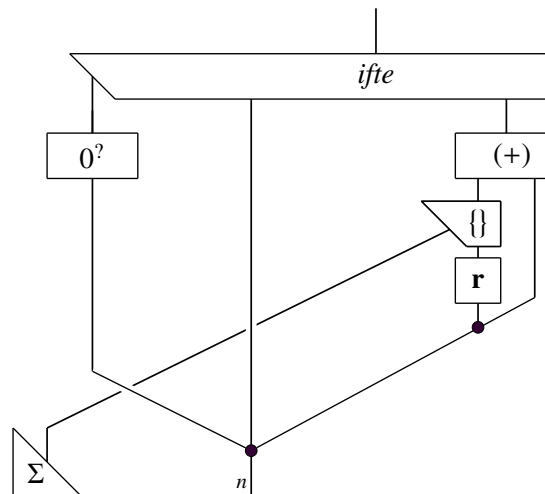
where  $\mathbf{r}$  is the predecessor. The corresponding string program is



- iv) The recursive definition

$$\Sigma(0) = 0 \qquad \Sigma(n + 1) = \Sigma(n) + n + 1$$

gives the string program



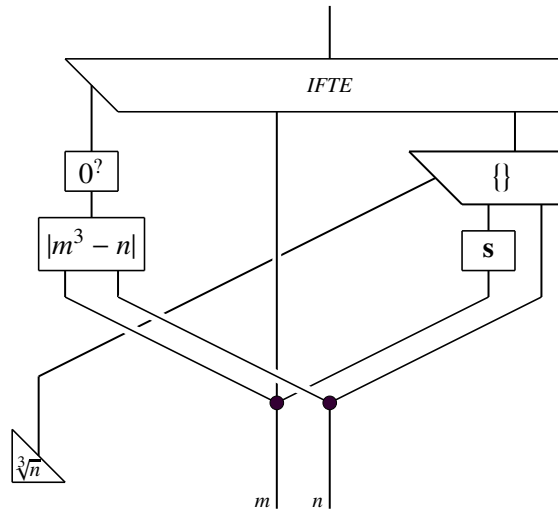
### Work 4.5.3: Search

a. In arithmetic,  $\sqrt[3]{n}$  is the smallest number  $m$  such that  $m \times m \times m = n$ . The recursive definition is thus

$$\sqrt[3]{n} = \mu m.0^? |(m \times m \times m) - n|$$

where  $|x - y| = \max(x \div y, y \div x)$

Since  $\max(a, b) = (a \div b) + b$  is clearly recursive,  $|x - y|$  is recursive. Using the unbounded search, we can thus define  $\sqrt[3]{n} = MU(|m^3 - n|)$ , which is the following diagram

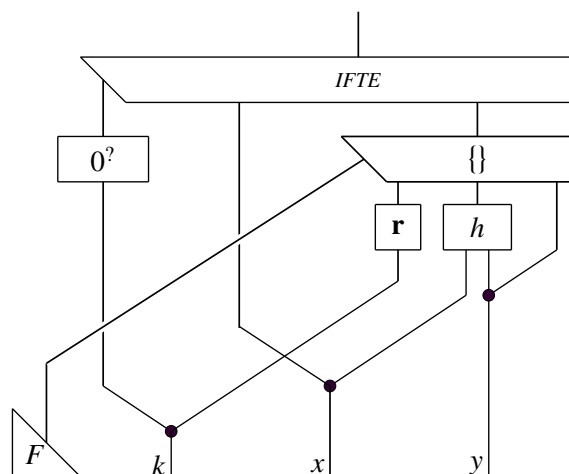


b. ii) Define  $\tilde{f}(p, k, x, y) = \text{ifte}(0^2(k), x, \{p\}(\mathbf{r}(k), h(x, y), y))$ . If  $F$  is the Kleene fixed point of  $\tilde{f}$ , i.e.

$$\{F\}(k, x, y) = ifte(0?(k), x, \{F\}(\mathbf{r}(k), h(x, y), y))$$

then set  $f = FOR(k, h) = \{[F]k\}$  reduces to

$$f(k, x, y) = \begin{cases} x & \text{if } k = 0 \\ f(\mathbf{r}(k), h(x, y), y) & \text{otherwise} \end{cases}$$



### Work 5.5: Rice

a.  $P_1(x) \iff \{x\}4 = 7$  is **undecidable** by Rice's Theorem, because it is

**extensional:**  $\{x\} = \{y\} \implies \{x\}4 = \{y\}4 \implies P_1(x) = P_1(y)$ .

**nontrivial:** Let  $\hat{n}$  be a program such that

$$\{\hat{n}\}x = n \quad (31)$$

for all  $x$ . Then  $P_1(\widehat{7}) \neq P_1(\widehat{8})$ .

b.  $P_2(x) \iff \{4\}x = 7$ : **We cannot tell** whether  $P_2$  is decidable without the particular encoding of programs by numbers. Just like any word in a language can be declared to denote any chosen concept, the number 4 can be declared to encode any chosen computation. There are thus the encodings where  $P_2$  is decidable, e.g. because it is trivial, since 4 could encode a total function, or an empty function. And there are also encodings where  $P_2$  is undecidable because the number 4 encodes a program evaluator that runs  $x$ .

c.  $P_3(x) \iff \{x\}25 < 1000$  is **undecidable** by Rice's Theorem, because it is

**extensional:**  $\{x\} = \{y\} \implies \{x\}25 = \{y\}25 \implies P_3(x) = P_3(y)$

**nontrivial:** Using the programs  $\hat{n}$  from (31) again, this time we have  $P_3(\widehat{999}) \neq P_3(\widehat{1000})$ .

d.  $P_4(x) \iff \{x\}2 \leq \{x\}3$  is **undecidable** by Rice's Theorem, because it is

**extensional:**  $\{x\} = \{y\} \implies P_4(x) = P_4(y)$  follows from

$$\begin{aligned} \{x\} = \{y\} &\implies ((\{x\}2 \leq \{x\}3) \iff (\{y\}2 \leq \{y\}3)) \\ &\implies P_4(x) = P_4(y) \end{aligned}$$

**nontrivial:** Using the program  $\hat{n}$  from (31) and  $\tilde{n}$  with

$$\{\tilde{n}\}x = n \div x \quad (32)$$

we have  $P_4(\widehat{7}) \neq P_4(\widehat{\tilde{7}})$ .

e.  $P_5(x) \iff [x]2 = 2$  is **decidable** because  $[x]$  is a cartesian function. The value  $[x]2$  can be computed and compared with 2.

f.  $P_6(x) \iff \{x\}13 \uparrow$  is **undecidable** by Rice's Theorem, because it is

**extensional:**  $\{x\} = \{y\} \implies \{x\}13 = \{y\}13 \implies P_6(x) = P_6(y)$

**nontrivial:** Using the programs  $\hat{n}$  again, which define total functions, and  $\uparrow$  which always diverges, we have  $P_6(\widehat{1}) \neq P_6(\uparrow)$ .

g.  $P_7(x) \iff \{13\}x \uparrow$ : **We cannot tell** whether the computation  $\{13\}$  is defined on  $x$  because this depends on which computation is encoded by the number 13, and this is a matter of convention.

h.  $P_8(x) \iff [[x]]x \uparrow$ : is **undecidable** because it is always false, since  $[[ ]]$  is cartesian and therefore always defined.

i.  $P_9(x) \iff (\{x\}7 \downarrow \implies \{x\}8 \downarrow)$  is **undecidable** by Rice's Theorem, because it is

**extensional**:  $\{x\} = \{y\} \implies P_9(x) = P_9(y)$  follows from

$$\begin{aligned} \{x\} = \{y\} &\implies ((\{x\}7 \downarrow \implies \{x\}8 \downarrow) \iff (\{y\}7 \downarrow \implies \{y\}8 \downarrow)) \\ &\implies P_9(x) = P_9(y) \end{aligned}$$

**nontrivial**: Using the programs  $\hat{n}$  from (a), and the programs  $\check{n}$  defined

$$\{\check{n}\}x = \begin{cases} \uparrow & \text{if } x = n \\ x & \text{otherwise} \end{cases}$$

then  $P_9(\widehat{8}) \neq P_9(\check{8})$ . The reason is that  $\{\widehat{8}\}$  is everywhere defined, and in particular on 7 and 8, and thus  $P_9(\widehat{8}) = \top$ , whereas  $\{\check{8}\}7 \downarrow$  whereas  $\{\check{8}\}8 \uparrow$ , so that the implication  $\{\check{8}\}7 \downarrow \implies \{\check{8}\}8 \uparrow$  is false, and thus  $P_9(\check{8}) = \perp$ .

j.  $P_{10}(x) \iff \forall n. \{x\}n \leq \{x\}(n+1)$  is **undecidable** by Rice's Theorem, because it is

**extensional**:  $\{x\} = \{y\} \implies P_{10}(x) = P_{10}(y)$  follows from

$$\begin{aligned} \{x\} = \{y\} &\implies (\forall n. (\{x\}n \leq \{x\}n) \iff \forall n. (\{y\}n \leq \{y\}n)) \\ &\implies P_{10}(x) = P_{10}(y) \end{aligned}$$

**nontrivial**: Using the programs  $\hat{n}$  from (a), and  $\widetilde{n}(x) = n \div x$  we have  $P_{10}(\widehat{7}) \neq P_{10}(\widetilde{7})$ .

k.  $P_{11}(x) \iff \exists n. \{x\}n \downarrow$  is **undecidable** by Rice's Theorem, because it is

**extensional**:  $\{x\} = \{y\} \implies P_{11}(x) = P_{11}(y)$  follows from

$$\begin{aligned} \{x\} = \{y\} &\implies \exists n. \{x\}n \downarrow \iff \exists n. \{y\}n \downarrow \\ &\implies P_{11}(x) = P_{11}(y) \end{aligned}$$

**nontrivial**: Using the programs  $\hat{n}$  from (a), which define total functions, and, e.g., a program  $T_1$  for  $t_1(n) = \mu x. n + x + 1 = 0$ , which is always undefined, we have  $P_{11}(\widehat{1}) \neq P_{11}(\widehat{T_1})$ .

l.  $P_{12}(x) \iff \{x\}x \downarrow$  is **undecidable** using the construction from the proof that the Halting Problem is undecidable. If  $P_{12}$  is decidable then we can define

$$\dagger P_{12}(x) = \begin{cases} \uparrow & \text{if } P_{12}(x) \\ \top & \text{if } \neg P_{12}(x) \end{cases} \quad (33)$$

so that  $\dagger P_{12}(x) \downarrow = \neg P_{12}(x)$ .

Let  $\Pi$  be a program for  $\mathfrak{P}_{12}$ . This means that  $\mathfrak{P}_{12}(x) = \{\Pi\} x$ , and in particular

$$\mathfrak{P}_{12}(\Pi) = \{\Pi\} \Pi \quad (34)$$

But now

$$P_{12}(\Pi) \xLeftrightarrow{(*)} \{\Pi\} \Pi \downarrow \xLeftrightarrow{(34)} \mathfrak{P}_{12}(\Pi) \downarrow \xLeftrightarrow{(33)} \neg P_{12}(\Pi) \nmid$$

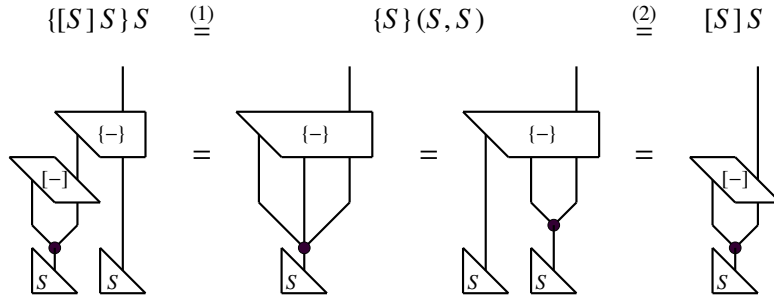
where  $(*)$  follows from the definition of  $P_{12}$ . So the assumption that  $P_{12}$  is decidable leads to contradiction. Therefore,  $P_{12}$  must be undecidable.

### Work 6.5.1: Fourth Futamura?

Using a specializer  $S$  defined by  $\{S\}(x, y) = [x]y$ , the three Futamura projections in figures 6.3–6.5 are

$$\begin{aligned} C_1 x &= [H]x = \{S\}(H, x) = \{[S]H\}x \\ C_2 &= [S]H = \{S\}(S, H) = \{[S]S\}H \\ C_3 &= [S]S = \{S\}(S, S) = \{[S]S\}S \end{aligned}$$

The pattern is thus that  $C_i$  is recovered as an image along  $\{C_{i+1}\}$ . But the process settles on  $C_3 = \{C_3\}S$  because



where

(1)  $\{[x]y\}z = \{x\}(y, z)$  holds by the definition of  $[-]$ , and

(2)  $\{S\}(x, y) = [x]y$  holds by the definition of  $S$ .

### Work 6.5.2: Hyper

a. i) The assumption  $f = \langle g, h \rangle$  means

$$f(0, x) = g(x) \quad f(1 + k, x) = h(k, f(k, x), x)$$

and implies

$$\varphi(0, x, y) = \gamma(x, y) \quad \varphi(1 + k, x, y) = \chi(k, f(k, x), x, y) \quad (35)$$

On the other hand, the desired conclusion  $\varphi = \langle \gamma, \chi \rangle$  requires

$$\varphi(0, x, y) = \gamma(x, y) \quad \varphi(1 + k, x, y) = \chi(k, \{f(k, x)\}y, x, y) \quad (36)$$

Draw the diagram to see the difference! Under which condition can (36) be derived from (35)?

**Work ??: Order**

- a. Prove that (??) determines a wellordering on an arbitrary type  $A$ .
- b. Specify diagrammatic programs for infima and suprema of pairs of programs with respect to the program order.
- c. Draw the diagrammatic program in (??).

20221212



# Bibliography

- [1] Samson Abramsky. Intensionality, definability and computation. In Alexandru Baltag and Sonja Smets, editors, *Johan van Benthem on Logic and Information Dynamics*, pages 121–142. Springer, 2014.
- [2] Samson Abramsky and Achim Jung. Domain theory. In Samson et al Abramsky, editor, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.
- [3] Jiri Adamek. Introduction to coalgebra. *Theory and Applications of Categories*, 14:157–199, 2005.
- [4] Jiri Adamek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. Wiley, 1990. available online.
- [5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Series in Computer Science and Information Processing. Addison-Wesley, 1986. 2nd edition in 2006, coauthored with Monica Lam.
- [6] Peter M. Alberti and Armin Uhlmann. *Stochasticity and partial order: double stochastic maps and unitary mixing*. Mathematics and its applications. Deutscher Verlag der Wissenschaften, 1981.
- [7] Christoph Arndt. *Information Measures: Information and its Description in Science and Engineering*. Signals and Communication Technology. Springer Berlin Heidelberg, 2012.
- [8] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [9] Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier, editors. *Séminaire de Géométrie Algébrique: Théorie des Topos et Cohomologie Étale des Schemas (SGA 4)*, volume 269,270,305 of *Lecture Notes in Mathematics*. Springer-Verlag, 1964. Second edition, 1972.
- [10] Steve Awodey. *Category Theory*. Oxford Logic Guides. OUP, 2010.
- [11] John Baez and Aaron Lauda. A prehistory of  $n$ -categorical physics. In *Deep Beauty: Understanding the Quantum World Through Mathematical Innovation*, pages 13–128. Cambridge University Press, 2011.
- [12] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.
- [13] Charles H Bennett. The thermodynamics of computation — a review. *International Journal of Theoretical Physics*, 21(12):905–940, 1982.
- [14] Charles H. Bennett. Logical depth and physical complexity. In *A half-century survey on The Universal Turing Machine*, pages 227–257, New York, NY, USA, 1988. Oxford University Press, Inc.

- [15] Ingemar Bethke. *Notes on Partial Combinatory Algebras*. PhD thesis, University of Amsterdam, 1988.
- [16] Lars Birkedal. *Developing Theories of Types and Computability via Realizability*, volume 34 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.
- [17] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2):322–336, April 1967.
- [18] Francis Borceux. *Handbook of Categorical Algebra*. Number 50 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1994. Three volumes.
- [19] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.*, 4(POPL):28:1–28:28, 2020.
- [20] Cristian Calude. *Theories of Computational Complexity*, volume 35 of *Annals of Discrete Mathematics*. North-Holland, 1988.
- [21] Georg Cantor. Über eine elementare Frage der Mannigfaltigkeitslehre. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 1:75–78, 1891.
- [22] Georg Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 46(4):481–512, 1895.
- [23] Georg Cantor. *Gesammelte Abhandlungen mathematischen und philosophischen Inhalts*. Springer-Verlag, 1932. Edited by Ernst Zermelo; reprinted by Olms, Hildeshaim, 1962.
- [24] Aurelio Carboni and Robert F.C. Walters. Cartesian bicategories, I. *J. of Pure and Applied Algebra*, 49:11–32, 1987.
- [25] Jason Castiglione, Dusko Pavlovic, and Peter-Michael Seidel. Privacy protocols. In Joshua Guttman et al., editor, *CathFest: Proceedings of the Symposium in Honor of Catherine Meadows*, volume 11565 of *Lecture Notes in Computer Science*, pages 167–192. Springer, 2019.
- [26] Noam Chomsky. *Language and Mind*. Cambridge University Press, 2006.
- [27] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936. Reprinted in [41].
- [28] Alonzo Church. The constructive second number class. *Bulletin of the American Mathematical Society*, 44(4):224–232, 1938.
- [29] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.
- [30] Alonzo Church. *The Calculi of Lambda Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, 1941.
- [31] Alonzo Church and Stephen C. Kleene. Constructions of formal definitions of functions of ordinal numbers. *Bull. Amer. math. Soc.* 42, 639, 1936.

- [32] Alonzo Church and Stephen Cole Kleene. Formal definitions in the theory of ordinal numbers. *Fundam. Math.*, 28:11–21, 1937.
- [33] J.Robin B. Cockett and Pieter J.W. Hofstra. Introduction to turing categories. *Annals of Pure and Applied Logic*, 156(2):183 – 209, 2008.
- [34] Bob Coecke, Éric Oliver Paquette, and Dusko Pavlovic. Classical and quantum structuralism. In Simon Gay and Ian Mackie, editors, *Semantical Techniques in Quantum Computation*, pages 29–69. Cambridge University Press, 2009.
- [35] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.
- [36] John H. Conway. *On Numbers and Games*. Number 6 in London Mathematical Society Monographs. Academic Press, 1976.
- [37] John H. Conway and Richard Guy. *The Book of Numbers*. Springer, 2012.
- [38] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76:95–120, 1988.
- [39] Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20:584–590, 1934.
- [40] Joseph W. Dauben. *Georg Cantor: His Mathematics and Philosophy of the Infinite*. Princeton University Press, 2020.
- [41] Martin Davis, editor. *The Undecidable : Basic papers on undecidable propositions, unsolvable problems, and computable functions*. Raven Press, Helwett, N.Y., 1965.
- [42] Richard Dedekind. *Essays on the Theory of Numbers*. The Religion of Science Library. Open Court Publishing Company, 1901. reprinted by Dover in 1963.
- [43] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [44] John Doner and Alfred Tarski. An extended arithmetic of ordinal numbers. *Fundamenta Mathematicae*, 65(1):95–127, 1969.
- [45] Beno Eckmann and Peter Hilton. Structure maps in group theory. *Fundamenta Mathematicae*, 50(2):207–221, 1961.
- [46] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.
- [47] Samuel Eilenberg and Norman Steenrod. *Foundations of Algebraic Topology*. Princeton University Press, 1952.
- [48] Yu.L. Ershov. Enumeration of families of general recursive functions. *Siberian Mathematical Journal*, 8(5):771–778, 1967.
- [49] Yu.L. Ershov. *Theory of Enumerations*. Mathematical Logic and Foundations of Mathematics.

- Nauka, Moscow, 1977. (in Russian).
- [50] Solomon Feferman. A language and axioms for explicit mathematics. In *Algebra and logic*, pages 87–139. Springer, 1975.
  - [51] Richard Feynman. *The Character of Physical Law*. The MIT Press. MIT Press, 1967.
  - [52] Peter Freyd and Dusko Pavlovic. Conversation on the categories mailing list 12–13 february 1994. <https://www.mta.ca/cat-dist/archive/>.
  - [53] Yoshihiko Futamura. Partial evaluation of computation process—approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, December 1999.
  - [54] Robin Gandy. The confluence of ideas in 1936. In Rolf Herken, editor, *A half-century survey on The Universal Turing Machine*, pages 55–111. Springer, 1988.
  - [55] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *Continuous Lattices and Domains*. EBSCO ebook academic collection. Cambridge University Press, 2003.
  - [56] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theoretical computer science*, 45:159–192, 1986.
  - [57] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
  - [58] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English translations, “On Formally Undecidable Propositions of ‘Principia Mathematica’ and Related Systems” published by Oliver and Boyd, 1962 and Dover, 1992; also in [67], pp. 596–616; also in [41], pages 5–38.
  - [59] Kurt Gödel. Remarks before the princeton bicentennial conference of problems in mathematics. In Martin Davis, editor, *The Undecidable : Basic papers on undecidable propositions, unsolvable problems, and computable functions*, pages 84–87. Raven Press, Helwett, N.Y., 1965.
  - [60] Hermann Grassmann. *A New Branch of Mathematics: The “Ausdehnungslehre” of 1844 and Other Works*. Open Court, 1995.
  - [61] Hermann Grassmann. *Extension Theory*, volume 19 of *History of Mathematics*. American Mathematical Soc., 2000.
  - [62] Alexandre Grothendieck. Sur quelques points d’algèbre homologique. *Tohoku Mathematical Journal, Second Series*, 9(2):119–183, 1957.
  - [63] Andrzej Grzegorzcyk. *Some classes of recursive functions*, volume 4 of *Rosprawy Matematyczne*. Instytut Matematyczny Polskiej Akademi Nauk, 1953.
  - [64] Carl Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
  - [65] Yuri Gurevich. Logic activities in Europe. *ACM SIGACT News*, 25(2):11–24, June 1994.
  - [66] Susumu Hayashi. Adjunction of semifunctors: Categorical structures in nonextensional lambda

- calculus. *Theor. Comput. Sci.*, 41:95–104, 1985.
- [67] Jean van Heijenoort, editor. *From Frege to Gödel: a Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, 1967. Reprinted 1971, 1976.
- [68] Arend Heyting. Die formalen Regeln der intuitionistischen Logik. I, II, III. *Sitzungsber. Preuß. Akad. Wiss., Phys.-Math. Kl.*, 1930:42–56, 57–71, 158–169, 1930.
- [69] David Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–190, 1926. English translation in [67], pp. 367–392.
- [70] Arie Hinkis. *Proofs of the Cantor-Bernstein Theorem: A Mathematical Excursion*, volume 45 of *Science Networks. Historical Studies*. Springer, 2013.
- [71] Ralph Hinze and Dan Marsden. *The Art of Category Theory Part I: Introducing String Diagrams*, 2019. Completed book, under review.
- [72] C.A.R. Hoare. Hints on the design of a programming language for real-time command and control. In J.P. Spencer, editor, *Real-time Software: International State of the Art Report*, pages 685–99. Infotech International, 1976.
- [73] Douglas R. Hofstadter. *Gödel, Escher, Bach*. Basic Books. Basic Books, 1979.
- [74] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, USA, 3rd edition edition, 2006.
- [75] Günter Hotz. Eine Algebraisierung des Syntheseproblems von Schaltkreisen I–II. *Elektronische Informationsverarbeitung und Kybernetik*, 1–2(3):185–205, 209–231, 1965.
- [76] William A. Howard. The fomulæ-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism*, volume 2, pages 479–490. Oxford University Press, 1980.
- [77] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, sep 1989.
- [78] Martin Hyland. The effective topos. In Anne Sjerp Troelstra and Dirk van Dalen, editors, *L. E. J. Brouwer Centenary Symposium*, number 110 in *Studies in Logic and the Foundations of Mathematics*, pages 165–216. North-Holland, 1982.
- [79] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*, volume 59 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2016.
- [80] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*. Oxford Logic Guides. Clarendon Press, 2002.
- [81] Neil D. Jones. *Computability and Complexity: From a Programming Perspective*. Foundations of Computing. MIT Press, 1997.
- [82] André Joyal and Ross Street. The geometry of tensor calculus I. *Adv. in Math.*, 88:55–113, 1991.

- [83] André Joyal and Ross Street. The geometry of tensor calculus II, 1995. Draft available from Ross Street's website as item 53.
- [84] Daniel M. Kan. Adjoint functors. *Transactions of the American Mathematical Society*, 87(2):294–329, 1958.
- [85] Gregory Maxwell Kelly. *Basic Concepts of Enriched Category Theory*. Number 64 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1982. Reprinted in *Theory and Applications of Categories*, No. 10 (2005) pp. 1–136.
- [86] Stephen C. Kleene. General recursive functions of natural numbers. *Bull. Amer. Math. Soc.*, 41, 1935.
- [87] Stephen C. Kleene. General recursive functions of natural numbers. *Math. Ann.*, 112:727–742, 1936.
- [88] Stephen C. Kleene.  $\lambda$ -definability and recursiveness. *Duke Math. J.*, 2:340–353, 1936.
- [89] Stephen C. Kleene. On notation for ordinal numbers. *Journal of Symbolic Logic*, 3(4):150–155, 1938.
- [90] Stephen C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10(4):109–124, 1945.
- [91] Stephen C. Kleene. *Introduction to Meta-Mathematics*. North-Holland Publ. Co., Amsterdam, 1952.
- [92] Christiaan P.J. Koymans. Models of the lambda calculus. *Inf. Control.*, 52(3):306–332, 1982.
- [93] Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In Arend Heyting, editor, *Constructivity in Mathematics. Proceedings of the Colloquium Held at Amsterdam 1957*, Studies in logic and the foundations of mathematics, pages 101–128. North-Holland, 1959.
- [94] Joachim Lambek and Philip Scott. *Introduction to Higher Order Categorical Logic*. Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986.
- [95] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961.
- [96] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(1):869–872, 1963.
- [97] F. William Lawvere. The category of categories as a foundation for mathematics. In *Proceedings of the Conference on Categorical Algebra held in La Jolla*, pages 1–20. Springer, 1966.
- [98] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23:281–296, 1969. reprint in *Theory and Applications of Categories*, No. 16, 2006, pp.1–16.
- [99] F. William Lawvere. Equality in hyperdoctrines and the comprehension schema as an adjoint functor. In Alex Heller, editor, *Applications of Categorical Algebra*, number 17 in *Proceedings of Symposia in Pure Mathematics*, pages 1–14. American Mathematical Society, 1970.

- [100] F. William Lawvere. Metric spaces, generalised logic, and closed categories. In *Rendiconti del Seminario Matematico e Fisico di Milano*, volume 43. Tipografia Fusi, Pavia, 1973. Reprinted in *Theory and Applications of Categories*, No. 1, 2002, pp. 1–37.
- [101] Leonid A. Levin. Forbidden information. *J. ACM*, 60(2), may 2013.
- [102] Ming Li and Paul M. B. Vitányi. *An introduction to Kolmogorov complexity and its applications* (2. ed.). Graduate texts in computer science. Springer, 1997.
- [103] John R. Longley. *Realizability toposes and language semantics*. PhD thesis, University of Edinburgh, 1995. available at [www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-332/](http://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-332/).
- [104] Giuseppe Longo and Eugenio Moggi. Cartesian closed categories of enumerations for effective type structures. In Plotkin G. Kahn G., MacQueen D.B., editor, *Semantics of Data Types (SDT)*, number 173 in Lecture Notes in Computer Science, pages 235–255. Springer-Verlag, 1984.
- [105] Saunders Mac Lane. *Homology*. Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1963.
- [106] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.
- [107] Albert W. Marshall and Ingram Olkin. *Inequalities: Theory of Majorization and Its Applications*, volume 143 of *Mathematics in Science and Engineering*. Academic Press, 1979.
- [108] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.
- [109] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.
- [110] Yiannis N. Moschovakis. Kleene’s Amazing Second Recursion Theorem. *Bulletin of Symbolic Logic*, 16(2):189–239, 2010.
- [111] Philip S. Mulry. Generalized banach-mazur functionals in the topos of recursive sets. *J. of Pure and Applied algebra*, 26(1):71–83, 1982.
- [112] John Myhill. Creative sets. *Mathematical Logic Quarterly*, 1(2):97–108, 1955.
- [113] Ron P. Nederpelt, J. Herman Geuvers, and Roel C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 1994.
- [114] John von Neumann and A.W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1984.
- [115] Emmy Noether and Jean Cavaillès, editors. *Briefwechsel Cantor-Dedekind*, volume 518 of *Actualités Scientifiques et Industrielles*. Hermann & Cie., Paris, 1937.
- [116] Bent Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf’s Type Theory: An Introduction*. International series of monographs on computer science. Clarendon Press, 1990.
- [117] Piergiorgio Odifreddi. *Classical Recursion Theory : The Theory of Functions and Sets of Natural Numbers*, volume 125 of *Studies in logic and the foundations of mathematics*. North-Holland,

Amsterdam, New-York, Oxford, Tokyo, 1989.

- [118] Jaap van Oosten. *Realizability: An Introduction to its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 2008.
- [119] Robert A. Di Paola and Alex Heller. Dominical categories: Recursion theory without elements. *The Journal of Symbolic Logic*, 52(3):594–635, 1987.
- [120] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [121] Robert Paré. On absolute colimits. *J. Alg.*, 19:80–95, 1971.
- [122] Dusko Pavlovic. Categorical interpolation: descent and the Beck-Chevalley condition without direct images. In A. Carboni et al., editor, *Category Theory, Como 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 306–326. Springer Verlag, 1991.
- [123] Dusko Pavlovic. Constructions and predicates. In D. Pitt et al., editor, *Category Theory and Computer Science '91*, volume 530 of *Lecture Notes in Computer Science*, pages 173–197. Springer Verlag, 1991.
- [124] Dusko Pavlovic. Maps II: Chasing diagrams in categorical proof theory. *J. of the IGPL*, 4(2):1–36, 1996.
- [125] Dusko Pavlovic. Geometry of abstraction in quantum computation. *Proceedings of Symposia in Applied Mathematics*, 71:233–267, 2012. [arxiv.org:1006.1010](https://arxiv.org/abs/1006.1010).
- [126] Dusko Pavlovic. Monoidal computer I: Basic computability by string diagrams. *Information and Computation*, 226:94–116, 2013. [arxiv:1208.5205](https://arxiv.org/abs/1208.5205).
- [127] Dusko Pavlovic. Lambek pregroups are Frobenius spiders in preorders. *Compositionality*, 4(1):1–21, 2022. [arxiv:2105.03038](https://arxiv.org/abs/2105.03038).
- [128] Dusko Pavlovic and Douglas R. Smith. Composition and refinement of behavioral specifications. In *Automated Software Engineering 2001. The Sixteenth International Conference on Automated Software Engineering*. IEEE, 2001.
- [129] Dusko Pavlovic and Muzamil Yahia. Monoidal computer III: A coalgebraic view of computability and complexity. In C. Cîrstea, editor, *Coalgebraic Methods in Computer Science (CMCS) 2018 — Selected Papers*, volume 11202 of *Lecture Notes in Computer Science*, pages 167–189. Springer, 2018. [arxiv:1704.04882](https://arxiv.org/abs/1704.04882).
- [130] Roger Penrose. Applications of negative dimensional tensors. In D. J. A. Welsh, editor, *Combinatorial mathematics and its applications. Proceedings of a conference held at the Mathematical Institute, Oxford, 7–10 July, 1969*, pages 221–244. Academic Press, 1971.
- [131] Rozsa Péter. *Recursive Functions*. Academic Press, 1967.
- [132] Karl R. Popper. *The Logic of Scientific Discovery*. Basic Books, 1954.
- [133] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bull. Amer. math. Soc.*, 50:284–316, 1944.



- [134] William Poundstone. *The Recursive Universe: Cosmic Complexity and the Limits of Scientific Knowledge*. Dover Books on Science. Dover Publications, 2013.
- [135] H. Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.
- [136] Emily Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2017.
- [137] Hartley Rogers, Jr. Gödel numberings of partial recursive functions. *The Journal of Symbolic Logic*, 23(3):331–341, 1958.
- [138] Hartley Rogers, Jr. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, USA, 1987.
- [139] Bertrand Russell. Letter to Frege, 1902. In [67].
- [140] Bertrand Russell. Mathematical logic based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908. Reprinted in [67], pages 150–182.
- [141] Bertrand Russell and Alfred North Whitehead. *Principia Mathematica*. Cambridge University Press, 1910–13.
- [142] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92(3):305–316, 1924.
- [143] Dana Scott. Data types as lattices. *SIAM Journal on Computing*, 5:522–587, 1976.
- [144] Dana S. Scott. Outline of mathematical theory of computation. In *Proceedings of the 4th Annual Princeton Conf. on Information Sciences and Systems*, pages 169–176. Princeton University Press, 1970.
- [145] Dana S. Scott. Continuous lattices. In F.W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 2, pages 403–450. Springer, 1980.
- [146] Dana S. Scott. Relating theories of the lambda calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism*, volume 2, pages 403–450. Academic Press, 1980.
- [147] Joel I. Seiferas. Machine-independent complexity theory. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 163–186. Elsevier, Amsterdam, 1990.
- [148] Jeffrey Shallit. *A second course in formal languages and automata theory*. Cambridge University Press, 2008.
- [149] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. The Mathematical Theory of Communication. University of Illinois Press, 1962.
- [150] Wilfried Sieg. The Cantor-Bernstein theorem: how many proofs? *Philosophical Transactions of the Royal Society A*, 377(2140):20180031, 2019.

- [151] Harold Simmons. *An Introduction to Category Theory*. Cambridge University Press, 2011.
- [152] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2013.
- [153] Craig Smoryński. *Self-Reference and Modal Logic*. Universitext. Springer New York, 2012.
- [154] Raymond M. Smullyan. *Recursion Theory for Metamathematics*. Oxford Logic Guides. Oxford University Press, 1993.
- [155] Raymond M. Smullyan. *Diagonalization and Self-reference*. Logic Guides Series. Clarendon Press, 1994.
- [156] Raymond M. Smullyan. *What is the Name of this Book?* Dover Recreational Math Series. Dover Publications, 2011.
- [157] Raymond M. Smullyan. *To Mock a Mocking Bird*. Knopf Doubleday, 2012.
- [158] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982.
- [159] Michael B. Smyth. Topology. In S. Abramsky and D. Gabbay, editors, *Handbook of Logic in Computer Science (Vol. 1). Background: Mathematical Structures*, pages 641–761. Oxford University Press, Inc., USA, 1993.
- [160] Robert I. Soare. *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Perspectives in Mathematical Logic. Springer, 1999.
- [161] Robert I. Soare. *Turing Computability: Theory and Applications*. Theory and Applications of Computability. Springer, 2016.
- [162] Viggo Stoltenberg-Hansen, Ingrid Lindström, and Edward R. Griffor. *Mathematical theory of domains*. Number 22 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.
- [163] Ross Street. Posting on categories mailing list of 5 may, 2017.
- [164] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261 – 405, 1935.
- [165] Anne S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer, 1973. With contributions by C.A. Smoryński, J.I. Zucker and W.A. Howard.
- [166] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics. An Introduction*, volume 121,123 of *Studies in Logic and Foundations of Mathematics*. Elsevier Science, 1988.
- [167] Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, June 1986.
- [168] Alan Turing. *Morphogenesis*. Collected Works of A.M. Turing. Elsevier Science, 1992. edited by Saunders, P.T.

- [169] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42:230–265, 1936. Reprinted in [41].
- [170] Alan M. Turing. Systems of logic based on ordinals. *Proc. of the London Mathematical Society. Second Series*, 45:161–228, 1939.
- [171] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 1990.
- [172] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990.
- [173] Moshe Y. Vardi. Why doesn't ACM have a SIG for Theoretical Computer Science? *Commun. ACM*, 58(8):5, jul 2015.
- [174] Johann von Neumann. Zur Einführung der transfiniten Zahlen. *Acta litt. Acad. Sc. Szeged*, X(1):199–208, 1923. English translation, “On the introduction of transfinite numbers” in [67], pages 393–413.
- [175] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015.
- [176] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, 1994.
- [177] Noson S. Yanofsky. *The Outer Limits of Reason: What Science, Mathematics, and Logic Cannot Tell Us*. MIT Press, 2016.

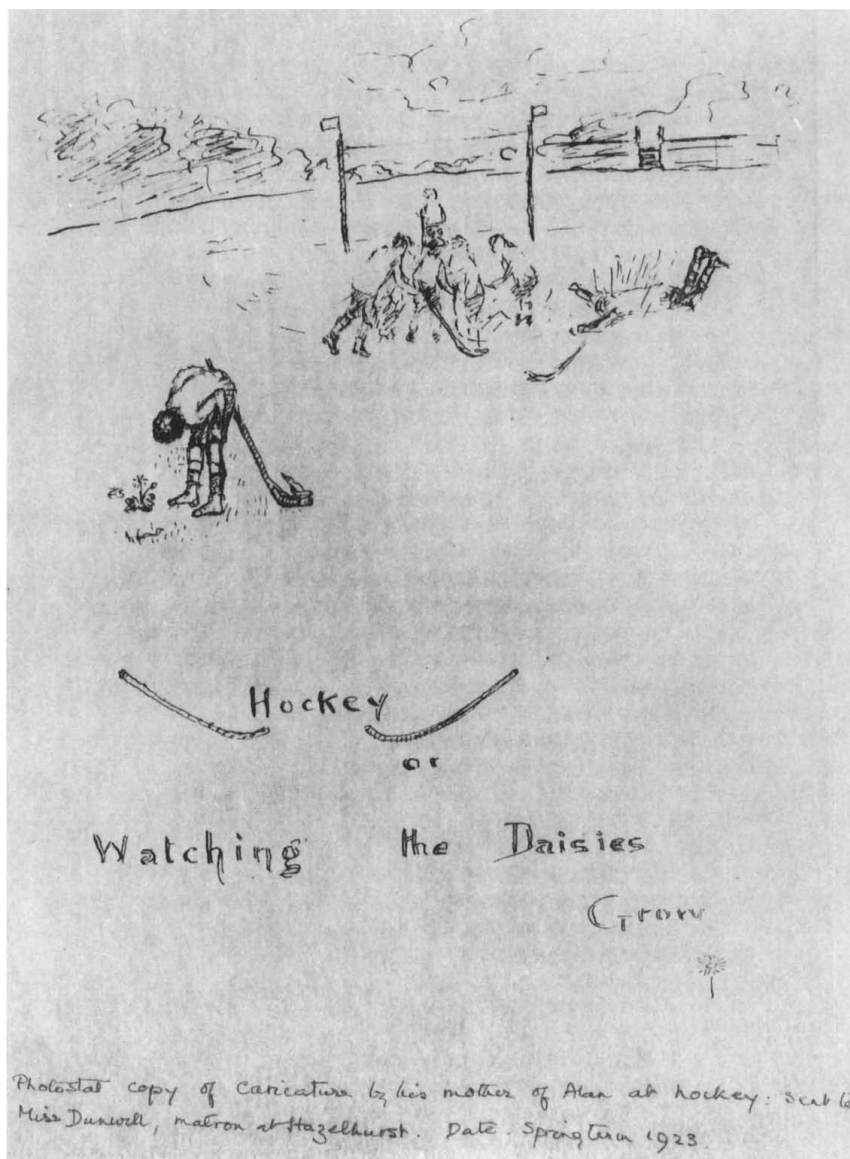


Figure 11: Drawing of 11-year old Alan Turing by his mother