Notes on realizability

[DRAFT: MAY 14, 2022]

Andrej Bauer

May 14, 2022

Preface

It is not an exaggeration to say that the invention of modern computers was a direct consequence of the great advances of the 20th century logic: Hilbert's putting a decision problem on his list, Gödel's amazing exercise in programming with numbers, Church's invention of λ-calculus, Gödel's of general recursive functions, and Turing's of his machines. Unfortunately, by the time computers took over the world and demanded a fitting foundation of mathematics, generations of mathematicians had been educated with little regard or sensitivity to questions of computability and constructivity. Some even cherished living in a paradise removed from earthly matters and encouraged others to take pride in the uselessness of their activity. Today such mathematics persists as the generally accepted canon.

How is the working mathematician to understand and study computable mathematics? Given their unshakable trust in classical mathematics, it is only natural for them to "bolt on computability as an afterthought", as was put eloquently by a friend of mine. Indeed, this is precisely how many experts practice computable mathematics, and so shall we.

A comprehensive account of realizability theory would be a monumental work, which we may hope to see one day in the form of a sketch of an elephant. These notes are at best a modest introduction that aims to strike a balance between approachable concreteness and inspiring generality. Because my purpose was to educate, I did not hesitate to include informal explanations and recollection of material that I could have relegated to background reading. Suggestions for further reading will hopefully help direct those who seek deeper knowledge of realizability.

Realizability theory weaves together computability theory, category theory, logic, topology and programming languages. I therefore recommend to the enthusiastic students the adoption of the Japanese martial art principle 修行.

An early version of these lecture notes were written to support a graduate course on computable topology, which I taught in 2009 at the University of Ljubljana. I copiously reused part of my dissertation. In 2022 I updated the notes and added a chapter on type theory, on the occasion of my lecturing at the Midlands Graduate School, hosted by the University of Nottingham.

Andrej Bauer
Ljubljana, March 2022

Contents

Preface ii

Contents iii

1 Introduction 1
1.1 Background material . . . 1

2 Models of Computation 4
2.1 Turing machines . . . 4
2.1.1 Type 1 machines . . . 6
2.1.2 Type 2 machines . . . 10
2.1.3 Turing machines with oracles . . . 15
2.1.4 Hamkin's infinite-time Turing machines . . . 15
2.2 Scott's graph model . . . 18
2.3 Church's λ-calculus . . . 21
2.4 Reflexive domains . . . 25
2.5 Partial combinatory algebras . . . 27
2.5.1 Examples of partial combinatory algebras . . . 29
2.6 Typed partial combinatory algebras . . . 31
2.7 Examples of Typed Partial Combinatory Algebras . . . 34
2.7.1 Partial combinatory algebras . . . 34
2.7.2 Simply typed λ-calculus . . . 34
2.7.3 Gödel's T . . . 35
2.7.4 Plotkin's PCF . . . 35
2.7.5 PCF^{∞} . . . 36
2.8 Simulations . . . 37
2.8.1 Properties of simulations . . . 38
2.8.2 Decidable simulations and $\mathbb{K}_1$ . . . 40
2.8.3 An adjoint retraction from $(\mathbb{P}, \mathbb{P}_\#)$ to $(\mathbb{B}, \mathbb{B}_\#)$ . . . 42

3 Realizability categories 44
3.1 Motivation . . . 44
3.2 Assemblies . . . 45
3.2.1 Modest sets . . . 46
3.2.2 The unit assembly $\mathbb{1}$ . . . 46
3.2.3 Natural numbers . . . 47
3.2.4 The constant assemblies . . . 47
3.2.5 Two-element assemblies . . . 48
3.3 Equivalent formulations . . . 49
3.3.1 Existence predicates . . . 49
3.3.2 Representations . . . 50
3.3.3 Partial equivalence relations . . . 50
3.3.4 Equivalence relations . . . 52
3.4 Applicative functors . . . 52
3.5 Schools of Computable Mathematics . . . 54
3.5.1 Recursive Mathematics . . . 54
3.5.2 Equilogical spaces . . . 54

3.5.3 Computable countably based spaces . . . 57
3.5.4 Computable equilogical spaces . . . 59
3.5.5 Type Two Effectivity . . . 60
3.6 The categorical structure of assemblies . . . 62
3.6.1 Cartesian structure . . . 62
3.6.2 Cocartesian structure . . . 64
3.6.3 Monos and epis . . . 69
3.6.4 Regular structure . . . 72
3.6.5 Cartesian closed structure . . . 74
3.6.6 The interpretation of λ-calculus in assemblies . . . 75
3.6.7 Projective assemblies . . . 79

4 Realizability and logic **81**
4.1 The set-theoretic interpretation of logic . . . 81
4.2 Realizability predicates . . . 81
4.3 The Heyting prealgebra of realizability predicates . . . 83
4.4 Quantifiers . . . 84
4.5 Substitution . . . 85
4.6 Equality . . . 86
4.7 Summary of realizability logic . . . 87
4.8 Classical and decidable predicates . . . 88
4.8.1 $\neg\neg$-stable predicates . . . 88
4.8.2 Decidable predicates . . . 89
4.8.3 Predicates classified by two-element assemblies . . . 90

5 Realizability and type theory **92**
5.1 Families of sets . . . 92
5.1.1 Products and sums of families of sets . . . 93
5.1.2 Type theory as the internal language . . . 94
5.2 Families of assemblies . . . 96
5.2.1 Products and sums of families of assemblies . . . 97
5.2.2 Contexts of assemblies . . . 98
5.3 Propositions as assemblies . . . 98
5.3.1 Propositional truncation of an assembly . . . 99
5.3.2 Realizability predicates and propositions . . . 100
5.4 Identity types . . . 102
5.4.1 UIP and equality reflection . . . 102
5.5 Inductive and coinductive types . . . 102
5.6 Universes . . . 102
5.6.1 Universes of propositions . . . 102
5.6.2 The universe of modest sets . . . 102
5.6.3 The universe of small assemblies . . . 102

6 The internal language at work **103**
6.1 Epis and monos . . . 103
6.2 The axiom of choice . . . 103
6.3 Heyting arithmetic . . . 103
6.4 Countable objects . . . 103
6.5 Markov's principle . . . 103
6.6 Church's thesis and the computability modality . . . 103
6.7 Aczel's presentation axiom . . . 103

6.8 Continuity principles 103
6.8.1 Brouwer's continuity 103
6.8.2 Kreisel-Lacombe-Schönfield-Ceitin continuity 103
6.9 Brouwer's compactness principle 103

Bibliography **104**

1 Introduction

1.1 Background material

In this section we overview a selection of concepts which we need later on. We also fix notation and a number of definitions. At the momement the sections are not listed in any particular order.

Free and bound variables

Occurrences of variables in an expression may be ***free*** or ***bound***. Variables are bound when they are used to indicate the range over which an operator acts. For example, in expressions

$$\forall x. \mathbb{R} x^2 + y \geq 0, \qquad \sum_{k=0}^{n} \frac{1}{k^2}, \qquad \int_a^b f(t)\,dt,$$

the variables x, k, and t are bound by the operators $\forall$, $\sum$, and $\int$, respectively. The remaining variables are free. It is really the ***occurrence*** of a variable that is bound or free, not the variable itself. In

$$P(x) \vee \exists x. \neg Q(x)$$

the left-most occurence of x is free whereas the other two are bound by $\exists$.

Functions

The set of all functions from A to B is denoted by B^A as well as $A \to B$. The arrow associates to the right, $A \to B \to C$ is $A \to (B \to C)$. We write $f : A \to B$ instead of $f \in A \to B$. If $f : A \to B$ and $x \in A$, the application $f(x)$ is also written as $f\,x$. We often work with ***curried*** functions which take several arguments in succession, i.e., if $f : A \to B \to C$ then f takes $x \in A$, and $y \in B$ to produce an element $f(x)(y)$ in C, also written $f\,x\,y$.

Partial functions

A ***partial*** function[1] $f : A \rightharpoonup B$ is a function that is defined on a subset $\mathsf{dom}(f) \subseteq A$, called the *domain* of f. Sometimes there is confusion between the domain $\mathsf{dom}(f)$ and the set A, which is also called the domain. We therefore call $\mathsf{dom}(f)$ the ***support*** of f. If $f : A \rightharpoonup B$ is a partial function and $x \in A$, we write $f\,x\!\downarrow$ to indicate that $f x$ is defined. For an expression e, we also write $e\!\downarrow$ to indicate that e and all of its subexpressions are defined. The symbol $\downarrow$ is sometimes inserted into larger expressions, for example, $f\,x\!\downarrow = y$ means that $f x$ is defined and is equal to y. If e_1 and e_2 are two expressions whose values are possibly undefined, we write $e_1 \simeq e_2$ to indicate that either e_1 and e_2 are both undefined, or they are both defined

1: In the literature on Type Two Effectivity the common notation is $f :\subseteq A \to B$.

and equal. The notation $e_1 \geq e_2$ means that if e_1 is defined then e_2 is defined and they are equal. Thus we have

$$e_1 \simeq e_2 \iff e_1 \geq e_2 \wedge e_2 \geq e_1.$$

A partial map $f : X \rightharpoonup Y$ between topological spaces X and Y is said to be ***continuous*** when it is continuous as a total map $f : \mathsf{dom}(f) \to Y$, where the support $\mathsf{dom}(f) \subseteq X$ is equipped with the subspace topology.

Primitive recursive and recursive function

The ***primitive recursive function*** are those function $\mathbb{N}^k \to \mathbb{N}$ that are built inductively from the following functions and operations:

1. constant functions $f(n_1, \dots, n_k) = c$, where $c \in \mathbb{N}$,
2. projections $p_i(n_1, \dots, n_k) = n_i$, where $1 \leq i \leq k$,
3. the successor function $s(n) = n + 1$,
4. composition of functions,
5. primitive recursion: given primitive recursive $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$, the function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined by

$$h(0, n_1, \dots, n_k) = f(n_1, \dots, n_k),$$
$$h(n + 1, n_1, \dots, n_k) = g(h(n, n_1, \dots, n_k), n, n_1, \dots, n_k)$$

   is primitive recursive.

Every primitive recursive function is computable, but not every computable function is primitive recursive.[2]

2: The Ackermann function is computable but not primitive recursive.

The ***(general) partial recursive functions*** are built from the above operations and ***minimization***: given a partial recursive $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ the function $g : \mathbb{N}^k \rightharpoonup \mathbb{N}$, defined by

$$g(n_1, \dots, n_k) = \min_n (f(n, n_1, \dots, n_k) \neq 0),$$

is partial recursive as well. When no n satisfies $f(n, n_1, \dots, n_k) \neq 0$ the value $g(n_1, \dots, n_k)$ is undefined.

The ***general recursive functions*** are those partial recursive functions whose domain and support coincide.

Order theory

A ***preorder*** $(P, \leq)$ is a set with a reflexive and transitive relation. A ***partially ordered set (poset)*** $(P, \leq)$ is a set with a reflexive, transitive, and anti-symmetric relation.

A function $f : P \to Q$ between posets is ***monotone*** if $x \leq y$ in P implies $f(x) \leq f(y)$ in Q.

A subset $S \subseteq P$ is an ***upper set*** if $x \in S$ and $x \leq y$ implies $y \in S$. Similarly, it is a ***lower set*** if $y \in S$ and $x \leq y$ implies $x \in S$. A subset $S \subseteq P$ of a poset $(P, \leq)$ is ***directed*** if it is non-empty and for every $x, y \in S$ there exists $z \in S$ such that $x \leq z$ and $y \leq z$. An ***upper bound*** of a subset $S \subseteq P$ in a poset is an element $x \in P$ such that $y \leq x$ for all $y \in S$. The

supremum $\sup S$ of a subset $S \subseteq P$ in a poset is its least upper bound, if it exists. More precisely, it is an upper bound x for S such that if y is also an upper bound for S then $x \leq y$.

A ***directed-complete partial order (dcpo)*** is a poset in which every directed set has a supremum. Let $(D, \leq)$ be a dcpo. For $x, y \in D$ we say that x is ***way below*** y, written $x \ll y$, when for every directed $S \subseteq D$ such that $y \leq \sup S$ there exists $z \in S$ for which $x \leq z$. An element $x \in D$ is ***compact*** (or ***finite***) when $x \ll x$. A subset $U \subseteq D$ is ***Scott open*** if it is an upper set and is inaccessible by suprema of directed sets, which means that, for every directed $S \subseteq D$, if $\sup S \in U$ then already $x \in U$ fo rsome $x \in S$. The Scott open sets form the ***Scott topology*** of D.

If D and E are dcpos then a function $f : D \to E$ is continuous with respect to the Scott topologies precisely when it preserved suprema of directed sets. It follows that such a function is monotone.

Topology

A topological space X is T_0***-space*** if each point is uniquely determined by its open neighborhoods: for all $x, y \in X$,

$$(\forall U \in \mathcal{O}(X).\, (x \in U \iff y \in U)) \Rightarrow x = y.$$

A topological space is ***zero-dimensional*** if it has a basis consisting of clopen sets.

Models of Computation 2

A *model of computation* describes what computation is and how it is done. The best known is Alan Turing's model [37] in which a machine manipulates the contents of a tape according to a finite set of instructions. It has become the yardstick with which we measure other models of computation. Turing's notion of computability is very robust. Firstly, it is robust because changes to the definition of Turing machines, such as increasing the number of tapes or heads, or allowing the head to jump around, does not change the computational power. Secondly, the notion is robust because many other models of computation turned out to be equivalent to Turing's in the sense that they can simulate Turing machines, and can be simulated by them.

[37]: Turing (1937), "On Computable Numbers, with an Application to the Entscheidungsproblem"

However, it would be wrong to conclude that other models can be safely ignored. Once a precise definition of simulation is given in Section 2.8, it will turn out that there is more to equivalence of computational models than mutual simulation.

We begin our investigations with a review of models of computation other than Turing machines. After having seen several examples, we take *(typed) partial combinatory algebras* as the common unifying notion that is well-suited for the later development. There are of course interesting models of computation that fall outside of our framework. For example, even though partical combinatory algebras can incorporate certain specific computational effects [24, 25], it is not clear how to give a systematic account of "effectful tpcas". John Longley [26] investigated more general structures that do.

[24]: Longley (1999), "Matching typed and untyped realizability"
[25]: Longley (1999), "Unifying typed and untyped realizability"

[26]: Longley (2014), "Computability structures, simulations and realizability"

2.1 Turing machines

We recall informally how a Turing machine operates. There is little point in giving a formal definition because we do not intend to actually write programs for Turing machines. If you are not familiar with Turing machines we recommend one of standard textbooks on the subject [10, 16, 28].

[10]: Davis (1958), *Computability and Unsolvability*
[16]: H.R. Rogers (1992), *Theory of Recursive Functions and Effective Computability*
[28]: Odifreddi (1989), *Classical Recursion Theory*

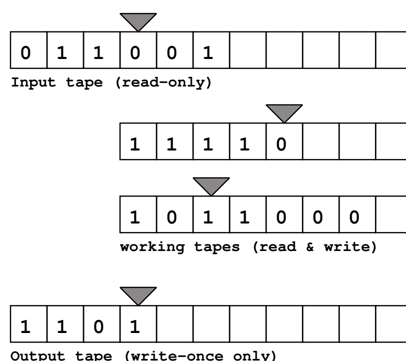


Figure 2.1: A Turing machine operates with tapes

A ***Turing machine*** is a device which operates on a number of tapes and heads, see Figure 2.1:

- a ***read-only input tape*** is equipped with a reading head that can move left and right, and read the symbols, but cannot write them.
- The ***read-write working tapes*** are equipped with heads that move left and right, and can both read and write symbols.
- The ***write-once output tape*** is equipped with a head which can move left and right, and it can write into each cell *at most once*. Once a cell is filled with a non-blank symbol all subsequent writes to it are ignored.

The tapes are infinite[1] and contain symbols from a given finite alphabet. A common choice for the alphabet is 0, 1, and a special symbol 'blank'. The machine manipulates the contents of the tapes according to a ***program***, which is a *finite* list of simple instructions that control the heads and the tapes. The machine executes one instruction at a time in a sequential manner. It may ***terminate*** after having executed finitely many computation steps. If it does not terminate then it runs forever, in which case we say that it ***diverges***.

1: If you are worried about having actual infinite tapes in your room, note that at each step of the computation only a finite portion of tapes has been inspected. In this sense the tapes are *potentially* infinite.

Our version of Turing machine is different from the usual one, where a machine is equipped with only a single tape that serves for input, output, and intermediate work. The two formulations are equivalent in the sense that a single-tape machine can simulate the workings of a Turing machine with several tapes, and vice versa. Having working tapes will ease the description of infinite computations in Subsection 2.1.2.

The state of a Turing machine may be encoded onto a single tape as follows. First we write down the program, suitably encoded by the symbols from the alphabet, then the current state (the next instruction to be executed), and positions of the heads. Finally, we copy the contents of all the tapes by interleaving them into a single tape.

If we were going to build just one machine, which one would we build? The answer was given by Turing.

Theorem 2.1.1 (Turing) *There exists a* **universal machine***: a machine that takes a description of another machine, as explained above, and simulates it.*

Proof. A traditional proof may be found in any book on computability theory, and there is nothing wrong with reading the original proof [37] either. For me a much more convincing proof is the fact that a universal machine is sitting right here on my desk. (You have to ignore the fact that several hundred gigabytes of storage are not quite the same thing as an infinite tape. Also, modern computers are really *Von Neumann* machines [13] because they have a central processing unit and random access memory instead of a tape.) □

[37]: Turing (1937), "On Computable Numbers, with an Application to the Entscheidungsproblem"

[13]: Goldstein et al. (1947), *Report on the mathematical and logical aspects of an electronic computing instrument*

Once we have a universal machine, we can make it behave like any other machine. It is just "a simple matter of programming" to tell it what to do.

We mentioned in the introduction that many kinds of computing devices are equivalent to Turing machines. We shall therefore not insist on describing computation solely in terms of Turing machines, but rather rely on familiarity with modern computers and programming languages.

After all, programs can actually be run on computers, whereas Turing machines are hard to get by.

2.1.1 Type 1 machines

How do we use Turing machines to compute a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$? A natural idea is to write the argument n onto the input tape, run the machine until it terminates, and read the result $f(n)$ off the output tape. If the machine diverges then $f(n)$ is undefined. Of course, the input n must be suitably encoded onto the input tape, for example it can be written in binary form. The output tape contains the result encoded in the same manner.

Definition 2.1.1 A partial map $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is ***computable*** if there exists a Turing machine M such that for every $n \in \mathbb{N}$: if $f(n)$ is defined then M terminates on input n and gives output $f(n)$; if $f(n)$ is undefined then M diverges on input n.

It is convenient to view *every* Turing machine as one computing a function $\mathbb{N} \rightharpoonup \mathbb{N}$. This can be arranged as long as we read the result off the output tape correctly. Suppose the alphabet contains symbols 0, 1, and blank. We encode the input n onto the input tape in binary followed by blanks, and run the machine. If and when it terminates it has written at most finally many symbols onto the output tape. Some of the symbols it has written might be different from 0 and 1. If we ignore everything that comes after the first blank, we can interpret the output tape as a number written in binary (the empty sequence encodes zero).

We can similarly define how a Turing machine computes a multivariate partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$. We just have to correctly encode the arguments on the tape by placing special markers between them so that we can tell where one ends and the next one begins.

Exercise 2.1.1 Devise a coding scheme for k-tuples of numbers using the symbols 0, 1 and blank. Make sure that every tape which contains at most finitely many 0's and 1's encodes a k-tuple of numbers.

It is common knowledge that computers encode everything with 0's and 1's, but logicians prefer to encode everything with natural numbers. We shall write in general $\ulcorner e \urcorner$ for ***encoding*** of e by a natural number. Of course, we must specify what $\ulcorner e \urcorner$ is in each particular case. For example, a pair of numbers (m, n) may be encoded into a single number as

$$\ulcorner (m, n) \urcorner = 2^m (2n + 1).$$

Every number except 0 represents the code of a unique pair so we also have computable ***projections*** `fst` and `snd` which recover m and n from $\ulcorner (m, n) \urcorner$, respectively. Once we know how to encode pairs of numbers, lists of numbers can be encoded as iterated pairs:

$$\begin{aligned} \ulcorner [\,] \urcorner &= 0 \\ \ulcorner [n_0, \ldots, n_k] \urcorner &= \ulcorner (n_0, \ulcorner [n_1, \ldots, n_k] \urcorner) \urcorner. \end{aligned}$$

Because we defined $\ulcorner(m, n)\urcorner$ so that it is never zero, the elements of a sequence may be uniquely reconstructed from its code. By iterating the scheme we may encode lists of lists of numbers, etc.

Turing machines can be encoded with numbers, also. A program is a finite list of instructions, so it can be encoded as a finite sequence of 0's and 1's (your computer does this every time you save a piece of source code in a file), which in turn represents a number in binary form. In fact, every number may be thought of as a code of a program by the reverse process. Given a number, write it in binary form and interpret it as a sequence of 0's and 1's and decode from it a list of instructions. It may happen that the binary sequence does not properly encode a list of instructions, in which case we interpret it as some fixed silly little program.

The next step is to encode tapes and entire computations with numbers. Because an infinite tape cannot be encoded in a single natural number, we limit attention to the so-called ***type 1 machines*** which accept only *finite* inputs. More precisely, the input always consists of a finite string of 0's and 1's followed by blanks. Such input may be encoded by a single number. Furthermore, at every step of computation the machine has used up only a finite portion of its working tapes, whose contents may again be encoded by a single number.

By continuing in this manner we may encode with a single number a finite sequence of computation steps, including the contents of the tapes and positions of the heads at each step. Stephen Kleene [19] worked out the details of all this and defined the predicate $T(x, y, z)$ whose meaning is

[19]: Kleene (1943), "Recursive predicates and quantifiers"

> "Machine encoded by x with input tape that encodes the number y performs a sequence of computation steps encoded by z and terminates."

The amazing thing is that T may be defined in Peano arithmetic just in terms of 0, successor, + and ×. There is an associated computable partial function $U(z)$ whose meaning is "the number encoded by the contents of the output tape in the last step of computation encoded by z". With it we can extract the result of a computation. It is easy to arrange U so that it is defined for all z, even those that do not encode terminating computations.

Kleene's normal form theorem [19] says that every partial computable function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ may be written in the form

[19]: Kleene (1943), "Recursive predicates and quantifiers"

$$y \mapsto U(\min\{z \in \mathbb{N} \mid T(x, y, z)\}). \tag{2.1}$$

The number x is the encoding of a machine that computes f. We emphasize that we are completely ignoring questions of computational efficiency. Just consider how we would compute $f(y)$ according to (2.1): for each $z = 0, 1, 2, \ldots$, test whether z encodes a computation of machine x with input y. When you find the first such z, extract the result $U(z)$ from it. I dare you to compute the identity function $y \mapsto y$ this way!

Kleene's normal form may be used to define a standard enumeration of

partial recursive functions. Let

$$\varphi_x(y) = U(\min\{z \in \mathbb{N} \mid T(x, y, z)\}).$$

The sequence $\varphi_0, \varphi_1, \varphi_2, \ldots$ is an enumeration of all computable partial functions (with repetitions).

The preceding discussion may be generalized to functions of several variables. For each $k \in \mathbb{N}$ there is Kleene's predicate $T^{(k)}(x, y_1, \ldots, y_k, z)$ and the corresponding $U^{(k)}(z)$ that extracts results from computations. Similarly, there is a standard enumeration of k-place computable partial functions

$$\varphi_x^{(k)}(y_1, \ldots, y_k) = U^{(k)}(\min\{z \in \mathbb{N} \mid T^{(k)}(x, y_1, \ldots, y_k, z)\}).$$

These enumerations are not arbitrary, because they have the following important properties.

Theorem 2.1.2 (utm) *There exists a partial computable function $u : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ such that, for all $x, y \in \mathbb{N}$,*

$$u(x, y) \simeq \varphi_x(y).$$

Theorem 2.1.3 (smn) *There exists a computable function $s : \mathbb{N}^2 \to \mathbb{N}$ such that, for all $x, y, z \in \mathbb{N}$,*

$$\varphi_{s(x,y)}(z) \simeq \varphi_x^{(2)}(y, z).$$

The utm theorem is essentially a restatement of Theorem 2.1.1 in terms of computable partial functions. Detailed proofs of the utm and smn theorems would involve a lot of technical manipulations of Turing machines, you may consult [10] to get a taste of it. It is illuminating to see how the utm and smn theorems manifest themselves in modern programming languages, say in Haskell. Keeping in mind that numbers are just codes for programs and data, the universal function u from the utm theorem is

[10]: Davis (1958), *Computability and Unsolvability*

```
u (f, y) = f y
```

and the function s is the currying operation[2]

```
s (f, y) = \z -> f (y, z)
```

2: In Haskell the notation `\x -> e` stands for λ-abstraction $\lambda x.\, e$, which in turn means "the function which maps x to e", see Section 2.3.

This may seem like a triviality to the programmer but is surely not considered one by the implementors of the Haskell compiler. The definition of s uses function application, pairing, currying and λ-abstraction, which are "the essence" of functional programming, just like the utm and smn theorems are the essence of partial computable functions.

The following theorem is important in the theory of computable functions because it allows us to define partial computable functions by recursion.

Theorem 2.1.4 (Recursion theorem) *For every* total *computable $f : \mathbb{N} \to \mathbb{N}$ there exists $n \in \mathbb{N}$ such that $\varphi_{f(n)} = \varphi_n$.*

Proof. The classical proof goes as follows. First we define a computable partial map $\psi : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ such that

$$\psi(u, x) = \boldsymbol{\varphi}_{\varphi_u(u)}(x),$$

where $\psi(u, x)$ is undefined when $\boldsymbol{\varphi}_u(u)$ is undefined. By the smn theorem there is a computable function $g : \mathbb{N} \to \mathbb{N}$ such that $\boldsymbol{\varphi}_{g(u)}(x) = \psi(u, x)$. Now consider any computable $f : \mathbb{N} \to \mathbb{N}$. Because $f \circ g$ is computable, there exists $v \in \mathbb{N}$ such that $\boldsymbol{\varphi}_v = f \circ g$. Since $f \circ g$ is a total function, $\boldsymbol{\varphi}_v(v)$ is defined. The number $n = g(v)$ has the desired property: $\boldsymbol{\varphi}_{g(v)} = \boldsymbol{\varphi}_{\varphi_v(v)} = \boldsymbol{\varphi}_{f(g(v))} = \boldsymbol{\varphi}_{f(n)}$. □

This was a typical argument in the theory of computable functions. Let us prove the recursion theorem in Haskell to see what is going on. The function $f : \mathbb{N} \to \mathbb{N}$ operates on codes of computable partial maps. Haskell has higher-order functions that work directly with functions as arguments and results, so f should be given the type

```
f :: (Integer -> Integer) -> (Integer -> Integer)
```

Rather than looking for a number n we are looking for a function

```
n :: Integer -> Integer
```

such that `f n = n`. Because Haskell already has recursion built in this is very easy, just define

```
n = f n
```

The Recursion theorem is nothing but definition by recursion for type 1 machines.

We finish this section with a theorem which we shall often use to show *non*-computability results.

Theorem 2.1.5 (Halting oracle) *The* **halting oracle***,*

$$h(x) = \begin{cases} 1 & \textit{if } \boldsymbol{\varphi}_x(0) \textit{ is defined,} \\ 0 & \textit{if } \boldsymbol{\varphi}_x(0) \textit{ is not defined,} \end{cases}$$

is not *computable.*

Proof. Let us prove the theorem in Haskell. We must show that there is no

```
h :: (Integer -> Integer) -> Integer
```

such that, for all `f :: Integer -> Integer`,

$$\texttt{h f} = \begin{cases} \texttt{1} & \text{if } \texttt{f 0} \text{ terminates,} \\ \texttt{0} & \text{if } \texttt{f 0} \text{ diverges.} \end{cases}$$

Suppose there were such an `h`. Define

```
g n = if h g == 1 then g n else 0
```

By assumption `h g` is either `0` or `1`. In either case there is a contradiction because `g` does just the opposite of what `h` says it will do. □

2.1.2 Type 2 machines

Type 1 machines from previous section only operate on finite inputs. In practice we often see programs whose input and output are (potentially) infinite. For example, when you listen to an Internet radio station, the player accepts a never-ending stream of data which it outputs to the speakers. Also, many useful programs, such as servers, operating systems, and browsers are potentially non-terminating. We therefore need a model of computation that describes non-terminating programs with infinite inputs and outputs.

A popular one is ***type 2 machine***, which accepts an infinite sequence on its input tape and is allowed to work forever. It may or may not fill the output tape entirely with non-blank symbols. Note that the requirement for the output tape to be write-once makes it possible to tell when the machine has actually produced an output in a given cell. Had we allowed the machine to write to each output cell many times, it could keep coming back and changing what it has already written.

An important distinction between type 1 and type 2 machines is that the latter may accept non-computable inputs, from which non-computable outputs may be produced.

For type 2 machines there are analogues of the standard enumeration $\boldsymbol{\varphi}$, utm and the smn theorems. These are more easily expressed if we allow the machines to write natural numbers in the cells, rather than symbols from a finite alphabet. We also equip the machines with instructions for manipulating numbers, say, instructions for extracting the bits and for testing equality with zero. These changes are inessential because an infinite sequence $n_0, n_1, n_2, \ldots$ of natural numbers may be encoded as a binary sequence $1^{n_0}01^{n_1}01^{n_2}0\cdots$, where 1^k means that the symbol 1 is repeated k-times.

Definition 2.1.2 Say that a type 2 machine M **computes** a partial map $f : \mathbb{N}^\mathbb{N} \rightharpoonup \mathbb{N}^\mathbb{N}$ when for every $\alpha \in \mathbb{N}^\mathbb{N}$: if $f(\alpha)$ is defined then M eventually writes to every output cell, and the output tape equals $f(\alpha)$; if $f(\alpha)$ is not defined, then at least one output cell to which M never writes to.

A partial map $f : \mathbb{N}^\mathbb{N} \rightharpoonup \mathbb{N}^\mathbb{N}$ is **type 2 computable** if it is computed by a type 2 machine.

We may similarly define what it means for a machine to compute a multivariate partial function $f : (\mathbb{N}^\mathbb{N})^k \rightharpoonup \mathbb{N}^\mathbb{N}$. The input $(\alpha_0, \ldots, \alpha_{k-1})$ is written onto the input tape in an interleaving manner, so that $\alpha_i(j)$ is found in the cell at position $k \cdot j + i$.

The Baire space

We recall a few basic facts about the ***Baire space*** $\mathbb{B} = \mathbb{N}^\mathbb{N}$. Let $\mathbb{N}^*$ be the set of all finite sequences of natural numbers. If $a, b \in \mathbb{N}^*$ we write $a \sqsubseteq b$ when a is a prefix of b. The length of a finite sequence a is denoted by $\|a\|$. Similarly, we write $a \sqsubseteq \alpha$ when a is a prefix of an infinite sequence $\alpha \in \mathbb{B}$. Define $\overline{a}(n) = [\alpha(0), \ldots, \alpha(n-1)]$ to be the prefix of α consisting of the first n terms.

Write $n{::}\alpha$ for the sequence $n, \alpha(0), \alpha(1), \alpha(2), \ldots$, and $a \mathbin{+\!\!+} \beta$ for concatenation of the finite sequence $a \in \mathbb{N}^*$ with the infinite sequence $\beta \in \mathbb{B}$.

We equip $\mathbb{B}$ with the product topology, which is the topology whose countable topological base consists of the basic open sets, for $a \in \mathbb{N}^*$,

$$a{+\!\!+}\mathbb{B} = \{a \mathbin{+\!\!+} \beta \mid \beta \in \mathbb{B}\} = \{\alpha \in \mathbb{B} \mid a \sqsubseteq \alpha\}.$$

Because the basic open sets are both closed and open (clopen), $\mathbb{B}$ is in fact a countably based 0-dimensional[3] Hausdorff space. It is also a complete separable metric space for the ***comparison metric*** $d : \mathbb{B}\times\mathbb{B} \to \mathbb{R}$, defined by

$$d(\alpha, \beta) = \inf \{2^{-n} \mid \overline{\alpha}(n) = \overline{\beta}(n)\}.$$

3: Recall that a space is 0-***dimensional*** when its clopen subsets form a base for its topology.

If the first term in which α and β differ is the n-th one, then $d(\alpha, \beta) = 2^{-n}$. The comparison metric is an ***ultrametric***, which means that it satisfies the inequality $d(\alpha, \gamma) \leq \max(d(\alpha, \beta), d(\beta, \gamma))$. In an ultrametric space every point of a ball is its center. The clopen sets $a{+\!\!+}\mathbb{B}$ are precisely the balls of radius $2^{-\|a\|}$.

Encoding of partial maps $\mathbb{B} \rightharpoonup \mathbb{B}$

Earlier we encoded the partial computable maps $\mathbb{N} \rightharpoonup \mathbb{N}$ with numbers describing Turing machines. We would like to similarly encode computable partial maps $\mathbb{B} \rightharpoonup \mathbb{B}$ with sequences. An obvious idea, which we shall not pursue, is to use the first term of a sequence to encode a Turing machine. Instead, we shall use sequences as "lookup tables", with which even some non-computable maps will be encoded.

Consider a partial map $f : \mathbb{B} \rightharpoonup \mathbb{B}$ computed by M. Suppose that, given an input tape

$$\alpha = n_0, n_1, n_2, \ldots$$

M writes j to the i-th output cell after k steps of computation, hence $f(\alpha)(i) = j$. Because in k steps M inspects at most the first k input cells, it would have done the same for any other input that agrees with the given one in the first k terms. Thus f is determined by pieces of information of the form:

> "If the input starts with $n_0, n_1, \ldots, n_{k-1}$ then the i-th term of the output is j."

A code $\gamma \in \mathbb{B}$ of f just has to contain such information, which we can arrange by coding lists and pairs as numbers. To determine the value $\gamma(m)$ for a given $m \in \mathbb{N}$, decode $m + 1$ as a finite sequence $m + 1 = [i, n_0, \ldots, n_{k-1}]$ and simulate M for k steps with the input tape

$$n_0, \ldots, n_{k-1}, 0, 0, \ldots$$

If the machine writes j into the i-th output cell during the simulation, set $\gamma(m) = j + 1$, otherwise set $\gamma(m) = 0$. Think of γ as a lookup table which maps a key $i{::}[n_0, \ldots, n_{k-}]$ to an optional value j. Clearly, γ can be computed from a description of M.

To decode the function $\boldsymbol{\eta}_\gamma : \mathbb{B} \rightharpoonup \mathbb{B}$ encoded by $\gamma \in \mathbb{B}$, we just have to devise a lookup procedure. Given input $\alpha \in \mathbb{B}$, we compute the value

of the i-th output cell by successively looking up keys $i{::}\overline{\alpha}(k)$ for ever larger k, until we find an answer j. More precisely, for $\alpha \in \mathbb{B}$, define $\ell(\gamma, \alpha) : \mathbb{N} \rightharpoonup \mathbb{N}$ as

$$\ell(\gamma, \alpha)(i) = \gamma(\ulcorner i{::}\overline{\alpha}(k)\urcorner) - 1 \quad \text{where} \quad k = \min_k(\gamma(\ulcorner i{::}\overline{\alpha}(k)\urcorner) \neq 0)$$

(if not such k exists then $\ell(\gamma, \alpha)(i)$ is undefined), and let the map $\boldsymbol{\eta}_\gamma : \mathbb{B} \rightharpoonup \mathbb{B}$ encoded by γ be

$$\boldsymbol{\eta}_\gamma(\alpha) = \begin{cases} \ell(\gamma, \alpha) & \text{if } \ell(\gamma, \alpha) \text{ is a total map,} \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{2.2}$$

Definition 2.1.3 A partial function $f : \mathbb{B} \rightharpoonup \mathbb{B}$ is ***(type 2) realized*** by $\gamma \in \mathbb{B}$, called a ***Kleene associate*** of f, when $f = \boldsymbol{\eta}_\gamma$.

Clearly, if γ is a computable sequence then $\boldsymbol{\eta}_\gamma$ is type 2 computable. But what sort of partial map is $\boldsymbol{\eta}_\gamma$ in general?

An arbitrary $\gamma \in \mathbb{B}$ may give inconsistent answers, in the sense that looking up values at $i{::}\overline{\alpha}(k)$ and $i{::}\overline{\alpha}(m)$ may give inconsistent answers. Above we resolved the problem by taking the answer given by the least k. We may also rectify γ to a realizer that gives consistent answers.

Say that $\gamma \in \mathbb{B}$ is ***consistent*** when, for all $k, m \in \mathbb{N}$ and $\alpha \in \mathbb{B}$, if $k < m$ and $\gamma(\ulcorner\overline{\alpha}(k)\urcorner) \neq 0$ then $\gamma(\ulcorner\overline{\alpha}(m)\urcorner) = \gamma(\ulcorner\overline{\alpha}(k)\urcorner)$.

Lemma 2.1.6 *Every realized function is realized by a consistent realizer.*

Proof. Suppose $f : \mathbb{B} \rightharpoonup \mathbb{B}$ is realized by γ. Define δ by induction on the length of $a \in \mathbb{N}^\star$ by

$$\delta(\ulcorner a\urcorner) = \begin{cases} \gamma(\ulcorner a\urcorner) & \text{if } a = [\,], \\ \gamma(\ulcorner a\urcorner) & \text{if } a = i{::}a' \text{ and } \delta(\ulcorner a'\urcorner) = 0, \\ \delta(\ulcorner a'\urcorner) & \text{if } a = i{::}a' \text{ and } \delta(\ulcorner a'\urcorner) \neq 0, \end{cases}$$

The realizer δ is consistent by construction. It is easy to check that $\boldsymbol{\eta}_\gamma = \boldsymbol{\eta}_\delta$. □

Theorem 2.1.7 (Extension Theorem for $\mathbb{B}$) *Every partial continuous map $f : \mathbb{B} \rightharpoonup \mathbb{B}$ can be extended to a realized one.*

Proof. Suppose $f : \mathbb{B} \rightharpoonup \mathbb{B}$ is a partial continuous map, and let

$$A = \{(a, i, j) \in \mathbb{N}^* \times \mathbb{N}^2 \mid \\ a{+\!\!+}\mathbb{B} \cap \mathsf{dom}(f) \neq \emptyset \wedge \forall \alpha.\, (a{+\!\!+}\mathbb{B} \cap \mathsf{dom}(f)) f(\alpha)(i) = j\}.$$

If $(a, i, j) \in A$ and $(a', i, j') \in A$ and $a \sqsubseteq a'$ then $j = j'$ because there is $\alpha \in a'{+\!\!+}\mathbb{B} \cap \mathsf{dom}(f) \subseteq a{+\!\!+}\mathbb{B} \cap \mathsf{dom}(f)$ such that $j = f(\alpha)(i) = j'$. We define a sequence $\gamma \in \mathbb{B}$ as follows. For every $(a, i, j) \in A$ let $\gamma(\ulcorner i{::}a\urcorner) = j + 1$, and for all other arguments n let $\gamma(n) = 0$. Suppose

that $\gamma(\ulcorner i{::}a \urcorner) = j + 1$ for some $i, j \in \mathbb{N}$ and $a \in \mathbb{N}^*$. Then for every prefix $a' \sqsubseteq a$, $\gamma(\ulcorner i{::}a' \urcorner) = 0$ or $\gamma(\ulcorner i{::}a' \urcorner) = j+1$. Thus, if $(a, i, j) \in A$ and $a \sqsubseteq \alpha$ then $\boldsymbol{\eta}_\gamma(\alpha)(i) = j$. Let us how that $\boldsymbol{\eta}_\gamma(\alpha)(i) = f(\alpha)(i)$ for all $\alpha \in \mathsf{dom}(f)$ and all $i \in \mathbb{N}$. Because f is continuous, for all $\alpha \in \mathsf{dom}(f)$ and $i \in \mathbb{N}$ there exists $(a, i, j) \in A$ such that $a \sqsubseteq \alpha$ and $f(\alpha)(i) = j$. Now we get $\boldsymbol{\eta}_\gamma(\alpha)(i) = j = f(\alpha)(i)$. □

Recall that a G_δ***-set*** is a countable intersection of open sets.

Lemma 2.1.8 *If $U \subseteq \mathbb{B}$ is a G_δ-set then the partial function $u : \mathbb{B} \rightharpoonup \mathbb{B}$, defined by*

$$u(\alpha) = \begin{cases} 1 & \text{if } \alpha \in U. \\ \text{undefined} & \text{otherwise,} \end{cases}$$

is realized.

Proof. The set U may be expressed as a countable intersection of countable unions of basic open sets,

$$U = \bigcap_{i \in \mathbb{N}} \bigcup_{j \in \mathbb{N}} a_{i,j} +\!\!+ \mathbb{B}.$$

Define $\gamma \in \mathbb{B}$ by setting $\gamma(\ulcorner i{::}a_{i,j} \urcorner) = 2$ for all $i, j \in \mathbb{N}$, and $\gamma(n) = 0$ for all other arguments n. Clearly, if $\boldsymbol{\eta}_\gamma(\alpha)$ is defined then its value is the constant sequence $1, 1, 1, \ldots$, so we only need to verify that $\mathsf{dom}(\boldsymbol{\eta}_\gamma) = U$. If $\alpha \in \mathsf{dom}(\boldsymbol{\eta}_\gamma)$ then $\ell(\gamma, \alpha)(i)$ is defined for every $i \in \mathbb{N}$. For every $i \in \mathbb{N}$ there exists $j_i \in \mathbb{N}$ such that $\gamma(\ulcorner i{::}\overline{\alpha}(j_i) \urcorner) = 2$, which implies that $a_{i,j_i} \sqsubseteq \alpha$. Hence

$$\alpha \in \bigcap_{i \in \mathbb{N}} a_{i,j_i} +\!\!+ \mathbb{B} \subseteq U.$$

Conversely, suppose $\alpha \in U$ and consider any $i \in \mathbb{N}$. There exists $j \in \mathbb{N}$ such that $a_{i,j} \sqsubseteq \alpha$, therefore $\gamma(\ulcorner i{::}\overline{\alpha}(|a_{i,j}|) \urcorner) = \gamma(\ulcorner i{::}a_{i,j} \urcorner) = 2$ so that $\ell(\gamma, \alpha)(i)$ is defined. We conclude that $\alpha \in \mathsf{dom}(\boldsymbol{\eta}_\gamma)$. □

Lemma 2.1.9 *Suppose $\alpha \in \mathbb{B}$ and $U \subseteq \mathbb{B}$ is a G_δ-set. Then there exists $\delta \in \mathbb{B}$ such that $\mathsf{dom}(\boldsymbol{\eta}_\delta) = U \cap \mathsf{dom}(\boldsymbol{\eta}_\alpha)$ and $\boldsymbol{\eta}_\alpha(\beta) = \boldsymbol{\eta}_\delta(\beta)$ for all $\beta \in \mathsf{dom}(\boldsymbol{\eta}_\alpha) \cap U$.*

Proof. By Lemma 2.1.6 we may assume that α is normalized. Define $f : \mathbb{B} \rightharpoonup \mathbb{B}$ by

$$f(\beta)(n) = \begin{cases} \boldsymbol{\eta}_\alpha(\beta)(n) & \text{if } \beta \in \mathsf{dom}(\boldsymbol{\eta}_\alpha) \cap U, \\ \text{undefined} & \text{otherwise} \end{cases}$$

We would like to show that f is realized. From Lemma 2.1.8 we obtain a normalized $\gamma \in \mathbb{B}$ such that, for all $\beta \in \mathbb{B}$,

$$\boldsymbol{\eta}_\gamma(\beta) = \begin{cases} \lambda n.\, 1 & \beta \in U, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We claim that f is realized by

$$\delta(k) = \alpha(k) \cdot \gamma(k)/2.$$

Recall that $\gamma(k)$ is either 0 or 2 so $\delta(k)$ is either 0 or $\alpha(k)$. Hence $\boldsymbol{\eta}_\delta$ is a restriction of $\boldsymbol{\eta}_\alpha$, by which we mean that $\mathsf{dom}(\boldsymbol{\eta}_\delta) \subseteq \mathsf{dom}(\boldsymbol{\eta}_\alpha)$ and $\boldsymbol{\eta}_\delta(\beta) = \boldsymbol{\eta}_\alpha(\beta)$ for all $\beta \in \mathsf{dom}(\boldsymbol{\eta}_\delta)$. Also, $\mathsf{dom}(\boldsymbol{\eta}_\delta) \subseteq \mathsf{dom}(\boldsymbol{\eta}_\gamma)$ because $\delta(k) \neq 0$ implies $\gamma(k) \neq 0$. It remains to be shown that $\beta \in \mathsf{dom}(\boldsymbol{\eta}_\alpha) \cap U$ implies $\beta \in \mathsf{dom}(\boldsymbol{\eta}_\delta)$, i.e., that for such β, $\ell(\delta, \beta)(i)$ is defined for every $i \in \mathbb{N}$. Because $\ell(\alpha, \beta)(i)$ and $\ell(\gamma, \beta)(i)$ are defined, there exist k_1 and k_2 such that

$$\alpha(\ulcorner i{::}\overline{\beta}(k_1)\urcorner) \neq 0 \qquad \text{and} \qquad \gamma(\ulcorner i{::}\overline{\beta}(k_2)\urcorner) \neq 0.$$

Because α and γ are normalized, for $k = \max(k_1, k_2)$,

$$\alpha(\ulcorner i{::}\overline{\beta}(k)\urcorner) \neq 0 \qquad \text{and} \qquad \gamma(\ulcorner i{::}\overline{\beta}(k)\urcorner) \neq 0,$$

hence $\delta(\ulcorner i{::}\overline{\beta}(k)\urcorner) \neq 0$, which we wanted to show. □

We are now able to characterize the realized maps.

Theorem 2.1.10 *A partial function $f : \mathbb{B} \rightharpoonup \mathbb{B}$ is realized if, and only if, f is continuous and its support is a G_δ-set.*

Proof. First we show that $\boldsymbol{\eta}_\alpha$ is a continuous map whose support is a G_δ-set. It is continuous because the value of $\boldsymbol{\eta}_\alpha(\beta)(n)$ depends only on n and a finite prefix of β. The support of $\boldsymbol{\eta}_\alpha$ is the G_δ-set

$$\begin{aligned}\mathsf{dom}(\boldsymbol{\eta}_\alpha) &= \{\beta \in \mathbb{B} \mid \forall n.\, \mathbb{N}\ell(\alpha,\beta)(n) \text{ defined and } > 0\} \\ &= \textstyle\bigcap_{n\in\mathbb{N}} \{\beta \in \mathbb{B} \mid \ell(\alpha,\beta)(n) \text{ defined and } > 0\} \\ &= \textstyle\bigcap_{n\in\mathbb{N}} \bigcup_{m\in\mathbb{N}} \{\beta \in \mathbb{B} \mid \ell(\alpha,\beta)(n) = m+1\}.\end{aligned}$$

Each of the sets $\{\beta \in \mathbb{B} \mid \ell(\alpha,\beta)(n) = m\}$ is open because ℓ is a continuous operation.

Now let $f : \mathbb{B} \rightharpoonup \mathbb{B}$ be a partial continuous function whose support is a G_δ-set. By Extension Theorem 2.1.7 there exists $\gamma \in \mathbb{B}$ such that $f(\alpha) = \boldsymbol{\eta}_\gamma(\alpha)$ for all $\alpha \in \mathsf{dom}(f)$. By Lemma 2.1.9 there exists $\psi \in \mathbb{B}$ such that $\mathsf{dom}(\boldsymbol{\eta}_\psi) = \mathsf{dom}(f)$ and $\boldsymbol{\eta}_\psi(\alpha) = \boldsymbol{\eta}_\gamma(\alpha)$ for every $\alpha \in \mathsf{dom}(f)$. □

Finally, we formulate the utm and smn theorems for type 2 machines.

Theorem 2.1.11 (type 2 utm) *There exists a computable partial function $u : \mathbb{B} \times \mathbb{B} \rightharpoonup \mathbb{B}$ such that $u(\alpha, \beta) \simeq \boldsymbol{\eta}_\alpha(\beta)$ for all $\alpha, \beta \in \mathbb{B}$.*

Proof. Let us write a machine for computing u in Haskell, but without resorting to an explicit encoding of finite sequences by numbers. Define the type

```
type Baire = Integer -> Integer
```

The universal `u :: ([Integer] -> Integer, Baire) -> Baire` is just the transliteration of (2.2):

```
u (a, b) i = x - 1
    where x = head $
              filter (/= 0) $
              [a (i : map b [0..(k-1)]) | k <- [0..]]
```

You may entertain yourself by learning Haskell and figuring out how it works. □

The type 2 variant of the smn theorem uses the representation $\boldsymbol{\eta}^{(2)}$ for encoding partial maps $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ by

$$\boldsymbol{\eta}^{(2)}_{\alpha}(\beta, \gamma) = \boldsymbol{\eta}_{\alpha}(\langle \beta, \gamma \rangle)$$

where $\langle \beta, \gamma \rangle$ is the interleaved sequence $\beta(0), \gamma(0), \beta(1), \gamma(1), \ldots$.

Theorem 2.1.12 (type 2 smn) *There exists a computable $s : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ such that, for all $\alpha, \beta, \gamma \in \mathbb{B}$,*

$$\boldsymbol{\eta}_{s(\alpha,\beta)}(\gamma) = \boldsymbol{\eta}^{(2)}_{\alpha}(\beta, \gamma).$$

Proof. Exercise in Haskell programming. □

2.1.3 Turing machines with oracles

The computational models from Subsection 2.1.12.1.2 *relativize*, i.e., they may be adapted to use ***oracle Turing machines***. Recall that an ***oracle*** is an infinite binary sequence $\omega : \mathbb{N} \to \{0, 1\}$ and that a Turing machine with oracle ω is a Turing machine with an extra read-only tape containing ω. The machine consults the tape to obtain the values of ω. When ω is a non-computable sequence, the oracle machine exceedds the computational power of ordinary Turing machines.

Each oracle ω yields a type 1 and a type 2 model of computation in which the machines have access to ω. That is, in any given such model all machines access the same fixed oracle ω.

2.1.4 Hamkin's infinite-time Turing machines

Another model of computation that exceeds the power of Turing machines are Hamkin's ***infinite-time Turing machines*** [15]. We give here just a brief overview and recommend the cited reference for background reading.

[15]: Hamkins et al. (2000), "Infinite time Turing machines"

An infinite-time Turing machine, or just *machine*, is like a Turing machine which is allowed to run infinitely long, with the computation steps counted by ordinal numbers. The machine has a finite program, an input tape, work tapes, an output tape, etc. We assume that the tape cells contain 0's and 1's. At successor ordinals the machine acts like an ordinary Turing machine. At limit ordinals it enters a special "limit" state, its heads are placed at the beginnings of the tapes, and the content of each tape cell is computed as the lim sup of the values written in the cell

at earlier stages. More precisely, if c_α denotes the value of the cell c at step α, then for a limit ordinal β we have

$$c_\beta = \begin{cases} 0 & \text{if } \exists \alpha < \beta .\, \forall \gamma .\, (\alpha \leq \gamma < \beta \Rightarrow c_\alpha = 0), \\ 1 & \text{otherwise.} \end{cases}$$

The machine terminates by entering a special halt state, or it may run forever. It turns out that a machine which has not terminated by step ω_1 runs forever.

We can think of machines as computing partial functions $2^{\mathbb{N}} \rightharpoonup 2^{\mathbb{N}}$ where $2 = \{0, 1\}$: initialize the input tape with an infinite binary sequence $x \in 2^{\mathbb{N}}$, run the machine, and observe the contents of the output tape if and when the machine terminates. We can also consider infinite time computation of partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$: initialize the input tape with the input number, run the machine, and interpret the contents of the output tape as a natural number, where we ignore anything that is beyond the position of the output head. By performing the usual encoding tricks, we can feed the machines more complicated inputs and outputs, such as pairs, finite lists, and even infinite lists of numbers. We say that a function is ***infinite-time computable*** if there is an infinite-time Turing machine that computes it.

The power of infinite-time Turing machines is vast and extends far beyond the halting problem for ordinary Turing machines, although of course they cannot solve their own halting problem. For example, for every Π^1_1-subset $S \subseteq 2^{\mathbb{N}}$ there is a machine which, given $x \in 2^{\mathbb{N}}$ on its input tape, terminates and decides whether $x \in S$.

There is a standard enumeration $M_0, M_1, M_2, \ldots$ of infinite-time Turing machines, where M_n is the machine whose program is encoded by the number n in some reasonable manner. The associated enumeration $\psi_0, \psi_1, \psi_2, \ldots$ of infinite-time computable partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$ is defines as

$$\psi_n(k) = \begin{cases} m & \text{if } M_n(k) \text{ terminates and outputs } m, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We may similarly an enumeration $\psi^{(k)}_0, \psi^{(k)}_1, \psi^{(k)}_2, \ldots$ of k-ary infinite-time computable partial maps $\mathbb{N}^k \rightharpoonup \mathbb{N}$.

The enumeration satisfies the smn and utm theorems.

Theorem 2.1.13 (infinite-time smn) *There is an infinite time computable total map* $s : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *such that* $\psi_{s(m,n)}(j) = \psi^{(2)}_m(n, j)$ *for all* $m, n, j \in \mathbb{N}$.

Theorem 2.1.14 (infinite-time utm) *There is an infinite-time computable partial map* $u : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ *such that* $\psi_t(m) = u(t, m)$ *for all* $t, m \in \mathbb{N}$.

To convince ourselves that the utm theorem holds, we think a bit how a universal infinite-time Turing machine works. It accepts the description n of a machine and the initial input tape x. At successor steps the simulation of machine M_n on input x proceeds much like it does for the ordinary Turing machines. Thus it takes finitely many successor steps to simulate

one successor step of M_n. Each limit step of M_n is simulated by one limit step of the universal machine, followed by finitely many successor steps. Indeed, whenever the universal machine finds itself in the special limit state, it puts the simulated machine in the simulated limit state, and moves the simulated heads to the beginnings of the simulated tapes. These actions take finitely many steps. The contents of the simulated tapes need not be worried about, as it will be updated correctly at limit stages.

To see what sort of tasks can be performed by infinite-time Turing machines, we consider several examples that will be useful later on.

Example 2.1.1 There is a machine which decides whether two infinite sequences $x, y \in 2^{\mathbb{N}}$ are equal. It first initializes a fresh work cell with 0, and then for each k, it compares $x(k)$ and $y(k)$. If they differ, it sets the work cell to 1. After ω steps the work cell will be 1 if, and only if, $x \neq y$.

Example 2.1.2 A more complicated problem is to *semi*decide whether a given machine M_n computes a given sequence $x \in 2^{\mathbb{N}}$. The machine which performs such a task accepts n and x as inputs and begins by writing down the sequence $y_k = \psi_n(k)$ onto a work tape. This it can do by simulating M_n successively on inputs $0, 1, 2, \ldots$ and writing down the values y_k as they are obtained. The machine also keeps track for which k's the values y_k have been computed by flipping the k-th bit of a separate "tally" tape from 0 to 1 whenever $M_n(k)$ terminates. If any of the y_k's is undefined, the machine will run forever. Otherwise it will be able to detect in ω steps that the entire sequence y has been computed and written down by checking that all bits on the separate "tally" tape have been flipped to 1. After that, the machine verifies that $x_k = y_k$ for all $k \in \mathbb{N}$, as described previously.

Example 2.1.3 Suppose we have a machine M which expects as input an infinite sequence x and a number n. We would like to construct another machine which accepts an infinite sequence x and outputs a number n such that $M(x, n)$ terminates, if one exists. We use the familiar dovetailing technique to tackle the problem. Given $x \in 2^{\mathbb{N}}$ as input, we simulate in parallel the executions of machine M on inputs of the form (x, n), one for each n:

$$M(x, 0), \quad M(x, 1), \quad M(x, 2), \quad \ldots$$

Each of these requires several infinite tapes, but since we only need countably many of them, they may be interleaved into a single tape. At successor steps the simulation performs the usual dovetailing technique. At limit steps the simulation inserts extra ω bookkeeping steps, during which it places the simulated machines in the "limit" state and moves their head positions. The extra steps do not ruin the limits of the simulated tapes, because those are left untouched. After the extra steps are performed, dovetailing starts over again. As soon as one of the simulations $M(x, n)$ terminates, we return the results n. Note that n is computed from x in a deterministic fashion (that depends

on the details of the dovetailing and simulation).

2.2 Scott's graph model

A model of computation may introduce features which are not easily detected, until we compare it with other models. For example, the innocuous looking idea that the input be stored on a tape gives a type 2 machine the ability to take into account the *order* in which data appear. In this section we consider a different model of infinite computation, the *graph model*, introduced by Dana Scott [34], in which we use sets of numbers rather than sequences.

[34]: Scott (1976), "Data Types as Lattices"

How is computation of a map $f : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ on the powerset of natural numbers to be performed? One natural idea would be to use computation with respect to an oracle: f is computable if there is a Turing machine which computes $f(A)$ when given A as an oracle, i.e., it may test membership in A. However, this is still just type 2 computability in disguise, because asking an oracle whether a number belongs to A is equivalent to having an infinite input tape with a 1 in the n-th cell when $n \in A$, and a 0 otherwise.

An oracle provides both *positive* and *negative* information about membership in A. In contrast, Scott's graph model operates only with positive information. Rather than describing it explicitly as a kind of Turing machines, we shall take a different route this time and first describe the topological aspects of the model. Computability will then follow naturally.

The set $\mathcal{P}(\mathbb{N})$ may be equipped with a topology in two natural ways. One is the product topology arising from the observation that $\mathcal{P}(\mathbb{N})$ is isomorphic to the countable product $2^{\mathbb{N}}$. This topology encodes positive and negative information. The other is the ***Scott topology***, which arises from the lattice structure of $\mathcal{P}(\mathbb{N})$, ordered by $\subseteq$. A subbasic open set for the Scott topology on $\mathcal{P}(\mathbb{N})$ is one of the form

$$\uparrow n = \{A \subseteq \mathbb{N} \mid n \in A\}.$$

By forming finite intersections we get the basic open sets

$$\uparrow\{n_0, \ldots, n_{k-1}\} = \{A \subseteq \mathbb{N} \mid \{n_0, \ldots, n_{k-1}\} \subseteq A\}.$$

Let us write

$$A \ll B \iff A \subseteq B \text{ and } A \text{ is finite.}$$

We may use $A \ll \mathbb{N}$ as a convenient shorthand for "A is a finite subset of $\mathbb{N}$". In the induced topology $\mathcal{U} \subseteq \mathcal{P}(\mathbb{N})$ is open if, and only if,

$$\mathcal{U} = \bigcup\{\uparrow A \mid A \in \mathcal{U} \land A \ll \mathbb{N}\}$$

or equivalently

$$B \in \mathcal{U} \iff \exists A \ll B.\, A \in \mathcal{U}.$$

It follows that a Scott open set $\mathcal{U}$ is upward closed: if $B \in \mathcal{U}$ and $B \subseteq C$ then $C \in \mathcal{U}$. Henceforth we let $\mathbb{P}$ denote $\mathcal{P}(\mathbb{N})$ by $\mathbb{P}$ qua topological space equipped with the Scott topology.

In information processing and computation the open sets are not about geometry but about (positively) observable properties. A basic observation about a set $B \in \mathcal{P}(\mathbb{N})$ is that it contains a number n, whence the Scott topology is generated by sets of the form $\uparrow n$. The Scott topology is not Hausdorff, not even a T_1-space, but is a T_0-space.[4] Indeed, if $A, B \in \mathcal{P}(\mathbb{N})$ have the same neighborhoods, then they have the same subbasic neighborhoods $\uparrow n$, but then they have the same elements.

4: The T_0 separation property is a form of Leibniz's principle of identity which states that two things are equal if they have exactly the same properties.

Another way to get the Scott topology of $\mathbb{P}$ is to observe that $\mathcal{P}(\mathbb{N})$ is in bijective correspondence with the set of all functions $\{\bot, \top\}^{\mathbb{N}}$. If we equip the two-element set $\mathbb{S} = \{\bot, \top\}$ with the ***Sierpinski topology*** in which the open sets are $\emptyset$, $\{\top\}$, and $\mathbb{S}$, then $\mathbb{P}$ turns out to be homeomorphic to $\mathbb{S}^{\mathbb{N}}$ equipped with the product topology.

Next we characterize the continuous maps on $\mathbb{P}$.

Proposition 2.2.1 *The following are equivalent for a map $f : \mathbb{P} \to \mathbb{P}$:*

1. *f is continuous,*
2. *$f(B) = \bigcup\{f(A) \mid A \ll B\}$ for all $B \in \mathbb{P}$,*
3. *f preserves directed unions.*

Proof. A map $f : \mathbb{P} \to \mathbb{P}$ is continuous precisely when the inverse image $f^*(\uparrow n)$ of every subbasic open set is open. By noting that $B \in f^*(\uparrow n)$ is equivalent to $n \in f(B)$ and using the characterization of Scott open sets, we may phrase continuity of f as

$$\forall n \in \mathbb{N}. \forall B \in \mathbb{P}. (n \in f(B) \iff \exists A \ll B. n \in f(A)),$$

which is equivalent requiring, for all $B \in \mathbb{P}$,

$$f(B) = \bigcup\{f(A) \mid A \ll B\}.$$

We have proved the equivalence of the first two statements. Since $\{A \mid A \ll B\}$ is a directed family, the third statement obviously implies the first one. The remaining implication is established as follows. Suppose $f : \mathbb{P} \to \mathbb{P}$ satisfies the second statement and $\mathcal{F} \subseteq \mathbb{P}$ is a directed family. Observe that the families $\mathcal{G} = \{A \in \mathbb{P} \mid \exists B. \mathcal{F} A \ll B\}$ and $\mathcal{H} = \{A \in \mathbb{P} \mid A \ll \bigcup \mathcal{F}\}$ are actually the same, both are directed, and $\bigcup \mathcal{F} = \bigcup \mathcal{H}$. Then

$$\begin{aligned} f(\bigcup \mathcal{F}) &= \bigcup\{f(A) \mid A \ll \bigcup \mathcal{F}\} \\ &= \bigcup\{f(A) \mid A \in \mathcal{H}\} \\ &= \bigcup\{f(A) \mid A \in \mathcal{G}\} \\ &= \bigcup\{\bigcup\{f(A) \mid A \ll B\} \mid B \in \mathcal{F}\} \\ &= \bigcup\{f(B) \mid B \in \mathcal{F}\}. \end{aligned}$$

We used the second statement in the first and last line. □

A continuous map on $\mathbb{P}$ is also called an ***enumeration operator***.[5] The second part of the last proposition says that every enumeration operator is determined by its values on finite sets. Thus, to encode an enumeration

5: The terminology (probably) originates from computability theory, where the Scott continuous maps on $\mathbb{P}$ correspond to higher-order functions operating on computably enumerable sets.

operator as a set of numbers, it suffices to encode its values on finite sets. We encode $A \ll \mathbb{N}$ as

$$\ulcorner A \urcorner = \sum_{n \in A} 2^n.$$

and assign to every continuous $f : \mathbb{P} \to \mathbb{P}$ its ***graph***[6]

$$\Gamma(f) = \{\ulcorner(\ulcorner A \urcorner, n)\urcorner \in \mathbb{N} \mid A \ll \mathbb{N} \land n \in f(A)\}.$$

6: The graph of $g : X \to Y$ is usually defined as $\{(x, y) \in X \times Y \mid f(x) = y\}$. Our definition records the *finitary* part of the graph of an enumeration operator.

Conversely, to every $A \in \mathbb{P}$ we assign a map $\Lambda(A) : \mathbb{P} \to \mathbb{P}$, defined by

$$\Lambda(A)(B) = \{n \in \mathbb{N} \mid \exists C \ll B.\, \ulcorner(\ulcorner C \urcorner, n)\urcorner \in A\},$$

which is easily seen to be continuous. Moreover, for any continuous $f : \mathbb{P} \to \mathbb{P}$, $B \in \mathbb{P}$, and $n \in \mathbb{N}$ we have

$$\begin{aligned}
\Lambda(\Gamma(f))(B) &= \{n \in \mathbb{N} \mid \exists C \ll B.\, \ulcorner(\ulcorner C \urcorner, n)\urcorner \in \Gamma(f)\} \\
&= \{n \in \mathbb{N} \mid \exists C \ll B.\, n \in f(C)\} \\
&= \bigcup \{f(C) \mid C \ll B\} \\
&= f(B),
\end{aligned}$$

where we appealed to continuity of f in the last step. Therefore, Γ and Λ form a section-retraction pair[7]

$$\mathscr{C}(\mathbb{P}, \mathbb{P}) \underset{\Lambda}{\overset{\Gamma}{\rightleftarrows}} \mathbb{P} \tag{2.3}$$

7: Given $s : X \to Y$ and $r : Y \to X$ such that $r \circ s = \mathrm{id}_X$, we say that s is a ***section*** of the ***retraction*** r. Together they form a section-retraction pair.

and are continuous when the set of continuous maps $\mathscr{C}(\mathbb{P}, \mathbb{P})$ is equipped with the compact-open topology. Concretely, the topology of $\mathscr{C}(\mathbb{P}, \mathbb{P})$ is generated by subbasic open sets

$$\{f \in \mathscr{C}(\mathbb{P}, \mathbb{P}) \mid n \in f(A)\}$$

where $A \ll \mathbb{N}$ and $n \in \mathbb{N}$. We shall use Γ and Λ in Section 2.4 to model the untyped λ-calculus in $\mathbb{P}$.

Exercise 2.2.1 There is a pairing function $\langle -, - \rangle : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ which interleaves sets A and B as odd and even numbers, respectively,

$$\langle A, B \rangle = \{2m \mid m \in A\} \cup \{2n + 1 \mid n \in B\}.$$

Verifty that $\langle -, - \rangle$ is a homeomorphism between $\mathbb{P} \times \mathbb{P}$ and $\mathbb{P}$.

Let us now discuss the role of $\mathbb{P}$ as a model of computation. An ***enumeration*** of a set $A \subseteq \mathbb{N}$ is a function $e : \mathbb{N} \to \mathbb{N}$ such that

$$A = \{n \in \mathbb{N} \mid \exists k.\, \mathbb{N} e(k) = n + 1\}.$$

In words, e enumerates the elements of A, incremented by 1. The increment is needed so that e may enumerate the empty set by outputting only zeroes. The enumeration e may enumerate an element of A many times. Clearly, every $A \subseteq \mathbb{N}$ has an enumeration.

A ***computably enumerable set (c.e. set)***[8] $A \subseteq \mathbb{N}$ is one that has a computable enumeration e. Define the ***k-th stage*** of $e : \mathbb{N} \to \mathbb{N}$ to be the set

8: Such sets are also called "recursively enumerable (r.e.)" because "computable functions" used to be called "(general) recursive functions".

of elements enumerated by the first k terms of e,

$$e|_k = \{n \in \mathbb{N} \mid \exists j < k.\, e(j) = n + 1\}.$$

The stages form an increasing chain of finite sets

$$e|_0 \subseteq e|_1 \subseteq e|_2 \subseteq \cdots$$

whose union is the set enumerated by e.

We say that an enumeration operator $f : \mathbb{P} \to \mathbb{P}$ is ***computable*** when its graph $\Gamma(f)$ is a c.e. set. In what sense can we "compute" with a computable enumeration operator? Suppose the graph of $f : \mathbb{P} \to \mathbb{P}$ is enumerated by e_f and $A \subseteq \mathbb{N}$ is enumerated by e_A. Then we may compute an enumeration e_B of $B = f(A)$ as

$$e_B(\ulcorner(n, i, j)\urcorner) = \begin{cases} n + 1 & \text{if } \ulcorner(\ulcorner e_A|_i \urcorner, n)\urcorner \in e_f|_j, \\ 0 & \text{otherwise.} \end{cases}$$

Indeed, suppose $e_B(\ulcorner(n, i, j)\urcorner) = n + 1$. Then $n \in f(e_A|_j)$, hence $n \in f(A)$ Conversely, if $n \in f(A)$, there exists $C \ll A$ such that $n \in f(C)$, and for large enough i and j we have $C \subseteq e_A|_i$ and $\ulcorner(\ulcorner e_A|_i\urcorner, n)\urcorner \in e_f|_j$, so that $e_B(\ulcorner(n, i, j)\urcorner) = n + 1$.

You might expect to see the utm and smn theorems for $\mathbb{P}$ at this point, but we take a different route to computing with $\mathbb{P}$, namely by using (2.3) to interpret the λ-calculus in $\mathbb{P}$, see Section 2.4. We conclude the section by observing that enumeration operators yield the standard notion of computability on numbers.

Exercise 2.2.2 Show that a partial map $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is computable if, and only if, there exists a computable enumeration operator $F : \mathbb{P} \to \mathbb{P}$ such that, for all $n \in \mathbb{N}$,

$$F(\{n\}) = \begin{cases} \{f(n)\} & \text{if } f(n)\downarrow, \\ \emptyset & \text{otherwise.} \end{cases}$$

2.3 Church's λ-calculus

We have so far considered a model of computation based on Turing machines, and two models of a topological nature, the Baire space $\mathbb{B}$ and the graph model $\mathbb{P}$. We now look at a purely syntactic model, the λ-calculus, in which computation is expressed as manipulation of symbolic expressions. It was proposed by Alonzo Church [8] as a notion of general computation before Turing invented his machines. Only later did it turn out that Church's and Turing's models can simulate each other.

[8]: Church (1932), "A set of postulates for the foundation of logic"

The λ-calculus is the abstract theory of functions, just like group theory is the abstract theory of symmetries. There are two basic operations that can be performed with functions. The first one is the ***application*** of a function to an argument: if f is a function and a is an argument, then $f\ a$ is the application of f to a. The second operation is ***abstraction***: if x is a

variable and t is an expression in which x may appear freely, then there is a function f defined by

$$f(x) = t.$$

We named the newly formed function f, but we could have specified it without naming it by writing "x is mapped to t" as

$$x \mapsto t,$$

In λ-calculus we write the above as a ***λ-abstraction***

$$\lambda x.\, t.$$

For example, $\lambda x.\, \lambda y.\, (x^2 + y^3)$ is the function which maps an argument a to the function $\lambda y.\, (a^2 + y^3)$. In the expression $\lambda x.\, t$ the variable x is *bound* in t.

There are two kinds of λ-calculus, the *typed* and the *untyped* one. In the untyped version there are no restrictions on how application is formed, so that an expression such as

$$\lambda x.\, x\, x$$

is valid, whatever it means. In the simply typed λ-calculus every expression has a *type*, and there are rules for forming valid expressions and types. For example, we can only form an application $f\, a$ when a has a type A and f has a type $A \to B$, which indicates a function taking arguments of type A and giving results of type B. We postpone discussion of the type λ-calculus to Subsection 2.7.2.

In the *untyped* version no restrictions are imposed on application and abstraction. More precisely, the calculus consists of:

- An infinite supply of variables $x, y, z, \ldots$,
- For any expressions e_1 and e_2 we may form their application $e_1\, e_2$. Application associates to the left so that $e_1\, e_2\, e_3 = (e_1\, e_2)\, e_3$.
- If e is an expression, then $\lambda x.\, e$ is its abstraction, where x is bound in e. Think of $\lambda x.\, e$ as the function which maps x to e. We abbreviate a nested abstraction $\lambda x_1. \cdots \lambda x_n.\, e$ as $\lambda x_1 x_2 \ldots x_n.\, e$.

The above can be expressed succinctly by the grammar rules:

$$\begin{aligned} \text{Variable } v &::= x \mid y \mid z \mid \cdots \\ \text{Expression } e &::= v \mid e_1\, e_2 \mid \lambda x.\, e \end{aligned}$$

There are no constants, numbers, or other constants – we are studying the *pure* λ-calculus.

Expressions which only differ in the naming of bound variables are equal, thus $\lambda x.\, y\, x = \lambda z.\, y\, z \neq \lambda y.\, y\, y$. Substitution replaces free variables with expressions. We write $e[e_1/x_1, \ldots, e_n/x_n]$ for a simultaneous substitution of expressions $e_1, \ldots, e_n$ for variables $x_1, \ldots, x_n$ in e, respectively. The usual rules for bound variables must be observed when we perform substitutions.[9]

The basic axiom of λ-calculus is ***β-reduction***:

$$(\lambda x.\, e_1)\, e_2 = e_1[e_2/x].$$

9: It is notoriously easy to commit errors when defining the details of substitution. The best way to understand all the intricacies is to write a program that performs substitutions.

It says that the application of a function $\lambda x.\, e_1$ to an argument e_2 is computed by replacing x with e_2 in the function body e_1. A second axiom, which is sometimes assumed is ***η-reduction***, which says that

$$\lambda x.\, e\, x = e,$$

provided x does not occur freely in e. We will *not* assume η-reduction unless explicitly stated otherwise.

A sub-expression of the form $(\lambda x.\, e_1)\, e_2$ is called a ***β-redex***. If e contains such a redex, we may replace it by $\subset e_1 e_2/x$ to obtain a new expression e'. We say that we performed a ***β-reduction*** and write

$$e \mapsto e'$$

A chain of reductions $e \mapsto e' \mapsto \cdots \mapsto e''$ is written $e \mapsto^* e''$.

In a given expression there may be several β-redeces. For example, we can reduce $((\lambda x.\, x)\, a)\, ((\lambda y.\, y)\, b)$ either as

$$((\lambda x.\, x)\, a)\, ((\lambda y.\, y)\, b) \mapsto a\, ((\lambda y.\, y)\, b)$$

or as

$$((\lambda x.\, x)\, a)\, ((\lambda y.\, y)\, b) \mapsto ((\lambda x.\, x)\, a)\, b.$$

A theorem of Church and Rosser's [9] states that λ-calculus is ***confluent***, which means that the order of β-reductions is not important in the sense that two different ways of reducing and expression may always be reconciled by further reductions. In the above example we get $a\, b$ in both cases after one more reduction.

[9]: Church et al. (1936), "Some Properties of Conversion"

There are expressions which we can keep reducing forever, for example the term $\omega\, \omega$ where $\omega = \lambda x.\, f\, (x\, x)$ has an infinite reduction sequence

$$\omega\, \omega \mapsto f(\omega\, \omega) \mapsto f(f(\omega\, \omega)) \mapsto f(f(f(\omega\, \omega))) \mapsto \cdots$$

An expression in which no β-reductions are possible is called a ***normal form***. Think of normal forms[10] as "finished" computations, and those which cannot be reduced to a normal form as "non-terminating" computations.

10: From a programming-language perspective, it is unusual to compute under a λ-abstraction, as we do to reach a normal form. A good alternative is to consider the *weak-head normal form*, which avoids doing so.

We outline programming in λ-calculus but do not provide the proofs. First, a pairing with projections may be defined as follows:

$$\begin{aligned} \texttt{pair} &= \lambda xyz.\, z\, x\, y, \\ \texttt{fst} &= \lambda p.\, p\, (\lambda xy.\, x), \\ \texttt{snd} &= \lambda p.\, p\, (\lambda xy.\, y). \end{aligned}$$

With these we have

$$\texttt{fst}\, (\texttt{pair}\, a\, b) = a \qquad \text{and} \qquad \texttt{snd}\, (\texttt{pair}\, a\, b) = b,$$

for instance

$$\texttt{fst}\, (\texttt{pair}\, a\, b) = (\lambda z.\, z\, a\, b)\, (\lambda xy.\, x) = (\lambda xy.\, x)\, a\, b = a.$$

The Boolean values and the conditional statement are encoded as

$$\begin{aligned}\mathtt{if} &= \lambda x.\, x,\\ \mathtt{true} &= \lambda x y.\, x,\\ \mathtt{false} &= \lambda x y.\, y.\end{aligned}$$

They satisfy

$$\mathtt{if}\,\mathtt{false}\,a\,b = b \qquad \text{and} \qquad \mathtt{if}\,\mathtt{true}\,a\,b = a.$$

The natural numbers are encoded by ***Church numerals***. The n-th Church numeral is a function which maps a function to its n-th iteration:

$$\begin{aligned}\overline{0} &= \lambda f\, x.\, x,\\ \overline{1} &= \lambda f\, x.\, f\, x,\\ \overline{2} &= \lambda f\, x.\, f\,(f\, x),\end{aligned}$$

and in general

$$\overline{n} = \lambda f\, x.\, \underbrace{f\,(\cdots (f}_{n}\; x)\cdots).$$

The successor, addition and multiplication operations are as follows:

$$\begin{aligned}\mathtt{succ} &= \lambda n\, f\, x.\, n f (f x),\\ \mathtt{add} &= \lambda m\, n\, f\, x.\, m f (n f x),\\ \mathtt{mult} &= \lambda m\, n\, f\, x.\, m (n f) x.\end{aligned}$$

We leave it as exercise to figure out how the following work and what they do:[11]

$$\begin{aligned}\mathtt{power} &= \lambda m\, n.\, n\, m,\\ \mathtt{iszero} &= \lambda n.\, n\,(\lambda x.\, \mathtt{false})\,\mathtt{true},\\ \mathtt{pred} &= \lambda n.\, \mathtt{snd}\,(n\,(\lambda p.\, \mathtt{pair}\,(\mathtt{succ}\,(\mathtt{fst}\,p))\,(\mathtt{fst}\,p))(\mathtt{pair}\,\overline{0}\,\overline{0})).\end{aligned}$$

11: A legend says that Alfred Tarski was at the dentist's when he figured out how to compute predecessors. Is programming the untyped λ-calculus like pulling one's teeth out?

Recursion is accomplished by means of the ***fixed-point operator***

$$\mathtt{fix} = \lambda f.\, (\lambda x.\, f\,(x\, x))(\lambda x.\, f\,(x\, x)).$$

For any a we have

$$\begin{aligned}\mathtt{fix}\, a &= (\lambda x.\, a\,(x\, x))(\lambda x.\, a\,(x\, x))\\ &= a\,((\lambda x.\, a\,(x\, x))\,(\lambda x.\, a\,(x\, x)))\\ &= a\,(\mathtt{fix}\, a).\end{aligned}$$

The fix-point operator is used to define recursive functions, for example equality of numbers is computed as follows:

$$\mathtt{equal} = \mathtt{fix}\,(\lambda e\, m\, n.\, \mathtt{if}\,(\mathtt{iszero}\, m)\,(\mathtt{iszero}\, n)\,(e\,(\mathtt{pred}\, m)\,(\mathtt{pred}\, n))).$$

By continuing in this manner we can build a general-purpose programming language. It turns out that the untyped λ-calculus computes exactly the same partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$ as Turing machines.

2.4 Reflexive domains

The Church-Rosser theorem implies that the untyped λ-calculus is consistent, i.e., not all expressions are equal. Indeed if $\lambda x.\, x$ and $\lambda x y.\, x$ were equal there would be a sequence of β-reductions (performed in either direction) leading from on to the other. By confluence we would obtain a normal form to which both reduce, but that cannot be since they already are distinct normal forms.

Still the question remains what the untyped λ-calculus is about, speaking mathematically as opposed to formalistically. A naive attempt at an interpretation runs into difficulties. Suppose we interpret the expressions of the λ-calculus as the elements of a set D, where λ-abstraction should correspond to formation of functions $D \to D$, and λ-application to application of such functions to elements of D. Because every λ-expression may be used either as an argument or a function, we require

$$D^D \underset{\Lambda}{\overset{\Gamma}{\rightleftarrows}} D \tag{2.4}$$

that mediate between the two roles. These can be used to interpret each λ-expression e with free variables $x_1, \ldots, x_n$ a map

$$[\![x_1, \ldots, x_n \mid e]\!] : D^n \to D$$

as follows, where $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{a} = (a_1, \ldots, a_n) \in D^n$:

$$\begin{aligned} [\![\vec{x} \mid x_i]\!]\, \vec{a} &= a_i \\ [\![\vec{x} \mid e_1\, e_2]\!]\, \vec{a} &= \Lambda([\![\vec{x} \mid e_1]\!]\, \vec{a})([\![\vec{x} \mid e_2]\!]\, \vec{a}), \\ [\![\vec{x} \mid \lambda y.\, e]\!]\, \vec{a} &= \Gamma(b \mapsto [\![\vec{x}, y \mid e]\!](\vec{a}, b)). \end{aligned} \tag{2.5}$$

Since we intend to interpret “functions as functions” and “application as application”, we expect, for all $f : D \to D$ and $a \in D$,

$$[\![x, y \mid x\, y]\!](\Gamma(f), a) = f\, a,$$

from which it follows that Γ is a section of Λ because

$$\Lambda(\Gamma\, f)\, a = [\![x, y \mid x\, y]\!](\Gamma(f), a) = f\, a.$$

The only set that contains its own function space as a retract is the singleton set. The λ-calculus has no non-trivial set-theoretic models. We need to look elsewhere, but where?

An answer was given by Dana Scott [33] who constructed a non-trivial topological space D_∞ such that the space of continuous functions $\mathscr{C}(D_\infty, D_\infty)$, equipped with the compact-open topology, is homeomorphic to D_∞. This gave a *topological* model of the untyped λ-calculus for $\beta\eta$-reduction. Since the construction involves more domain theory than we wish to assume here, we shall look at the simpler case of models that satisfy just β-reduction.

[33]: Scott (1972), “Continuous Lattices”

We seek a non-trivial topological space D that continuously retracts onto its own function space[12]

12: D must be nice enough for $\mathscr{C}(D, D)$, equipped with the compact-open topology, to be an *exponential* in the category of topological spaces and continuous maps.

$$\mathscr{C}(D, D) \underset{\Lambda}{\overset{\Gamma}{\rightleftarrows}} D \tag{2.6}$$

Such a space is called a ***reflexive domain***.

The denotations of λ-expressions in a reflexive domain D are given by (2.5), and β-reduction is valid thanks to Γ being a section of Λ:

$$\begin{aligned} [\![\vec{x} \mid (\lambda y.\, e_1)\, e_2]\!]\, \vec{a} &= \Lambda(\Gamma(b \mapsto [\![\vec{x}, y \mid e_1]\!]\, (\vec{a}, b)))\, ([\![\vec{x} \mid e_2]\!]\, \vec{a}) \\ &= (b \mapsto [\![\vec{x}, y \mid e_1]\!]\, (\vec{a}, b))\, ([\![\vec{x} \mid e_2]\!]\, \vec{a}) \\ &= [\![\vec{x}, y \mid e_1]\!]\, (\vec{a}, ([\![\vec{x} \mid e_2]\!]\, \vec{a})) \\ &= [\![\vec{x} \mid e_1[e_2/y]]\!]\, \vec{a}. \end{aligned}$$

Exercise 2.4.1 Prove the *substitution lemma*

$$[\![\vec{x}, y \mid e_1]\!]\, (\vec{a}, [\![\vec{x} \mid e_2]\!]\, \vec{a}) = [\![\vec{x}, y \mid e_1[e_2/y]]\!]\, \vec{a},$$

which we used in the last line above. You may proceed by induction on the structure of e_1, but should first generalize the statement so that the induction case of λ-abstraction works out.

We have already seen a diagram like (2.6), namely the section-retraction pair (2.3) for the graph model $\mathbb{P}$ from Section 2.2. We thus have at least one example of a reflexive domain. There are many other reflexive domains, such as Plotkin's T^ω [30] and the universal Scott domain [14].

[30]: Plotkin (1978), "$\mathbb{T}^\omega$ as a Universal Domain"
[14]: Gunter et al. (1990), "Semantic Domains"

To see how the λ-calculus helps prove things about reflexive domains, let us show that $D \times D$ is a retract of D with the aid of $\mathtt{pair} = \lambda x y z.\, z\, x\, y$ from the previous section. Let $p : D \times D \to D$ be defined as

$$p(a, b) = [\![x, y \mid \lambda z.\, z\, x\, y]\!](a, b),$$

and let $r : D \to D \times D$ be the map

$$r(c) = ([\![u \mid u\, (\lambda x y.\, x)]\!]\, c, [\![u \mid u\, (\lambda x y.\, y)]\!]\, c).$$

Then $r(p(a, b)) = (a, b)$ because λ-calculus proves

$$(\lambda x y.\, x)\, x\, y = x \qquad \text{and} \qquad (\lambda x y.\, y)\, x\, y = y.$$

The retraction Λ decodes an element of D as a continuous map $D \to D$. We may also decode $a \in D$ as a continuous map $\Lambda^{(2)}\, a : D \times D \to D$ of two arguments, namely $\Lambda^{(2)} a = \Gamma\, a \circ p$ where $p : D \times D \to D$ is the above section.

It is now easy to state and prove the utm and smn theorems for reflexive domains. Note however that they are somewhat redundant because the λ-calculus already does all the work.

Theorem 2.4.1 (reflexive domain smn) *Given a reflexive domain D, there is a continuous map $s : D \times D \to D$ such that $\Lambda(s(m, n))\, a = \Lambda^{(2)}\, m\, (n, a)$ for all $m, n, a \in D$.*

Proof. Take $s = [\![m, n \mid \lambda a.\, m\, (\lambda z.\, z\ n\ a)]\!]$. □

Theorem 2.4.2 (reflexive domain utm) *Given a reflexive domain D, there is a continuous map $u : D \times D \to D$ such that $u(t, m) = \Lambda\, t\, m$ for all $t, m \in D$.*

Proof. Take $u = [\![x, y \mid x y]\!]$. □

2.5 Partial combinatory algebras

The Baire space from Subsection 2.1.2 is almost a model of the untyped λ-calculus, because every $\alpha \in \mathbb{B}$ may be viewed both as a (realizer of) a function and an argument. One is then tempted to define application by $\alpha\,\beta = \boldsymbol{\eta}_\alpha(\beta)$, but this fails because the result $\boldsymbol{\eta}_\alpha(\beta)$ need not be defined, whereas application in λ-calculus is a total operation. We now consider a generalization of the λ-calculus which allows application to be a partial operation, and whose example is the Baire space.

Definition 2.5.1 A ***partial combinatory algebra (pca)*** $(\mathbb{A}, \cdot)$ is a set $\mathbb{A}$ with a *partial* binary operation $\cdot : \mathbb{A} \times \mathbb{A} \rightharpoonup \mathbb{A}$. We usually write $x\,y$ instead of $x \cdot y$, and recall that application associates to the left. Furthermore, there must exist $\mathtt{K}, \mathtt{S} \in \mathbb{A}$ such that, for all $x, y, z \in \mathbb{A}$,

$$\mathtt{K} \cdot x \cdot y = x, \qquad \mathtt{S} \cdot x \cdot y \cdot z \simeq (x \cdot z) \cdot (y \cdot z), \quad \text{and} \quad \mathtt{S} \cdot x \cdot y{\downarrow}. \tag{2.7}$$

A ***total combinatory algebra (CA)*** is a pca whose application is a total operation.

A ***sub-pca*** of $(\mathbb{A}, \cdot)$ is a subset $\mathbb{A}' \subseteq \mathbb{A}$ which is closed under application, and there exist $\mathtt{K}, \mathtt{S} \in \mathbb{A}'$ such that (2.7) is satisfied for all $x, y, z \in \mathbb{A}$.[13]

13: Read that carefully: the combinators $\mathtt{K}$ and $\mathtt{S}$ must come from the subaglebra $\mathbb{A}'$ but they must have the defining property with respect to $\mathbb{A}$, and so consequently also with respect to $\mathbb{A}'$.

The definition looks strange at first. Where did $\mathtt{K}$ and $\mathtt{S}$ come from? Theorem 2.5.1 below explains that the reason for this definition is a property called ***combinatory completeness***. For a pca $\mathbb{A}$ we define ***expressions over*** $\mathbb{A}$ inductively as follows:

- every $a \in \mathbb{A}$ is an expression over $\mathbb{A}$,
- a variable is an expression over $\mathbb{A}$,
- if e_1 and e_2 are expressions then $e_1 \cdot e_2$ is an expression over $\mathbb{A}$.

Application associates to the left, $x \cdot y \cdot z = (x \cdot y) \cdot z$. When no confusion can arise, we write $x\,y$ instead of $x \cdot y$.

An expression is ***closed*** if it contains no variables. We say that a closed expression e is ***defined*** and write $e{\downarrow}$ when all applications in e are defined so that e denotes an element of $\mathbb{A}$. More generally, if e is an expression with variables $x_1, \ldots, x_n$, we write $e{\downarrow}$ when, for all $a_1, \ldots, a_n \in \mathbb{A}$, the closed expression

$$e[a_1/x_1, \ldots, a_n/x_n]$$

is defined. If e and e' are expressions whose variables are among $x_1, \ldots, x_n$, we write $e \simeq e'$ when, for all $a_1, \ldots, a_n \in \mathbb{A}$,

$$e[a_1/x_1, \ldots, a_n/x_n] \simeq e'[a_1/x_1, \ldots, a_n/x_n].$$

Theorem 2.5.1 (Combinatory completeness) *Let* $(\mathbb{A}, \cdot)$ *be a pca. For every variable* x *and expression* e *over* $\mathbb{A}$ *there is an expression* e' *over* $\mathbb{A}$ *whose variables are those of* e *excluding* x *such that* $e'\downarrow$ *and* $e' \cdot a \simeq e[a/x]$ *for all* $a \in \mathbb{A}$.

Proof. We give a construction of such an expression $\langle x \rangle\, e$:

1. $\langle x \rangle\, x = \mathtt{S}\,\mathtt{K}\,\mathtt{K}$,
2. $\langle x \rangle\, y = \mathtt{K}\, y$ if y is a variable distinct from x,
3. $\langle x \rangle\, a = \mathtt{K}\, a$ if a is an element of $\mathbb{A}$,
4. $\langle x \rangle\, e_1\, e_2 = \mathtt{S}\,(\langle x \rangle\, e_1)\,(\langle x \rangle\, e_2)$

We omit the verification that $e' = \langle x \rangle\, e$ has the required properties. See [22] for details. □

[22]: Longley (1995), "Realizability Toposes and Language Semantics"

The meta-notation $\langle x \rangle\, e$ plays a role similar to that of λ-abstraction in the untyped λ-calculus. We abbreviate $\langle x \rangle\, \langle y \rangle\, e$ as $\langle x\, y \rangle\, e$, and similarly for more variables. We need to be careful with the "β-rule" because it is only valid in a restricted sense, see [22] for details.

We can build up the identity function, pairs, conditionals, natural numbers, and recursion by combining the two basic combinators K and S. The encoding of basic programming constructs is similar to the encoding by untyped λ-calculus. Pairing, projections, Boolean values and conditional statement are the same, except that λ-abstraction must be replaced by $\langle\,\rangle$-notation:

$$\begin{aligned} \mathtt{pair} &= \langle xyz \rangle\, z\, x\, y, & \mathtt{if} &= \langle x \rangle\, x, \\ \mathtt{fst} &= \langle p \rangle\, p\, (\langle x\, y \rangle\, x), & \mathtt{true} &= \langle x\, y \rangle\, x, \\ \mathtt{snd} &= \langle p \rangle\, p\, (\langle x\, y \rangle\, y) & \mathtt{false} &= \langle x\, y \rangle\, y. \end{aligned}$$

The notation $\langle\,\rangle$ makes expressions much more comprehensible and saves a lot of space, for example the term $\mathtt{pair} = \langle x\, y\, z \rangle\, z\, x\, y$ is quite unwieldy when written just with S and K:

$$\begin{aligned} \mathtt{pair} = {} & \mathtt{S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(S(KS)(S(KK)(KS)))} \\ & \mathtt{(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(KK)(KK))))(S(KK)(KK)))))} \\ & \mathtt{(S(S(KS)(S(KK)(KK)))(S(KK)(SKK))))))(S(S(KS)(S(KK)(KK)))} \\ & \mathtt{(S(S(KS)(KK))(KK)))}. \end{aligned}$$

Natural numbers are implemented as the ***Curry numerals***

$$\overline{0} = \mathtt{I} = \mathtt{S\,K\,K} \qquad \text{and} \qquad \overline{n+1} = \mathtt{pair}\,\mathtt{false}\,\overline{n}. \tag{2.8}$$

with successor, predecessor and zero-test

$$\begin{aligned} \mathtt{succ} &= \langle x \rangle\, \mathtt{pair}\,\mathtt{false}\, x, \\ \mathtt{iszero} &= \mathtt{fst}, \\ \mathtt{pred} &= \langle x \rangle\, \mathtt{if}\,(\mathtt{iszero}\, x)\,\overline{0}\,(\mathtt{snd}\, x). \end{aligned}$$

In a pca we can define functions by recursion by using the ***fixed point combinators*** $\mathtt{Y}$ ***and*** $\mathtt{Z}$, defined by

$$W = \langle xy\rangle\, y(x\,x\,y), \qquad \mathtt{Z} = W\,W,$$
$$X = \langle xyz\rangle\, y(x\,x\,y)z, \qquad \mathtt{Y} = X\,X.$$

These combinators satisfy, for all $f, x \in \mathbb{A}$,

$$\mathtt{Z}\,f \simeq f\,(\mathtt{Z}\,f), \qquad \mathtt{Y}f{\downarrow}, \qquad (\mathtt{Y}\,f)\,x \simeq f\,(\mathtt{Y}f)\,x.$$

The combinator $\mathtt{Z}$ can be used to implement primitive recursion on natural numbers as

$$R = \langle rxfm\rangle\, \mathtt{if}\,(\mathtt{iszero}\,m)\,(\mathtt{K}\,x)(\langle y\rangle\, f\,(\mathtt{pred}\,m)(r\,x\,f\,(\mathtt{pred}\,m)\,\mathtt{I}))$$
$$\mathtt{rec} = \langle xfm\rangle\,((\mathtt{Z}\,R)\,x\,f\,m\,\mathtt{I}).$$

It satisfies, for all $a, f \in \mathbb{A}$ and $n \in \mathbb{N}$,

$$\mathtt{rec}\,a\,f\,\overline{0} = a, \qquad \mathtt{rec}\,a\,f\,\overline{n+1} \simeq f\,\overline{n}\,(\mathtt{rec}\,a\,f\,\overline{n}).$$

Exercise 2.5.1 Implement the ***minimization combinator*** $\mathtt{min}$, which satisfies, for all $f \in \mathbb{A}$ and $n \in \mathbb{N}$,

$$\mathtt{min}\,f = \overline{n} \iff \exists k > 0.\, f\,\overline{n} = \overline{k} \text{ and } \forall m < n.\, f\,\overline{m} = \overline{0}.$$

With these combinators every general recursive function may be implemented in a pca.

2.5.1 Examples of partial combinatory algebras

The models of computation that we have considered so far are all examples of partial combinatory algebras.

The first Kleene Algebra $\mathbb{K}_1$

Turing machines, or more precisely their codes, form a pca $\mathbb{K}_1 = (\mathbb{N}, \cdot)$ whose application is defined as $m \cdot n = \boldsymbol{\varphi}_m(n)$. Because $m \cdot n$ can be easily confused with multiplication, we also write application with Kleene's notation $\{m\}(n)$.

The combinator $\mathtt{K}$ is easily obtained. The function $p(x, y) = x$ is easily seen to be computable. By the smn theorem there exists a computable map $q : \mathbb{N} \to \mathbb{N}$ such that $\boldsymbol{\varphi}_{q(x)}(y) = x$. Take $\mathtt{K}$ to be any number such that $q = \boldsymbol{\varphi}_{\mathtt{K}}$.

The combinator $\mathtt{S}$ requires a bit more thought. The partial function $g(x, y, z) = \boldsymbol{\varphi}_{\boldsymbol{\varphi}_x(z)}(\boldsymbol{\varphi}_y(z))$ is computable by several applications of the utm theorem. By the smn theorem there is a computable $r : \mathbb{N} \to \mathbb{N}$ such that $\boldsymbol{\varphi}_{r(x,y)}(z) = g(x, y, z)$. Another application of the smn theorem yields a computable function $q : \mathbb{N} \to \mathbb{N}$ such that $\boldsymbol{\varphi}_{q(x)}(y) = r(x, y)$. Take $\mathtt{S}$ to be any number such that $q = \boldsymbol{\varphi}_{\mathtt{S}}$.

The first Kleene Algebra $\mathbb{K}_1^A$ with an oracle

When Turing machines are replaced with oracle Turing machines, we obtain a variant of $\mathbb{K}_1$. More precisely, given any $A \subseteq \mathbb{N}$, we let $\mathbb{K}_1^A$ to be the pca whose underlying set is $\mathbb{N}$, equipped with the application $\{m\}^A(n) = \boldsymbol{\varphi}_m^A(n)$. Here $\boldsymbol{\varphi}_m^A$ is the m-th Turing machine with oracle A.

Hamkin's infinite-time pca $\mathbb{J}$

We may similarly use infinite-time Turing machines from Subsection 2.1.4 to define a pca $\mathbb{J}$ (for "Joel") whose underlying set is $\mathbb{N}$ and whose application is defined by $m \cdot n = \boldsymbol{\psi}_m(n)$, where $\boldsymbol{\psi}_m$ is the m-th infinite-time Turing machine.

The second Kleene Algebra $\mathbb{B}$

The Baire space $\mathbb{B}$, is a pca $(\mathbb{B}, |)$ whose application is $\alpha \mid \beta = \boldsymbol{\eta}_\alpha(\beta)$, as defined in (2.2). The combinators $\mathtt{K}$ and $\mathtt{S}$ exists by Theorems 2.1.11 and 2.1.12, analogously to the first Kleene algebra.

There are actually two version of the second Kleene algebra, the ***continuous*** one with carrier $\mathbb{B}$, and the ***computable*** one, whose carrier

$$\mathbb{B}_\# = \{\alpha \in \mathbb{B} \mid \alpha \text{ is computable}\}.$$

consists only of the computable sequences. Because $\mathtt{K}$ and $\mathtt{S}$ obtained above are computable, $\mathbb{B}_\#$ is a sub-pca of $\mathbb{B}$.

Combinatory logic $\mathbb{CL}$

The closed terms of cominbatory logic are generated from constants $\mathtt{K}$, $\mathtt{S}$ and a binary operation $\cdot$. We let the operation associate to the left and write it as juxtaposition. Let $\approx$ be the least congruence relation[14] on the set CL of all closed terms satisfying, for all $a, b, c \in \mathrm{CL}$,

$$\mathtt{K}\,a\,b \approx a \qquad\qquad \mathtt{S}\,a\,b\,c \approx (a\,c)\,(b\,c).$$

The quotient $\mathrm{CL}/{\approx}$ is the carrier of a total combinatory algebra $\mathbb{CL}$ whose structure is induced by the constants $\mathtt{K}$, $\mathtt{S}$ and the binary operation. It is called ***combinatory logic***.

14: A ***congruence relation*** is an equivalence relation that respects the operations. In the present case, if $a \approx a'$ and $b \approx b'$ then $a \cdot b \approx a' \cdot b'$.

The untyped λ-calculus Λ

The closed expressions of the untyped λ-calculus form a (total) combinatory algebra Λ whose application is the one from λ-calculus. More precisely, we quotient the set of closed expressions by the equivalence relation generated by β-reduction. The basic combinators are $\mathtt{K} = \lambda x y.\, x$ and $\mathtt{S} = \lambda x y z.\, (x\, z)(y\, z)$. In Subsection 2.7.2 we shall consider a typed version of the λ-calculus and variants that turn it into a programming language with an operational semantics.

Reflexive domains and the graph model

A reflexive domain D is a total combinatory algebra with application defined by $a \cdot b = \Lambda\, a\, b$, where $\Lambda : D \to \mathscr{C}(D, D)$ is the retraction onto the function space, see (2.6). The combinators `K` and `S` are (the interpretations of) those from the untyped λ-calculus.

The graph model comes in two versions. The ***continuous graph model*** has as its carrier the full powerset $\mathbb{P} = \mathscr{P}(\mathbb{N})$ and the ***computable graph model*** the computably enumerable sets

$$\mathbb{P}_{\#} = \{A \in \mathbb{P} \mid A \text{ is computably enumerable}\}.$$

The latter is a sub-pca of the former.

Other partial combinatory algebras

If we replace Turing machines with oracle Turing machines from Subsection 2.1.3, we obtain the ***oracle first Kleene algebra***, and by switching to infinite-time Turing machines from Subsection 2.1.4 the ***infinite-time first Kleene algebra***. In all cases the basic combinators `K` and `S` exist by the corresponding smn and utm theorems.

There are still many more partial combinatory algebras, but we shall have to stop here. We refer the interested readers to $|1|$.4 of [29].

[29]: Oosten (2008), *Realizability: An Introduction To Its Categorical Side*

2.6 Typed partial combinatory algebras

pcas and their models lack an important feature that most real-world programming languages have, namely *types* that impose well-formedness restrictions on programs. For instance, in a typical programming language we cannot meaningfully apply an expression to itself because the typing discipline prevents us from doing so.

In this section we look at pcas with types, which were defined by John Longley [25]. His definition sets up a type system that limits the applicability of the basic combinators, which is offset by additional basic combinators. In the end we obtain a notion that is closer to a programming language than to magical incantations with `K` and `S`.

[25]: Longley (1999), "Unifying typed and untyped realizability"

Definition 2.6.1 A ***type system*** $\mathcal{T}$ is a non-empty set, whose elements are called ***types***, equipped with two binary operations $\times$ and $\to$.

The operation $\to$ associates to the right, $s \to t \to u = s \to (t \to u)$.

Definition 2.6.2 A ***typed partial combinatory algebra (tpca)*** $\mathbb{A}$ over a type system $\mathcal{T}$ consists of

- a non-empty set $\mathbb{A}_t$ for each $t \in \mathcal{T}$, and
- a partial function $\cdot_{s,t} : \mathbb{A}_{s \to t} \times \mathbb{A}_s \to \mathbb{A}_t$, called ***application***, for all $s, t \in \mathcal{T}$,

such that for all $s, t, u \in \mathcal{T}$ there exist elements

$$\begin{aligned}
\mathtt{K}_{s,t} &\in \mathbb{A}_{s \to t \to s}, \\
\mathtt{S}_{s,t,u} &\in \mathbb{A}_{(s \to t \to u) \to (s \to t) \to s \to u}, \\
\mathtt{pair}_{s,t} &\in \mathbb{A}_{s \to t \to s \times t}, \\
\mathtt{fst}_{s,t} &\in \mathbb{A}_{s \times t \to s}, \\
\mathtt{snd}_{s,t} &\in \mathbb{A}_{s \times t \to t}.
\end{aligned}$$

We usually omit the types in subscripts and write $x\,y$ for $x \cdot_{s,t} y$. For all elements x, y, z of the appropriate types we require:

$$\begin{aligned}
\mathtt{K}\,x\,y &= x, \\
\mathtt{S}\,x\,y&{\downarrow}, \\
\mathtt{S}\,x\,y\,z &\succeq (x\,z)(y\,z), \\
\mathtt{fst}\,(\mathtt{pair}\,x\,y) &= x, \\
\mathtt{snd}\,(\mathtt{pair}\,x\,y) &= y.
\end{aligned}$$

We say that the elements $\mathtt{K}_{s,t}$, $\mathtt{S}_{s,t,u}$, $\mathtt{pair}_{s,t}$, $\mathtt{fst}_{s,t}$, $\mathtt{snd}_{s,t}$ are ***suitable*** for $\mathbb{A}$ when they satisfy the above properties.

A ***typed (total) combinatory algebra*** is a tpca whose application operations are total.

We have required the sets $\mathbb{A}_t$ to be non-empty. While this is not strictly necessary, it simplifies several constructions.

In a pca the natural numbers may be encoded with the basic combinators as the Curry numerals. In a tpca they are must be postulated separately.

Definition 2.6.3 A ***tpca with numerals (n-tpca)*** is a tpca $\mathbb{A}$ in which there is a type $\mathtt{nat}$ and elements

$$\begin{aligned}
\overline{0}, \overline{1}, \overline{2}, \ldots &\in \mathbb{A}_{\mathtt{nat}}, \\
\mathtt{succ} &\in \mathbb{A}_{\mathtt{nat} \to \mathtt{nat}}, \\
\mathtt{rec}_s &\in \mathbb{A}_{s \to (\mathtt{nat} \to s \to s) \to \mathtt{nat} \to s}.
\end{aligned}$$

such that for all x, f of appropriate types and all $n \in \mathbb{N}$

$$\begin{aligned}
\mathtt{succ}\,\overline{n} &= \overline{n+1}, \\
\mathtt{rec}\,x\,f\,\overline{0} &= x, \\
\mathtt{rec}\,x\,f\,\overline{n+1} &= f\,\overline{n}\,(\mathtt{rec}\,x\,f\,\overline{n}).
\end{aligned}$$

We say that $\mathtt{nat}$, $\overline{n}$, $\mathtt{succ}$, $\mathtt{rec}$, and the numerals satisfying these properties are ***suitable*** for $\mathbb{A}$.

Note that $\mathbb{A}_{\mathtt{nat}}$ may contain elements other than the numerals $\overline{n}$. An n-tpca has primitive recursion but may lack general recursion, so we need one more definition.

Definition 2.6.4 A ***tpca with numerals and general recursion (nr-tpca)***

is a n-tpca $\mathbb{A}$ containing, for all types s and t,

$$\mathtt{fix}_{s,t} \in \mathbb{A}_{((s\to t)\to(s\to t))\to(s\to t)}$$

such that, for all $f \in \mathbb{A}_{(s\to t)\to(s\to t)}$ and $x \in \mathbb{A}_s$,

$$\mathtt{fix}\, f\downarrow \qquad \text{and} \qquad \mathtt{fix}\, f\, x \succeq f\,(\mathtt{fix}\, f)\, x.$$

Such a $\mathtt{fix}_{s,t}$ is ***suitable*** for $\mathbb{A}$.

The relevant notion of substructure is as follows.

Definition 2.6.5 A ***sub-tpca*** $\mathbb{A}'$ of a tpca $\mathbb{A}$ is a collection of non-empty subsets $\mathbb{A}'_t \subseteq \mathbb{A}_t$, for each $t \in \mathscr{T}$, such that applications in $\mathbb{A}$ restrict to $\mathbb{A}'$, and there exist elements $\mathtt{K}_{s,t}$, $\mathtt{S}_{s,t,u}$, $\mathtt{pair}_{s,t}$, $\mathtt{fst}_{s,t}$, and $\mathtt{snd}_{s,t}$ in $\mathbb{A}'$ of appropriate types which are suitable for $\mathbb{A}$.

The notions of ***sub-n-tpca*** and ***sub-nr-tpca*** are defined analogously.

The basic theory of typed pcas follows the theory of untyped pcas. ***Expressions over*** $\mathbb{A}$ are defined inductively:

1. every $a \in \mathbb{A}_t$ is a an expression of type t, called a ***primitive constant***,
2. an annotated variable x^t is an expression of type t,
3. if e_1 is an expression of type $s \to t$ and e_2 is an expression of type s then $e_1 \cdot e_2$ is an expression of type t.

Note that variables are annotated with types. We also assume that the type of a primitive constant is unique (if not we tag constants with their types), so that every expression has at most one type.

A closed expression e of type t is defined, written $e\downarrow$, when all applications appearing in it are defined. Such an expression denotes an element of $\mathbb{A}_t$. If e contains variables $x_1^{t_1}, \ldots, x_n^{t_n}$, we write $e\downarrow$ when $e[a_1/x_1, \ldots, a_n/x_n]$ is defined for all $a_1 \in \mathbb{A}_{t_1}, \ldots, a_n \in \mathbb{A}_{t_n}$.

Theorem 2.6.1 (Combinatory completeness) *Let $\mathbb{A}$ be a tpca. For every expression e over $\mathbb{A}$ of type u and every variable x^t there is an expression e' of type $t \to u$ whose variables are those of e excluding x^t such that $e'\downarrow$ and $e' \cdot a \simeq e[a/x]$ for all $a \in \mathbb{A}_t$.*

Proof. Similarly to the untyped case we define $\langle x^t \rangle\, e$ rescursively as follows:

1. $\langle x^t \rangle\, x = \mathtt{S}_{t,t\to t,t}\, \mathtt{K}_{t,t\to t}\, \mathtt{K}_{t,t}$,
2. $\langle x^t \rangle\, y = \mathtt{K}_{s,t}\, y$ if y^s is a variable distinct from x^t,
3. $\langle x^t \rangle\, a = \mathtt{K}_{s,t}\, a$ if a is a primitive constant of type s.
4. $\langle x^t \rangle\, e_1\, e_2 = \mathtt{S}_{t,u,v}\, (\langle x^t \rangle\, e_1)\, (\langle x^t \rangle\, e_2)$ if e_1 and e_2 have types $u \to v$ and u, respectively.

The expression $e' = \langle x^t \rangle\, e$ satisfies the stated condition. □

2.7 Examples of Typed Partial Combinatory Algebras

2.7.1 Partial combinatory algebras

Every pca is an nr-tpca if we enrich it with the trivial types system that contains a single type, $\mathcal{T} = \{\star\}$ with (the only possible) operations $\star \times \star = \star \to \star = \star$, and $\mathbb{A}_\star = \mathbb{A}$. The required combinators are the ones we defined in Section 2.5, we just need to sprinkle $\star$ on `K` and `S` everywhere.

2.7.2 Simply typed λ-calculus

The simply typed λ-calculus is to the untyped λ-calculus as tpca is to a pca. The types are inductively generated from the ***unit type*** `unit`, a (possibly empty) collection of ***ground types***, by using the type constructors $\times$ and $\to$. Expressions are built inductively as follows, where s and t are types:

1. An annotated variable x^t is an expression of type t.
2. A set (possibly empty) of ***primitive constants*** with their associated types.
3. The constant $\star$, called the ***unit***, has type `unit`.
4. If e is an expression of type t then $\lambda x^s.\, e$ is an expression of type $s \to t$. The variable x is bound in e.
5. If e_1 and e_2 are expressions of types $s \to t$ and t, respectively, then $e_1\, e_2$ is an expressions of type t.
6. If e_1 and e_2 are expressions of types s and t, respectively, then (e_1, e_2) is an expression of type $s \times t$.
7. If e is an expression of type $s \times t$ then $\mathtt{fst}, e$ and $\mathtt{snd}, e$ are expressions of type s and t, respectively.

An alternative syntax omits annotations from variables, except in λ-abstractions where $\lambda x^t.\, e$ is then written as $\lambda x{:}t.\, e$. This is in fact the notation we normally prefer.

As in the untyped version, here too we have the rule of β-reduction

$$(\lambda x{:}t.\, e_1)e_2 = e_1[e_2/x].$$

There is also a corresponding η-reduction rule, for an expression e of type $s \to t$,

$$\lambda x{:}s.\, e\, x = e.$$

In the untyped λ-calculus we did not assume the η-rule, but now we do.

The unit type is characterized by the equation, for any expression e of type $\star$,

$$\star = e.$$

The rules for pairing and projections are, for all e_1, e_2, e of type s, t, $s \times t$:

$$\begin{aligned} \mathtt{fst}\,(e_1, e_2) &= e_1, \\ \mathtt{snd}\,(e_1, e_2) &= e_2, \\ (\mathtt{fst}\,e, \mathtt{snd}\,e) &= e, \end{aligned}$$

There may be additional equations involving primitive constants.

The ***pure simply typed λ-calculus*** has only a single ground type o and no primitive constants.

The simply typed λ-calculus is a tpca whose types are those of the calculus. For a type t we take $\mathbb{A}_t$ to be the set of closed expressions of type t, quotiented by the least equivalence relation generated by η- and β-reduction and the rules for the unit, projections, pairing, and primitive constants.

2.7.3 Gödel's T

The pure simply typed λ-calculus is not an n-tpca because it lacks the natural numbers. To make it into an n-tpca, we add a ground type `nat` and primitive constants

$$\begin{aligned} 0 &: \mathtt{nat} \\ \mathtt{succ} &: \mathtt{nat} \to \mathtt{nat} \\ \mathtt{rec}_t &: t \to (\mathtt{nat} \to t \to t) \to \mathtt{nat} \to t \quad \text{(for each type } t\text{)} \end{aligned}$$

We also add two further equations

$$\begin{aligned} \mathtt{rec}\,e_1\,e_2\,0 &= e_1 \\ \mathtt{rec}\,e_1\,e_2\,(\mathtt{succ}\,e_3) &= e_2\,e_3\,(\mathtt{rec}\,e_1\,e_2\,e_3). \end{aligned}$$

This gives us an n-tpca T with numerals defined by $\overline{0} = 0$ and $\overline{n+1} = \mathtt{succ}\,\overline{n}$. More precisely, the elements of $\mathbb{A}_t$ are the closed expressions of type t, quotiented by the equivalence relation generated by the equational rules of the simply typeda λ-calculus and primitive recursion.

This n-tpca goes by the name ***Gödel's** T* [12]. It was defined by Kurt Gödel to give a proof of consistency of Peano arithmetic. It is not as powerful as a general pca or type 1 machines because only the primitive recursive functions are expressible in this system.

[12]: Gödel (1958), "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes"

2.7.4 Plotkin's PCF

The programming language PCF ("programming computable functions") was introduced by Gordon Plotkin [31]. It has since served as the "laboratory mouse" of the theory of programming languages. The design of real-world functional languages, such as SML, Ocaml, and Haskell, was inspired by PCF. In fact, PCF is a fragment of Haskell.

[31]: Plotkin (1977), "LCF Considered as a Programming Language"

We present PCF as an extension of the simply typed λ-calculus. It has two ground types, `nat` for natural numbers and `bool` for Booleans. The

primitive constants of PCF with their types are:

$$\begin{aligned}
\overline{n} &: \texttt{nat} \quad \text{(for each } n \in \mathbb{N}\text{)},\\
\texttt{succ} &: \texttt{nat} \to \texttt{nat},\\
\texttt{pred} &: \texttt{nat} \to \texttt{nat},\\
\texttt{true} &: \texttt{bool},\\
\texttt{false} &: \texttt{bool},\\
\texttt{iszero} &: \texttt{nat} \to \texttt{bool},\\
\texttt{if}_t &: \texttt{bool} \to t \to t \to t \quad \text{(for each type } t\text{)},\\
\texttt{fix}_t &: (t \to t) \to t \quad \text{(for each type } t\text{)},
\end{aligned}$$

The specific equations are:

$$\begin{aligned}
\texttt{succ}\,\overline{n} &= \overline{n+1}\\
\texttt{pred}\,\overline{0} &= \overline{0}\\
\texttt{pred}\,\overline{n+1} &= \overline{n}\\
\texttt{iszero}\,\overline{0} &= \texttt{true}\\
\texttt{iszero}\,\overline{n+1} &= \texttt{false}\\
\texttt{if}_t\,\texttt{true}\,e_1\,e_2 &= e_1\\
\texttt{if}_t\,\texttt{false}\,e_1\,e_2 &= e_2\\
\texttt{fix}_t f &= e\,(\texttt{fix}_t\,e)
\end{aligned}$$

To see that PCF is a an nr-tpca, take PCF_t to be the programs of type t, quotiented by the equations of the simply typed λ-calculus and the above equations. The primitive recursion combinator $\texttt{rec}_t$ required by the definition of nr-tpca is implemented as

$$\texttt{rec}_t = \texttt{fix}\,(\lambda r\,x\,f\,n.\,\texttt{if}\,n = 0\,\texttt{then}\,x\,\texttt{else}\,f\,n\,(r\,x\,f\,(\texttt{pred}\,n))).$$

2.7.5 PCF$^\infty$

Type 2 machines and the enumeration operators from Subsection 2.1.2 and Section 2.2 compute with non-computable data, which is convenient for studying computability over topological spaces. To allow PCF computations with non-computable data we extend it oracles, as follows.

For every function $f : \mathbb{N} \to \mathbb{N}$ we add a constant $\texttt{func}_f$ of type $\texttt{nat} \to \texttt{nat}$ to PCF, together with equations

$$\texttt{func}_f\,\overline{n} = \overline{f(n)}.$$

We write f instead of $\texttt{func}_f$ when no confusion can arise.

The resulting nr-tpca is denoted as PCF^∞. The original PCF is a sub-nr-tpca of PCF^∞.

2.8 Simulations

If you open a book on computability theory, chances are that you will find a statement saying that "models of computation are equivalent". The claim refers to a collection of specific models of computation, such as variations of Turing machines, λ-calculi, and recursive functions. The book supports the claim by describing simulations between such models, with varying degrees of detail, after which it hurries on to core topics of computability theory. An opportunity is missed to ask about a general notion of simulation, and a study of its structural properties.

We seize the opportunity and study a notion of simulation between pcas. An excellent one is John Longley's *applicative morphism* [23]. His definition extends easily to account for pcas with sub-pcas. We dare rename applicative morphisms to *simulations*, and consider only the untyped simulations. The typed version of simulations can also be set up as well, see [24].

[23]: Longley (1994), "Realizability Toposes and Language Semantics"

[24]: Longley (1999), "Matching typed and untyped realizability"

Definition 2.8.1 A ***(pca) simulation***, originally called an ***applicative morphism*** [23], $\rho : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}$ between pcas $\mathbb{E}$ and $\mathbb{F}$ is a total relation $\rho \subseteq \mathbb{E} \times \mathbb{F}$ for which there exists a ***realizer*** $r \in \mathbb{F}$ such that, for all $u, v \in \mathbb{E}$ and $x, y \in \mathbb{F}$,

- if $\rho(u, x)$ then $r\,x{\downarrow}$ and
- if $\rho(u, x)$, $\rho(u, y)$ and $u\,v{\downarrow}$ then $r\,x\,y{\downarrow}$ and $\rho(u\,v, r\,x\,y)$.

We write $\rho[u] = \{x \in \mathbb{F} \mid \rho(u, x)\}$.

A ***(sub-pca) simulation*** $\rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F}, \mathbb{F}')$ between pcas with sub-pcas is a simulation $\rho : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}$ which has a realizer $r \in \mathbb{F}'$, and such that ρ restricted to $\mathbb{E}' \times \mathbb{F}'$ is a simulation $\mathbb{E}' \xrightarrow{\text{pca}} \mathbb{F}'$ realized by r.

A realizer $r \in \mathbb{F}$ of a simulation is of course precisely an implementation in $\mathbb{F}$ of the applicative structure of $\mathbb{E}$.

We defined a simulation to be a total relation rather than a function because an element of the domain may be simulated by many elements of the codomain, without any one being preferred or distinguished. The notation $\rho[u]$ suggests that ρ is construed as a multi-valued map rather than a relation.

One might expect that a simulation ought to be a map $f : \mathbb{E} \to \mathbb{F}$ such that $f(\mathsf{K}_\mathbb{E}) = \mathsf{K}_\mathbb{F}$, $f(\mathsf{S}_\mathbb{F}) = \mathsf{S}_\mathbb{F}$, and $f(x \cdot_\mathbb{E} y) \simeq f x \cdot_\mathbb{F} f y$. This is how an algebraist would define a morphism, but we are interested in the computational aspects of pcas, not the algebraic ones.

Simulations can be ***composed*** as relations. If $\rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F}, \mathbb{F}')$ and $\sigma : (\mathbb{F}, \mathbb{F}') \xrightarrow{\text{pca}} (\mathbb{G}, \mathbb{G}')$ then $\sigma \circ \rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{G}, \mathbb{G}')$ is defined, for $x \in \mathbb{E}$ and $z \in \mathbb{G}$, by

$$z \in (\sigma \circ \rho)[x] \iff \exists y \in \mathbb{F}.\, y \in \rho[x] \land z \in \sigma[y].$$

Exercise 2.8.1 Show that $\sigma \circ \rho$ is realized if ρ and σ are.

The identity simulation $\mathrm{id}_{(\mathbb{E},\mathbb{E}')} : (\mathbb{E},\mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{E},\mathbb{E}')$ is the identity relation on $\mathbb{E}$. It is realized by $\langle x\, y\rangle\, x\, y$.

Pcas with sub-pcas and simulations between them therefore form a category. We equip it with a ***preorder enrichment***[15] $\leq$ as follows. Given $\rho, \sigma : (\mathbb{E},\mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F},\mathbb{F}')$, define $\rho \leq \sigma$ to hold when there exists a ***translation*** $t \in \mathbb{F}'$ such that, for all $x \in \mathbb{E}$ and $y \in \rho[x]$, $t\, x{\downarrow}$ and $t\, y \in \sigma[x]$.

15: A category $\mathscr{C}$ is preorder enriched when hom-sets $\mathscr{C}(X, Y)$ are equipped with preorders (reflexive and transitive relations) under which composition is monotone.

We write $\rho \sim \sigma$ when $\rho \leq \sigma$ and $\sigma \leq \rho$.

Exercise 2.8.2 Given $\rho, \rho' : (\mathbb{E},\mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F},\mathbb{F}')$ and $\sigma, \sigma' : (\mathbb{F},\mathbb{F}') \xrightarrow{\text{pca}} (\mathbb{G},\mathbb{G}')$, show that if $\rho \leq \rho'$ and $\sigma \leq \sigma'$ then $\sigma \circ \rho \leq \sigma' \circ \rho'$.

The preorder enrichment induces the notions of equivalence and adjunction of simulations.

Definition 2.8.2 Consider simulations

$$\delta : (\mathbb{E},\mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F},\mathbb{F}'), \qquad \gamma : (\mathbb{F},\mathbb{F}') \xrightarrow{\text{pca}} (\mathbb{E},\mathbb{E}').$$

They form an ***equivalence*** when $\gamma \circ \delta \sim 1_{\mathbb{E}}$ and $\delta \circ \gamma \sim 1_{\mathbb{F}}$.

They form an ***adjunction***, written $\gamma \dashv \delta$, when $1_{\mathbb{F}} \leq \delta \circ \gamma$ and $\gamma \circ \delta \leq 1_{\mathbb{E}}$. We say that γ is ***left adjoint*** to δ, or that δ is ***right adjoint*** to γ.

Such an adjoint pair is an ***adjoint inclusion*** when $\gamma \circ \delta \sim \mathrm{id}_{\mathbb{E}}$, and a ***adjoint retraction*** when $\delta \circ \gamma \sim 1_{\mathbb{F}}$.

2.8.1 Properties of simulations

Nothing prevents a simulation from being trivial. In fact, there always is the constant simulation $\tau : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}$, defined by $\tau[x] = \{\mathtt{K}_{\mathbb{F}}\}$ and realized by $\langle x\, y\rangle\, \mathtt{K}$. To avoid such examples, we should identify further useful properties of simulations.

Discreteness prevents a simulation from conflating simulated elements.

Definition 2.8.3 A simulation $\rho : (\mathbb{E},\mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F},\mathbb{F}')$ is ***discrete*** when, for all $x, y \in \mathbb{E}$ if $\rho[x] \cap \rho[y]$ is in inhabited then $x = y$.

The next property is single-valuedness, up to equivalence.

Definition 2.8.4 A simulation $\rho : (\mathbb{E},\mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F},\mathbb{F}')$ is ***projective*** when there is a single-valued simulation (a function) ρ' such that $\rho' \sim \rho$.

Exercise 2.8.3 Prove that a simulation $\rho : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}$ is projective if, and only if, there is $t \in \mathbb{F}'$ such that, for all $x \in \mathbb{E}$ and $y, z \in \mathbb{F}$:

- if $y \in \rho[x]$ then $t\, y{\downarrow}$ and $t\, y \in \rho[x]$,
- if $y \in \rho[x]$ and $z \in \rho[x]$ then $t\, y = t\, z$.

Thus a simulation is projective if each element of $\mathbb{E}$ has a canonically chosen simulation in $\mathbb{F}$.

For every simulation $\rho : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}$ it is the case that the Boolean values of $\mathbb{F}$ can be converted to the simulated Boolean values. Indeed, take any $a \in \rho[\mathtt{true}_{\mathbb{E}}]$ and $b \in \rho[\mathtt{false}_{\mathbb{E}}]$ and define $e \in \mathbb{F}'$ to be $e = \langle x \rangle\, \mathtt{if}_{\mathbb{F}}\, x\, a\, b$, so that $e\, \mathtt{true}_{\mathbb{F}} \in \rho[\mathtt{true}_{\mathbb{E}}]$ and $e\, \mathtt{false}_{\mathbb{F}} \in \rho[\mathtt{false}_{\mathbb{E}}]$. The converse translation does not come for free.

Definition 2.8.5 A simulation $\rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F}, \mathbb{F}')$ is ***decidable*** when there is $d \in \mathbb{F}'$, called the ***decider*** for ρ, such that, for all $x \in \mathbb{F}$,

$$x \in \rho[\mathtt{true}_{\mathbb{E}}] \Rightarrow d\, x = \mathtt{true}_{\mathbb{F}},$$
$$x \in \rho[\mathtt{false}_{\mathbb{E}}] \Rightarrow d\, x = \mathtt{false}_{\mathbb{F}}.$$

Exercise 2.8.4 Say that a simulation $\rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F}, \mathbb{F}')$ ***preserves numerals*** when there is $c \in \mathbb{F}'$ such that, for all $n \in \mathbb{N}$ and $x \in \mathbb{F}$,

$$x \in \rho[\overline{n}_{\mathbb{E}}] \implies c\, x = \overline{n}_{\mathbb{F}}.$$

Prove that a simulation is decidable if, and only if, it preserves numerals.

We recall several basic results of John Longley's.

Theorem 2.8.1 *For $\delta : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}$ and $\gamma : \mathbb{F} \xrightarrow{\text{pca}} \mathbb{E}$:*

1. *If $\gamma \circ \delta \preceq \mathrm{id}_{\mathbb{E}}$ then δ is discrete and γ is decidable.*
2. *If $\gamma \dashv \delta$ then γ is projective.*

Proof. See [23, Theorem 2.5.3]. □

Corollary 2.8.2 *If $\gamma \dashv \delta$ is an adjoint retraction then both δ and γ are discrete and decidable, and γ is projective.*

Proof. Immediate. This is [23, Corollary 2.5.4]. □

Corollary 2.8.3 *If $\mathbb{E}$ and $\mathbb{F}$ are equivalent pcas, then the there exist an equivalence*

$$\delta : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}, \qquad \gamma : \mathbb{F} \xrightarrow{\text{pca}} \mathbb{E},$$

such that γ and δ are single-valued.

Proof. Both δ and γ are projective by Theorem 2.8.1. □

2.8.2 Decidable simulations and $\mathbb{K}_1$

Decidable simulations are the kind of simulations that arise in computability theory. We investigate them a bit, especially in relation to the first Kleene algebra $\mathbb{K}_1$.

Turing machines, embodied as Kleene's first algebra $\mathbb{K}_1$, are distinguished by a universal property.

Theorem 2.8.4 *Up to equivalence, the first Kleene algebra $\mathbb{K}_1$ is initial in the category of pcas and decidable simulations.*

Proof. We sketch the proof from [23, Theorem 2.4.18]. Given any pca $\mathbb{A}$, define $\kappa : \mathbb{K}_1 \xrightarrow{\text{pca}} \mathbb{A}$ by $\kappa[n] = \{\overline{n}_{\mathbb{A}}\}$. Because every partial computable function $\mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ can be represented in $\mathbb{A}$, there is $r \in \mathbb{A}$ such that, for all $k, m, n \in \mathbb{N}$,

$$r\,\overline{k}\,\overline{m} = \overline{n} \iff \varphi_k(m) = n.$$

Such an element r realizes κ. Furthermore, κ is decidable because it maps numbers to numerals.

Suppose $\mu : \mathbb{K}_1 \xrightarrow{\text{pca}} \mathbb{A}$ is another decidable simulation. Because μ preserves numerals there exists $f \in \mathbb{A}$ such that if $a \in \mu[n]$ then $f\,a = \overline{n} \in \kappa[n]$, therefore $\mu \leq \kappa$. The relation $\kappa \leq \mu$ holds by the next exercise, therefore $\kappa \sim \mu$. □

Exercise 2.8.5 Verify that for any $\rho : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F}$ there is $q \in \mathbb{F}$ such that $q\,\overline{n}_{\mathbb{F}} \in \rho[\overline{n}_{\mathbb{E}}]$ for all $n \in \mathbb{N}$.

Recall that a ***Turing reduction*** of $A \subseteq \mathbb{N}$ to $B \subseteq \mathbb{N}$, written $A \leq_T B$ is a B-oracle Turing machine M which computes the characteristic function of A.

Theorem 2.8.5 *Suppose $A, B \subseteq \mathbb{N}$. Then $A \leq_T B$ if, and only if, there is a decidable simulation $\mathbb{K}_1^A \xrightarrow{\text{pca}} \mathbb{K}_1^B$.*

Proof. This is [23, Proposition 3.1.6]. □

The following definition and exercise verify that having decidable simulations between $\mathbb{E}$ and $\mathbb{F}$ implies that they both compute the same number-theoretic functions, which is sometimes taken to be a notion of equivalence of computational models.

Definition 2.8.6 Say that a function $f : \mathbb{N} \to \mathbb{N}$ is ***realizable*** in a pca $\mathbb{A}$ when there exists $r \in \mathbb{A}$ such that $a\,\overline{n}{\downarrow}$ and $a\,\overline{n} = \overline{f(n)}$, for all $n \in \mathbb{N}$.

Pcas $\mathbb{E}$ and $\mathbb{F}$ are ***Turing-equivalent*** when they realize the same maps $\mathbb{N} \to \mathbb{N}$. A pca is ***Turing-complete*** when it is Turing-equivalent to $\mathbb{K}_1$.

Exercise 2.8.6 Suppose that

$$\delta : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{F} \qquad\qquad \gamma : \mathbb{F} \xrightarrow{\text{pca}} \mathbb{E}$$

are decidable simulations. Show that $\mathbb{E}$ and $\mathbb{F}$ are Turing-equivalent.

Consider again decidable simulations

$$\delta : \mathbb{E} \xrightarrow{\text{pca}} \mathbb{K}_1 \qquad\qquad \gamma : \mathbb{K}_1 \xrightarrow{\text{pca}} \mathbb{E}.$$

Because $\mathbb{K}_1$ is initial, γ is equivalent to $n \mapsto \overline{n}$, so we might as well assume that $\gamma[n] = \{\overline{n}\}$. Initiality also implies that $\delta \circ \gamma \sim \mathrm{id}_{\mathbb{K}_1}$.

Think of $n \in \delta[x]$ as the "source code" of $x \in \mathbb{E}$. A translation $t \in \mathbb{E}$ witnessing $\gamma \circ \delta \preceq \mathrm{id}_{\mathbb{E}}$ is a ***self-interpreter*** for $\mathbb{E}$. Indeed, given $x \in \mathbb{E}$ and $n \in \delta[x]$ we have $t\,\overline{n} = x$, which says that t evaluates the source code n to the value x represented by the source code. Therefore, an adjoint retraction $\gamma \dashv \delta$ from $\mathbb{E}$ onto $\mathbb{K}_1$ encompasses two features of $\mathbb{E}$, a self-interpreter and Turing-completeness.

An adjoint retraction from Λ to $\mathbb{K}_1$

We construct an adjoint retraction from the pca Λ of the closed terms of the untyped λ-calculus, and first Kleene algebra $\mathbb{K}_1$.

Define $\delta : \mathbb{K}_1 \xrightarrow{\text{pca}} \Lambda$ to be the simulation $\delta[n] = \{\overline{n}\}$ which encodes numbers as Curry numerals. It is a simulation because every partial computable map is λ-definable, and therefore so is Kleene application.

In the opposite direction, let $\gamma : \Lambda \xrightarrow{\text{pca}} \mathbb{K}_1$ be the total relation (remember that Λ is the set of closed terms quotiented by β-reduction),

$$\gamma[t] = \{\ulcorner t' \urcorner \mid t' \in \Lambda \wedge t =_\beta t'\}.$$

That is, an equivalence class of a closed terms is simulated by the codes of its members. The simulation is realized because there is a computable map $f : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ satisfying $f(\ulcorner t \urcorner, \ulcorner u \urcorner) = \ulcorner t\, u \urcorner$ for all $t, u \in \Lambda$.

Verifying $\delta \circ \gamma \preceq \mathrm{id}_\Lambda$ is a simple matter of programming a self-interpreter for the untyped λ-calculus. From a conceptual point of view it is clear that this can be done: the syntax of a term t can be discerned from the numeral $\overline{\ulcorner t \urcorner}$, so one just has to recursively traverse syntax tree of t and interpret it into λ-calculus.

Exercise 2.8.7 Why is it *not* the case that $\mathrm{id}_\Lambda \preceq \delta \circ \gamma$?

An adjoint retraction from $\mathbb{P}_\#$ to $\mathbb{K}_1$

Here is one more example, an adjoint retraction from the computable graph model $\mathbb{P}_\#$ and the first Kleene algebra [23, Proposition 3.3.7]. In one direction we define $\delta : \mathbb{K}_1 \xrightarrow{\text{pca}} \mathbb{P}_\#$ by

$$\delta[n] = \{\{n\}\}.$$

Careful with the nested singletons: the number n is simulated by the singleton $\{n\}$.

Exercise 2.8.8 Above we used singletons $\{n\}$ as numerals, but in (2.8) we defined the Curry numerals $\overline{n}$. Verify that in $\mathbb{P}_\#$ we can translate between these, i.e., that there are computable enumeration operators $f : \mathbb{P}_\# \to \mathbb{P}_\#$ and $g : \mathbb{P}_\# \to \mathbb{P}_\#$ such that $f(\{n\}) = \overline{n}_{\mathbb{P}_\#}$ and $g(\overline{n}_{\mathbb{P}_\#}) = \{n\}$, for all $n \in \mathbb{N}$.

In the other direction we define $\gamma : \mathbb{P}_\# \xrightarrow{\text{pca}} \mathbb{K}_1$ by taking

$$\gamma[A] = \{n \in \mathbb{N} \mid \mathrm{im}(\boldsymbol{\varphi}_n) = A\}$$

to be the index set[16] of A, i.e., the codes of partial computable maps whose image is A.

16: In computability theory, the ***index set*** of a set A is the set of all numbers (the indices) that encode elements of A.

Exercise 2.8.9 Verify that δ and γ are simulations.

To establish $\delta \circ \gamma \preceq \mathrm{id}_{\mathbb{P}_\#}$, observe that

$$S = \{\ulcorner\langle\ulcorner\{m\}\urcorner, n\rangle\urcorner \mid \exists k \in \mathbb{N}.\, \boldsymbol{\varphi}_m(k) = n\}$$

is computably enumerable. The computable enumeration operator $\Lambda(S) : \mathbb{P}_\# \to \mathbb{P}_\#$, where Λ is as in (2.3) is then a translation from $\delta \circ \gamma$ to $\mathrm{id}_{\mathbb{P}_\#}$.

Exercise 2.8.10 Why is it *not* the case that $\mathrm{id}_{\mathbb{P}_\#} \preceq \delta \circ \gamma$?

2.8.3 An adjoint retraction from $(\mathbb{P}, \mathbb{P}_\#)$ to $(\mathbb{B}, \mathbb{B}_\#)$

To give at least one example of simulations between pcas with sub-pcas, we review the adjoint retraction between the graph model and Kleene's second algebra, which was first given by Peter Lietz [21].

The map $\iota : \mathbb{B} \to \mathbb{P}$, defined by

$$\iota\,\alpha = \{\ulcorner a \urcorner \mid a \in \mathbb{N}^* \wedge a \sqsubseteq \alpha\}$$

represents a sequence α with the set (of codes) of its initial segments. It restricts to a map $\mathbb{B}_\# \to \mathbb{P}_\#$. Let us show that it is a simulation.

In Subsection 2.1.2 we defined the application $\alpha \cdot_{\mathbb{B}} \beta$ in $\mathbb{B}$ by a lookup procedure, by which every initial segment of $\alpha \cdot_{\mathbb{B}} \beta$ is determined by sufficiently long initial segments of α and β. Thus the relation $R \subseteq \mathbb{N}^* \times \mathbb{N}^* \times \mathbb{N}^*$ defined by

$$\begin{aligned}(a, b, c) \in R \iff& \\ &\forall \alpha, \beta, \gamma \in \mathbb{B}.\, (\alpha \cdot_{\mathbb{B}} \beta){\downarrow} \wedge a \sqsubseteq \alpha \wedge b \sqsubseteq \beta \Rightarrow c \sqsubseteq \alpha \cdot_{\mathbb{B}} \beta\end{aligned}$$

is computable. The enumeration operator $p : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$, defined by

$$p(A, B) = \{\ulcorner c \urcorner \mid \exists a, b \in \mathbb{N}^*.\, \ulcorner a \urcorner \in A \wedge \ulcorner b \urcorner \in B \wedge (a, b, c) \in R\},$$

is computable and is (the curried form of) a realizer for ι.

Let $\delta : \mathbb{P} \xrightarrow{\text{pca}} \mathbb{B}$ be the simulation defined by

$$\delta[A] = \{\alpha \in \mathbb{B} \mid A = \{n \in \mathbb{N} \mid \exists k \in \mathbb{N}.\, \alpha\, k = n + 1\}\}.$$

In words, α is a δ-simulation of A when it enumerates A, where the trick with adding 1 to n in the above definition makes it possible to enumerate the empty set. Clearly, if $\alpha \in \mathbb{B}_\#$ then $A \in \mathbb{P}_\#$. In order for δ to be a simulation, it suffices to find a partial computable map $f : \mathbb{B} \times \mathbb{B} \rightharpoonup \mathbb{B}$ such that, for all $A, B \in \mathbb{P}$,

$$\alpha \in \delta[A] \wedge \beta \in \delta[B] \Rightarrow f(\alpha, \beta) \in \delta[A \cdot_{\mathbb{P}} B].$$

To determine $f(\alpha, \beta)\ulcorner(m, n)\urcorner$, we look for $j < m$ such that $\alpha\, j = 1 + \ulcorner(\ulcorner C \urcorner, n)\urcorner$ and $C \ll A$. If there is one, we set $f(\alpha, \beta)\ulcorner(m, n)\urcorner = n + 1$, otherwise we set it to 0. Clearly, this is an effective procedure. If we compare the definition of f to the definition of application in $\mathbb{P}$, we see that they match.

Let us show that $\iota \dashv \delta$ is an adjoint retraction. Suppose $\alpha \in \mathbb{B}$ and $\beta \in \delta[\iota(\alpha)]$. We can computably reconstruct α from β, because β enumerates the initial segments of α. This shows that $\delta \circ \iota \preceq \mathrm{id}_{\mathbb{B}}$. Also, given α we can easily construct a sequence β which enumerates the initial segments of α, therefore $\mathrm{id}_{\mathbb{B}} \preceq \delta \circ \iota$, and we conclude that $\delta \circ \iota \sim \mathrm{id}_{\mathbb{B}}$.

To see that $\iota \circ \delta \preceq \mathrm{id}_{\mathbb{P}}$, consider $A, B \in \mathbb{P}$ and $\alpha \in \mathbb{B}$ such that $\alpha in \delta[A]$ and $B = \iota(\alpha)$. The sequence α enumerates A, and B consists of the initial segments of α. Hence, we can effectively reconstruct A from B, by

$$m \in A \iff \exists n \in B.\, n = 1 + \ulcorner a \urcorner \wedge \exists i < \|a\|.\, m = a_i.$$

Equivalence of Reflexive Domains

Consider reflexive domains

$$\mathscr{C}(D, D) \underset{\Lambda_D}{\overset{\Gamma_D}{\rightleftarrows}} D \qquad \mathscr{C}(E, E) \underset{\Lambda_E}{\overset{\Gamma_E}{\rightleftarrows}} D$$

which are also retracts of each other,

$$D \underset{r_D}{\overset{s_D}{\rightleftarrows}} E \qquad E \underset{r_E}{\overset{s_E}{\rightleftarrows}} D$$

so $r_D \circ s_D = \mathrm{id}_D$ and $r_E \circ s_E = \mathrm{id}_E$. Then D and E are equivalent as pcas.

Exercise 2.8.11 Verify the claim by constructing an equivalence $D \sim E$ from the given section-retraction pairs.

Realizability categories | 3

3.1 Motivation

Realizability was introduced by Stephen Kleene [17] who used it to build a model of intuitionistic arithmetic. We motivate it by asking a practical question: given a mathematical structure (a set equipped with operations and relations satisfying some axioms), what should its implementation look like?

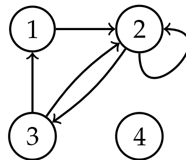
[17]: Kleene (1945), "On the Interpretation of Intuitionistic Number Theory"

For simple cases, the answer is obvious. A group is implemented by a type whose values represent its elements, a value representing the neutral element, and functions which compute the group operation and inverses. But for more interesting structures, especially those arising in mathematical analysis, the answer is less clear. How do we implement the real numbers? Which operations on a compact metric space can be implemented? How do we implement a space of smooth functions? Significant research goes into finding satisfactory answers to such questions [2, 4, 5, 36, 38].

[2]: Bauer (2000), "The Realizability Approach to Computable Analysis and Topology"
[4]: Bauer et al. (2010), "Canonical Effective Subalgebras of Classical Algebras as Constructive Metric Completions"
[5]: Blanck (1997), "Domain representability of metric spaces"
[36]: Tucker et al. (2000), "Computable Functions and Semicomputable Sets on Many-Sorted Algebras"
[38]: Weihrauch (2000), *Computable Analysis*

To explain the basic idea behind realizability we consider a small real-world programming example. Suppose we are asked to design a data structure for the set Graphs of all finite simple[1] directed graphs with vertices labeled by distinct integers, such at the graph G shown below:

1: A graph is ***simple*** when there is at most one edge between any two vertices.



A common representation of graphs uses a pair of lists (ℓ_V, ℓ_A), where ℓ_V is the list of vertex labels and ℓ_A the *adjacency list* representing the edges as pairs of labels. For the above graph these would be $\ell_V = [1, 2, 3, 4]$ and $\ell_A = [(1, 2), (2, 2), (2, 3), (3, 2), (3, 1)]$. Thus we define the datatype of graphs as[2]

2: We use Haskell notation in which $[t]$ is the type of lists of elements of type t, and (t_1, t_2) is the cartesian product of types t_1 and t_2.

```
type Graph = ([Int], [(Int, Int)])
```

This is not yet a complete description of the intended representation, as there are representation invariants and conditions not expressed by the type:

- the order in which the vertices and edges are listed is not important,
- every vertex and edge must be listed exactly once, and
- the source and target of each edge must appear in the list of vertices.

Such conditions can be expressed in terms of a ***realizability relation***

$$r \Vdash x$$

which tells which values r of the datatype correspond to which elements x of the set. We read $r \Vdash x$ as "r realizes (implements, represents, witnesses) x". In the above example we would write

$$([1,2,3,4],[(1,2),(2,2),(2,3),(3,2),(3,1)]) \Vdash G,$$

and also

$$([3,2,1,4],[(2,2),(1,2),(2,3),(3,2),(3,1)]) \Vdash G.$$

We also want to compute with the elements of Graphs. Programmers intuitively know what this mean, namely to implement, or realize, a map $f : \mathsf{Graphs} \to \mathsf{Graphs}$, is to give a program $p : \mathtt{graph} \to \mathtt{graph}$ which does to realizers what f does to elements: if $r \Vdash G$ then $p\,r \Vdash f(G)$. We say that f is ***realized*** or ***tracked*** by p.

3.2 Assemblies

We now give a precise definition of the ideas presented in the previous section.

Definition 3.2.1 An ***assembly*** over a tpca with a sub-tpca $(\mathbb{A}, \mathbb{A}')$ is a triple $S = (|S|, \|S\|, \Vdash_S)$ where $|S|$ is its ***underlying set***, $\|S\|$ its ***underlying type*** from $\mathbb{A}$, and $\Vdash_S$ is a relation between $\mathbb{A}_{|S|}$ and $|S|$ satisfying: for every $x \in |S|$ there is $\mathtt{x} \in \mathbb{A}_{|S|}$ such that $\mathtt{x} \Vdash_S x$.

An ***assembly map*** $f : S \to T$ between assemblies S and T is a map $f : |S| \to |T|$ between the underlying sets for which there exists $\mathtt{f} \in \mathbb{A}'_{\|S\| \to \|T\|}$, called a ***realizer*** of f, satisfying for all $\mathtt{x}, x$: if $\mathtt{x} \Vdash_S x$ then $\mathtt{f}\,\mathtt{x}{\downarrow}$ and $\mathtt{f}\,\mathtt{x} \Vdash_T f(x)$.

We sometimes require that the underlying tpca with sub-tpca $(\mathbb{A}, \mathbb{A}')$ is in fact an n-tpca, or an nr-tpca with a chosen substructure. Even though we may not be explicit about the requirement, it should be apparent from our using the type `nat` and the fixed-point combinators `Y`.

We often use the same letter for an element and its realizer, but differentiate between them by using different fonts, for instance the elements x, y, f, g would have realizers $\mathtt{x}$, $\mathtt{y}$, $\mathtt{f}$, $\mathtt{g}$, respectively.

There are many versions of realizability. Ours is known as ***typed relative realizability***. It is *typed* because we used typed pcas. It is *relative* because maps are realized relative to a choice of a sub-pca. In typical cases, such as type 2 machines and the graph model from Subsection 2.1.2 and Section 2.2, $\mathbb{A}'$ is the computable part of a topological pca $\mathbb{A}$, in accordance with the slogan

"Topological data – computable functions!"

When $\mathbb{A}' = \mathbb{A}$ we write $\mathsf{Asm}(\mathbb{A})$ instead of $\mathsf{Asm}(\mathbb{A}, \mathbb{A})$.

When $\mathbb{A}$ is untyped the definition of an assembly simplifies a bit because we need mention the (trivial) types.

Definition 3.2.2 An ***assembly*** over an untyped pca $\mathbb{A}$ is a pair $S = (|S|, \Vdash_S)$ where $|S|$ is a set and $\Vdash_S$ is a relation between $\mathbb{A}$ and $|S|$, such that for every $x \in |S|$ there is $r \in A$ and $r \Vdash_S x$.

Assemblies and maps over $(\mathbb{A}, \mathbb{A}')$ form a ***category*** $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$. Indeed, if $f : S \to T$ and $g : T \to U$ are realized by $\mathtt{f} \in \mathbb{A}'_{\|S\| \to \|T\|}$ and $\mathtt{g} \in \mathbb{A}'_{\|T\| \to \|U\|}$, respectively, then their composition $g \circ f$ is realized by $\langle x^{\|S\|} \rangle\, r\,(q\,x) = \mathtt{S}\,(\mathtt{K}\,r)\,(\mathtt{S}\,(\mathtt{K}\,q)\,(\mathtt{S}\,\mathtt{K}\,\mathtt{K}))$. The identity map $\mathrm{id}_S : |S| \to |S|$ is realized by $\langle x^{\|S\|} \rangle\, x = \mathtt{S}\,\mathtt{K}\,\mathtt{K}$. Composition is associative because it is just composition of maps.

3.2.1 Modest sets

In the definition of assemblies, nothing prevents several elements from sharing a common realizer. We sometimes want to prohibit such anomalies.

Definition 3.2.3 An ***modest*** assembly S, also called a ***modest set***,[3] is an assembly in which elements do not share realizers:

$$\forall r.\, \mathbb{A}_{|S|} \forall x, y \in |S|.\, (r \Vdash_S x \wedge r \Vdash_S y \Rightarrow x = y).$$

We let $\mathsf{Mod}(\mathbb{A}, \mathbb{A}')$ be the full subcategory of $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ on the modest sets.

3: The terminology was suggested by Dana Scott. It refers to the fact that the cardinality of a modest set S does not exceed the cardinality of $\mathbb{A}_{|S|}$.

Most structures in computable mathematics turn out to be modest, but assemblies are needed also, and they form a richer category than the modest sets.

3.2.2 The unit assembly $\mathbb{1}$

To gain a bit of intuition about assemblies, we look at several concrete examples of assemblies.

Let $\mathtt{unit}$ be a type with an element $\star \in \mathbb{A}'_{\mathtt{unit}}$. It always exists, because there is at least one type s, and then $\mathbb{A}_{s \to s \to s}$ contains $\mathtt{K}_{s,s}$.

The ***terminal assembly*** $\mathbb{1} = (\{\star\}, \mathtt{unit}, \Vdash_{\mathbb{1}})$ has the trivial realizability relation, $r \Vdash_{\mathbb{1}} \star$ for all $r \in \mathbb{A}'_t$.

Exercise 3.2.1 Show that $\mathbb{1}$ is the terminal object[4] in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$. Conclude from this that a different choice of $\mathtt{unit}$ results in an isomorphic copy of $\mathbb{1}$.

4: An object T in a category is ***terminal*** when there is precisely one morphism to T from every object.

The morphisms $\mathbb{1} \to S$ corresponds to those elements of $|S|$ that are realized by elements of $\mathbb{A}'_{\|S\|}$. Indeed, if $f : \star \to S$ is realized by $\mathtt{f} \in \mathbb{A}'_{\mathtt{unit} \to \|S\|}$ then $f\,\star$ is realized by $\mathtt{f}\,t \in \mathbb{A}'_{\|S\|}$, for any $t \in \mathbb{A}'_{\mathtt{unit}}$. Conversely, if $a \in |S|$ is realized by $\mathtt{a} \in \mathbb{A}'_{\|S\|}$ then $\star \mapsto a$ is $\langle x^{\mathtt{unit}} \rangle\, \mathtt{a}$.

This may be a good moment to point out the difference between the ***global points*** of S, which is the set of morphisms $\mathbb{1} \to S$, and the ***underlying***

set $|S|$ of S. Both induce functors $\mathsf{Asm}(\mathbb{A}, \mathbb{A}') \longrightarrow \mathsf{Set}$, which need not be equivalent, unless $\mathbb{A} = \mathbb{A}'$.

Exercise 3.2.2 The ***empty assembly*** $\mathbb{0}$ has as its underlying set the empty set $\emptyset$ and as its underlying type `unit`. Show that the choice of the underlying type does not matter and that the empty assembly is the initial object.

3.2.3 Natural numbers

Suppose $(\mathbb{A}, \mathbb{A}')$ is an n-tpca with a sub-n-tpca. Let $N = (\mathbb{N}, \mathtt{nat}, \Vdash_N)$ be the set of natural numbers $\mathbb{N}$ realized by the numerals, for all $r \in \mathbb{A}_{\mathtt{nat}}$ and $n \in \mathbb{N}$,

$$r \Vdash_N n \iff r = \overline{n}.$$

The successor $n \mapsto n + 1$ is realized by `succ`, and $0 \in \mathbb{N}$ is a global point because $\overline{0} \in \mathbb{A}'_{\mathtt{nat}}$.

The assembly $\mathbb{N}$ is the natural numbers object. Indeed, given an assembly S with $z \in |S|$ realized by $\mathtt{z} \in \mathbb{A}'_{\|S\|}$, and $f : |S| \longrightarrow |S|$, the unique map $\overline{f} : \mathbb{N} \longrightarrow |S|$ satisfying, for all $n \in \mathbb{N}$,

$$\overline{f}\,0 = 0 \qquad \text{and} \qquad \overline{f}\,(n+1) = f(\overline{f}\,n)$$

is realized by `rec z f`.

Exercise 3.2.3 Suppose $(\mathbb{A}, \mathbb{A}')$ is an tpca with a sub-tpca such that $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ has a natural numbers object. Show that $(\mathbb{A}, \mathbb{A}')$ is an n-tpca with a sub-n-tpca.

3.2.4 The constant assemblies

The extreme case of elements sharing the same realizer happens when all elements of a set share all realizers. Assemblies with this property are called the ***constant assemblies***.

Let t be a type such that $\mathbb{A}'_t$ is inhabited. Such a type always exists, because there is at least one type s, and then $\mathbb{A}_{s \to s \to s}$ contains $\mathtt{K}_{s,s}$. Given any set X, let

$$\nabla X = (X, t, \Vdash_{\nabla X})$$

be the assembly whose underlying set is X and the realizability relation is trivial, i.e., $r \Vdash_{\nabla X} x$ for all $x \in X$ and $r \in A_t$.

If $f : X \longrightarrow Y$ is any map between sets X and Y then f is a morphism $\nabla f : \nabla X \longrightarrow \nabla Y$ because it is tracked by $\langle x^t \rangle\, x$. Thus ∇ is a functor

$$\nabla : \mathsf{Set} \longrightarrow \mathsf{Asm}(\mathbb{A}, \mathbb{A}').$$

Up to natural isomorphism, ∇ is independent of the choice of type t. We will study the properties of ∇ later on. For now we notice that ∇ is full and faithful, which means that $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ contains the category of sets as a full subcategory.

The functor ∇ is devoid of any computational content because it represents a set X by a trivial realizability relation which conveys no information at all about the elements of X. Consequently, from the realizers we cannot compute anything interesting regarding X.

Exercise 3.2.4 Show that an assembly S is modest if, and only if, every assembly map $\nabla 2 \to S$ is constant.

Exercise 3.2.5 Given a set X and an assembly S, show that every map $X \to |S|$ is an assembly map $\nabla X \to S$.

The functor ∇ is part of an adjunction, as follows.

Exercise 3.2.6 Let $\Gamma : \mathsf{Asm}(\mathbb{A}, \mathbb{A}') \to \mathsf{Set}$ be the forgetful functor which assigns to an assembly its underlying set, and to an assembly map the underlying set-theoretic function. Show that Γ is left adjoint to ∇.

3.2.5 Two-element assemblies

We explore a bit what the two-element assemblies are like. For simplicity we consider the non-relative case $\mathsf{Asm}(\mathbb{A})$ of assemblies on a pca $\mathbb{A}$.

Without loss of generality we may assume that a two-element assembly T has $|T| = 2 = \{0, 1\}$ as its underlying set. Such an assembly is determined by the sets of realizers $\mathsf{E}_T 0 \subseteq \mathbb{A}$ and $\mathsf{E}_T 1 \subseteq \mathbb{A}$, which must both be inhabited.

We may partially order all two-element assemblies by stipulating that $T \leq U$, where $|T| = |U| = 2$ when id_2 is realized as an assembly map $T \to U$. That is, $T \leq U$ holds when the realizers of T are more informative than the realizers of U.

With respect to this ordering the largest two-element assembly is $\nabla 2$, since every map into a constant assembly is realized. We call $\nabla 2$ the ***classical truth values*** because it comes from classical set theory, where 2 is the object of truth values.

The least two-element assembly is $\mathbb{2} = (2, \Vdash_{\mathbb{2}})$ where, for $\mathtt{r} \in \mathbb{A}$ and $b \in 2$,

$$\mathtt{r} \Vdash_{\mathbb{2}} b \iff (\mathtt{r} = \mathtt{false} \land b = 0) \lor (\mathtt{r} = \mathtt{true} \land b = 1).$$

Indeed $\mathbb{2} \leq U$ is realized by $\langle \mathtt{r} \rangle \, \mathtt{if}\, \mathtt{r}\, \mathtt{a}\, \mathtt{b}$ where $\mathtt{a} \Vdash_U 0$ and $[b] \Vdash_U 1$. The assembly $\mathbb{2}$ is the assembly of ***Booleans*** or ***decidable truth values***.

There are plenty of assemblies between $\mathbb{2}$ and $\nabla 2$, for example the assembly Σ^0_1 of the ***semidecidable***[5] ***truth values***, also known as the ***Rosolini dominance***, is defined as

$$\begin{aligned} \mathtt{r} \Vdash_{\Sigma^0_1} b \iff & (\forall n .\, \mathtt{r}\, \overline{n} \in \{\mathtt{false}, \mathtt{true}\}) \land \\ & (b = 1 \Leftrightarrow \exists n .\, \mathtt{r}\, \overline{n} = \mathtt{true}). \end{aligned}$$

Its realizers compute infinite sequence of bits. Such a realizer represents 1 if, and only if, it computes a sequence that contains $\mathtt{true}$.

5: We need to be careful about the meaning of “semidecidable” because it depends on $\mathbb{A}$. For example, in assemblies over $\mathbb{K}_1$ the Rosolini dominance really does embody semidecidability, whereas in assemblies over $\mathbb{B}$ it is an admissible representation of the Sierpinski space, see Definition 3.5.3.

Exercise 3.2.7 Define two-element assemblies that correspond to the truth values in the *arithmetical hierarchy*, defined inductively as follows:

- $\Sigma^0_0 = \Pi^0_0 = \{\bot, \top\}$ are the *decidable truth values*,
- Σ^0_{n+1} are the truth values of the form $\exists n.\, pn$ where $p : \mathbb{N} \to \Pi^0_n$,
- Π^0_{n+1} are the truth values of the form $\forall n.\, pn$ where $p : \mathbb{N} \to \Sigma^0_n$.

In your definition you should replace the maps p with suitable realizers $\mathtt{r}$. Above we already constructed $\Sigma^1_0 = \Pi^0_0 = \mathbb{2}$ and Σ^0_1. Show that $\Sigma^0_n \leq \Pi^0_{n+1}$ and $\Pi^0_n \leq \Sigma^0_{n+1}$.

Real numbers

As our third example we ask how to equip the real numbers with a realizability structure. Here we give the correct answer, but leave it unexplained for the time being.

We work with an nr-tpca $\mathbb{A}$ with a sub-n-tpca $\mathbb{A}'$. Intuitively speaking, a realizer for $x \in \mathbb{R}$ should allow us to compute arbitrarily good approximations of x, so we define the relation $\Vdash_R$ between $\mathbb{A}_{\mathtt{nat}\to\mathtt{nat}\times\mathtt{nat}\times\mathtt{nat}}$ by stipulating that $\mathtt{x} \Vdash_R x$ holds if, and only if,

$$\forall k \in \mathbb{N}.\, \exists a, b, c \in \mathbb{N}.\, \mathtt{x}\,\overline{k} = (\overline{a}, \overline{b}, \overline{c}) \wedge |x - \frac{a-b}{1+b}| < 2^{-k}.$$

The triple of numbers (a, b, c) is just a clumsy way of encoding the rational $\frac{a-b}{c}$, so in essence $\mathtt{x}$ computes a sequence of rationals such that the k-th term is within 2^{-k} of x.

The assembly of real numbers $R = (\mathbb{R}_{\mathrm{rz}}, \mathtt{nat} \to \mathtt{nat} \times \mathtt{nat} \times \mathtt{nat}, \Vdash_R)$ has as its underlying set the realized reals

$$\mathbb{R}^{\mathrm{rz}} = \{x \in \mathbb{R} \mid \exists \mathtt{x} \in \mathbb{A}_{\mathtt{nat}\to\mathtt{nat}\times\mathtt{nat}\times\mathtt{nat}}.\, \mathtt{x} \Vdash_R x\}.$$

Which reals are so realized depends on the choice of $\mathbb{A}$. For example, first Kleene algebra realizes the ***Turing computable reals***, whereas the second Kleene algebra realizes all reals.

3.3 Equivalent formulations

Assemblies and modest sets have several equivalent formulations, which were formulated by different communities for particular choices of $(\mathbb{A}, \mathbb{A}')$, each using their own notation and terminology. In this section we review the equivalent formulations, and in Section 3.5 show how various "schools of computable mathematics" arise as special instances.

3.3.1 Existence predicates

A realizability relation $\Vdash_S$ is a subset of $\mathbb{A}_{|S|} \times |S|$. By transposition it may be equivalently expressed as a map $\mathsf{E}_S : |S| \to \mathscr{P}(\mathbb{A}_{|S|})$. The correspondence is

$$\mathtt{x} \Vdash_S x \iff \mathtt{x} \in \mathsf{E}_S(x).$$

Because every x is realized by something, $\mathsf{E}_S(x)$ always contains at least one element. Thus an assembly $(|S|, \|S\|, \Vdash_S)$ may be equivalently presented as a triple $(|S|, \|S\|, \mathsf{E}_S)$ where $\mathsf{E}_S : S \to \mathcal{P}(\mathbb{A}_{|S|})$ is a map, called the ***existence predicate***, such that $\mathsf{E}_S(x)$ contains at least one element for every $x \in |S|$. The name suggests that the elements of $\mathsf{E}_S(x)$ are computational witnesses for "existence of x".

An assembly S is modest if, and only if, $\mathsf{E}_S(x) \cap \mathsf{E}_S(y) \neq \emptyset$ implies $x = y$.

Under this formulation a map $f : S \to T$ is realized if there exists $\mathtt{f} \in \mathbb{A}'_{\|S\| \to \|T\|}$ such that, for all $x \in |S|$ and $\mathtt{x} \in \mathsf{E}_S(x)$, $\mathtt{f}\,\mathtt{x}\downarrow$ and $\mathtt{f}\,\mathtt{x} \in \mathsf{E}_T(f(x))$.

3.3.2 Representations

By transposing $\Vdash_S$ the other way around we obtain ***representations***. Suppose first that S is a modest set. Since every realizer $r \in \mathbb{A}_{|S|}$ realizes at most one $x \in |S|$, we may define a partial map $\delta_S : \mathbb{A}_{|S|} \rightharpoonup |S|$ by

$$\delta_S(r) = x \iff r \Vdash_S x.$$

The map δ_S is surjective because element is realized, but it need not be defined everywhere. The triple $(|S|, \|S\|, \delta_S)$ uniquely describes the modest set S. The map δ_S is called a ***representation*** of S.

A map $f : S \to T$ is realized or tracked by $\mathtt{f} \in \mathbb{A}'_{\|S\| \to \|T\|}$ when, for all $\mathtt{x} \in \mathsf{dom}(\delta_S)$, $\mathtt{f}\,\mathtt{x}\downarrow$ and $\delta_T(\mathtt{f}\,\mathtt{x}) = f(\delta_S(x))$.

Representations and realized maps form a category $\mathsf{Rep}(\mathbb{A}, \mathbb{A}')$, which is equivalent to $\mathsf{Mod}(\mathbb{A}, \mathbb{A}')$.

When we transpose $\Vdash_S$ for a a general assembly S the result is a ***multi-valued representation***, which is a map $\delta_S : \mathbb{A}_{|S|} \rightrightarrows \mathcal{P}(|S|)$ that takes each $r \in \mathbb{A}_{|S|}$ to the (possibly empty) set of elements it realizes,

$$\delta_S(r) = \{x \in |S| \mid r \Vdash_S x\}.$$

The map is surjective in the sense that for every $x \in |S|$ there is $r \in \mathbb{A}_{|S|}$ such that $x \in \delta_S(r)$.

To summarize, there are three ways of specifying the realizability structure of an assembly: with a realizability relation $\Vdash_S$, an existence predicate E_S, and a multi-valued representation δ_S. Each determines the other two by

$$r \Vdash_S x \iff r \in \mathsf{E}_S(x) \iff x \in \delta_S(r).$$

3.3.3 Partial equivalence relations

This formulation only works for modest sets. With each modest set S we may associate a partial equivalence relation[6] (per) $\approx_S$ on $\mathbb{A}_{|S|}$ which relates q and r when they realize the same element:

6: A ***partial equivalence relation*** is a transitive symmetric relation.

$$q \approx_S r \iff \exists x \in |S|.\, q \Vdash_S x \wedge r \Vdash_S x.$$

The pair $(\|S\|, \approx_S)$ suffices for the reconstruction of the original modest set, up to isomorphism, which we show next.

Let $(\mathbb{A}, \mathbb{A}')$ be a tpca with a sub-tpca. A ***partial equivalence relation*** on $\mathbb{A}$ is a pair $S = (\|S\|, \approx_S)$ where $\|S\|$ is a type and $\approx_S$ is a transitive and symmetric relation on $\mathbb{A}_{|S|}$. A realizer $r \in \mathbb{A}_{|S|}$ is ***total*** if $r \approx_S r$. The set of total realizers is denoted by $\|S\| = \{r \in \mathbb{A}_{|S|} \mid r \approx_S r\}$. Each $r \in \|S\|$ determines the equivalence class $[r]_S = \{q \in \mathbb{A}_{|S|} \mid r \approx_S q\}$.

An ***extensional realizer*** between pers S and T is $p \in \mathbb{A}'_{\|S\| \to \|T\|}$ such that, for all $q, r \in \mathbb{A}_{|S|}$, if $q \approx_S r$ then $p\,q\downarrow$, $p\,r\downarrow$, and $p\,q \approx_T p\,r$. Extensional realizers p and p' are ***equivalent*** when $q \approx_S r$ implies $p\,q \approx_T p'\,r$.

Pers and equivalence classes of extensional realizers form a category $\mathsf{Per}(\mathbb{A}, \mathbb{A}')$ whose objects are pers on $\mathbb{A}$ and morphisms are equivalence classes of extensional realizers. The composition of $[p] : S \to T$ and $[q] : T \to U$ is $[q \circ p] : S \to U$ where $q \circ p = \langle x^{\|S\|} \rangle\, q\,(p\,x)$. The identity morphism $\mathrm{id}_S : S \to S$ is represented by $\langle x^{\|S\|} \rangle\, x$. It is easy to check that this forms a category.

Let S and T be pers over $(\mathbb{A}, \mathbb{A}')$. A morphism between them may be alternatively described as a function $f : \|S\|/{\approx_S} \to \|T\|/{\approx_T}$ between the equivalence classes for which there exists a realizer $p \in \mathbb{A}'_{\|S\| \to \|T\|}$ that tracks it: for every equivalence class $[r]_S$, $p\,r\downarrow$ and $[p\,r]_T = f([r]_S)$.

Lemma 3.3.1 *Suppose S is an assembly, T is a set, and $f : T \to S$ is a bijection. Then S is isomorphic to $T = (T, \|S\|, \Vdash_T)$ where $r \Vdash_T x$ is defined as $r \Vdash_S f(x)$.*

Proof. The map f is a morphism from S to T because it is tracked by $\langle x^{\|S\|} \rangle\, x$. Similarly, f^{-1} is a morphism because it is also tracked by the same realizer. Obviously, f and f^{-1} are inverses of each other. □

Proposition 3.3.2 *The categories* $\mathsf{Mod}(\mathbb{A}, \mathbb{A}')$ *and* $\mathsf{Per}(\mathbb{A}, \mathbb{A}')$ *are equivalent.*

Proof. A modest set $(|S|, \|S\|, \Vdash_S)$ determines a per $(S, \approx_S)$, as described above. A morphism $f : S \to T$ which is tracked by $p \in \mathbb{A}'_{\|S\| \to \|T\|}$ determines a morphism of pers $[p] : (S, \approx_S) \to (T, \approx_T)$. This defines a functor $F : \mathsf{Mod}(\mathbb{A}, \mathbb{A}') \to \mathsf{Per}(\mathbb{A}, \mathbb{A}')$.

In the other direction the functor $G : \mathsf{Per}(\mathbb{A}, \mathbb{A}') \to \mathsf{Mod}(\mathbb{A}, \mathbb{A}')$ sends a per $(\|T\|, \approx_T)$ to the modest set $(\|T\|/{\approx_T}, \|T\|, \Vdash_T)$ whose realizability relation is

$$r \Vdash_T [q] \iff r \approx_T q.$$

A morphism $[p] : (S, \approx_S) \to (T, \approx_T)$ is mapped to the map $G[p] : \|S\|/{\approx_S} \to \|T\|/{\approx_T}$, defined by $G[p][r]_S = [p\,r]_T$, which is obviously tracked by p.

The functors F and G form an equivalence of categories. The composition $F \circ G$ is actually equal to identity, as is easily verified. A modest set $(|S|, \|S\|, \Vdash_S)$ is isomorphic to $G(F(S))$ by Lemma 3.3.1 applied to the bijection which takes an $x \in |S|$ to $[r]_{G(F(S))}$, where $r \in \mathbb{A}_{|S|}$ is any

realizer such that $r \Vdash_S x$. We leave the verification that the isomorphisms are natural as exercise. □

3.3.4 Equivalence relations

A per $(\|S\|, \approx_S)$ may be viewed as an equivalence relation on $\|S\| = \{r \in \mathbb{A}_{|S|} \mid r \approx_S r\}$. This gives us yet another equivalent formulation of modest sets, this time in terms of equivalence relations.

The category $\mathsf{Er}(\mathbb{A}, \mathbb{A}')$ of equivalence relations has as objects triples $(S, \|S\|, \equiv_S)$ where $\|S\|$ is a type, $S \subseteq \mathbb{A}_{|S|}$, and $\equiv_S$ is an equivalence relation on S. As in the case of pers, a morphism $(S, \|S\|, \equiv_S) \to (T, \|T\|, \equiv_T)$ is represented by an extensional realizer $p \in \mathbb{A}'_{\|S\| \to \|T\|}$.

The difference between pers and equivalence relations is mostly a bureaucratic one. Nevertheless, it is useful to know about $\mathsf{Er}(\mathbb{A}, \mathbb{A}')$ because sometimes we can describe it in enlightening alternative ways, e.g., in Subsection 3.5.2 we describe pers on the graph model as equivalence relations on topological spaces.

3.4 Applicative functors

Categories of assemblies themselves form a category whose morphisms are functors induced by simulations, known as *applicative functors*. These were defined and studied by John Longley [23], and are the appropriate notion of morphisms of assemblies, as well as realizability toposes. We review their definition and several basic results about them, which we cannot do without assuming some knowledge of basic category theory.

A simulation $\rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F}, \mathbb{F}')$ induces an ***applicative functor***

$$\widehat{\rho} : \mathsf{Asm}(\mathbb{E}, \mathbb{E}') \longrightarrow \mathsf{Asm}(\mathbb{F}, \mathbb{F}')$$

which maps an assembly $S = (S, \Vdash_S)$ over $(\mathbb{E}, \mathbb{E}')$ to the assembly $\widehat{\rho}\, S$ over $(\mathbb{F}, \mathbb{F}')$, whose underlying set is S and

$$q \Vdash_{\widehat{\rho S}} x \iff \exists r \in \mathbb{E}.\, q \in \rho[r] \land r \Vdash_S x.$$

That is, $\widehat{\rho}$ replaces realizers in $\mathbb{E}$ with their simulations in $\mathbb{F}$.

Suppose $r \in \mathbb{F}'$ is a realizer for ρ. An assembly map $f : S \to T$, realized by $\mathtt{f} \in \mathbb{E}'$, is mapped by $\widehat{\rho}$ to the same underlying map $\widehat{\rho}\, f = f$, which is realized by $r\, \mathtt{g}$ for any $\mathtt{g} \in \rho[\mathtt{f}]$.

Exercise 3.4.1 Prove that and applicative functor induced by a *discrete* simulation restricts to modest sets.

The properties of the induced applicative functor depend on the properties of the simulation, as follows. (We presume existence of categorical structure on assemblies which will only be established in **??**.)

Proposition 3.4.1 *Let* $\rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\text{pca}} (\mathbb{F}, \mathbb{F}')$ *be a simulation.*

1. *The functor* $\widehat{\rho}$ *is faithful and it preserves finite limits.*
2. *If* ρ *is projective then* $\widehat{\rho}$ *preserves projective objects.*
3. *If* ρ *is decidable then* $\widehat{\rho}$ *preserves finite colimits and the natural numbers object.*

Proof. The functor $\widehat{\gamma}$ preserves finite limits by [23, Proposition 2.2.2]. It is faithful because it acts trivially on morphisms. For the second claim see [23, Theorem 2.4.12], and for the third one [23, Theorem 2.4.19]. □

Adjunctions and equivalences between simulations carry over to the induced morphisms.

Theorem 3.4.2 *Consider simulations*

$$\delta : (\mathbb{E}, \mathbb{E}') \xrightarrow{\mathsf{pca}} (\mathbb{F}, \mathbb{F}'), \qquad \gamma : (\mathbb{F}, \mathbb{F}') \xrightarrow{\mathsf{pca}} (\mathbb{E}, \mathbb{E}').$$

1. *If* $\gamma \dashv \delta$ *is an adjoint pair, then* $\widehat{\gamma} \dashv \widehat{\delta}$ *is an adjunction of functors.*
2. *If* $\gamma \dashv \delta$ *is an adjoint inclusion then the counit of the adjunction* $\widehat{\gamma} \dashv \widehat{\delta}$ *is a natural isomorphism.*
3. *If* $\gamma \dashv \delta$ *is an adjoint retraction then the unit of the adjunction* $\widehat{\gamma} \dashv \widehat{\delta}$ *is a natural isomorphism.*
4. *If* γ *and* δ *form an equivalence, then so do* $\widehat{\gamma}$ *and* $\widehat{\delta}$*.*

Proof. The first three claims are subsumed by the easy part of [23, Proposition 2.5.9], except that we are using simulations on pcas with sub-pcas. Also, we are restricting attention to categories of assemblies rather than realizability toposes, but this is not a problem because by applicative functors on realizability toposes restrict to assemblies.

That equivalences of simulations induce equivalences of categories is shown in [23, Theorem 2.5.6]. □

The construction of assemblies and induced applicative functors extends to a 2-functor between 2-categories. Indeed, given $\gamma, \delta : (\mathbb{E}, \mathbb{E}') \xrightarrow{\mathsf{pca}} (\mathbb{F}, \mathbb{F}')$ such that $\gamma \leq \delta$, there is an induced natural transformation $\zeta : \widehat{\gamma} \Rightarrow \widehat{\delta}$ defined by

$$\zeta_S = \mathrm{id}_S : \widehat{\gamma}\, S \longrightarrow \widehat{\delta}\, S.$$

This is a valid definition, for if $t \in \mathbb{F}'$ is a translation witnessing $\gamma \leq \delta$, then t tracks every ζ_S, and the naturality condition is trivial.

An applicative functor induced by $\rho : (\mathbb{E}, \mathbb{E}') \xrightarrow{\mathsf{pca}} (\mathbb{F}, \mathbb{F}')$ commutes up to natural isomorphism with the constant assembly functor ∇ from Subsection 3.2.4,

$$\begin{array}{ccc} \mathsf{Asm}(\mathbb{E}, \mathbb{E}') & \xrightarrow{\widehat{\rho}} & \mathsf{Asm}(\mathbb{F}, \mathbb{F}') \\ & {}_{\nabla}\nwarrow \quad \nearrow_{\nabla} & \\ & \mathsf{Set} & \end{array}$$

as well as with the underlying set functor Γ,

$$\begin{array}{ccc} \mathsf{Asm}(\mathbb{E},\mathbb{E}') & \xrightarrow{\widehat{\rho}} & \mathsf{Asm}(\mathbb{F},\mathbb{F}') \\ & {\scriptstyle\Gamma}\searrow \quad \swarrow{\scriptstyle\Gamma} & \\ & \mathsf{Set} & \end{array}$$

See [23, Proposition 2.2.4] for the proof.

3.5 Schools of Computable Mathematics

Realizability is a unifying framework for several "schools" of computable mathematics [7]. To get a particular variation we just choose an appropriate tpca with sub-tpca. We look at some of them and relate the traditional terminology and notions to ours. The material should be of interest to those who care about computability on topological spaces.

[7]: Bridges et al. (1987), *Varieties of Constructive Mathematics*

3.5.1 Recursive Mathematics

Recursive Mathematics, also known as *type one effectivity* or *Russian constructivism* [20, 35, 39], is computable mathematics done with type 1 machines, cf. Subsection 2.1.1. In our settings it corresponds to the category $\mathsf{Rep}(\mathbb{K}_1)$ of representations over the first Kleene algebra.

[20]: Kušner (1984), *Lectures on Constructive Mathematical Analysis*
[35]: Spreen (1998), "On Effective Topological Spaces"
[39]: Šanin (1968), *Constructive Real Numbers and Constructive Function Spaces*

An object of $\mathsf{Rep}(\mathbb{K}_1)$ is called a ***numbered set*** [11]. It is a pair (S, δ_S), where S is a set $\delta_S : \mathbb{N} \rightharpoonup S$ a partial surjection, called a ***numbering*** of S. A function $f : S \to T$ is ***realized*** by $n \in \mathbb{N}$ when, for all $m \in \mathsf{dom}(\delta_S)$,

[11]: Eršov (1999), "Handbook of Computability Theory"

$$\boldsymbol{\varphi}_n(m)\downarrow \text{ and } \delta_T(\boldsymbol{\varphi}_n(m)) = f(\delta_S(m)).$$

A numbered set (S, δ_S) has countably many elements because it is covered by the countable set $\mathsf{dom}(\delta_S)$. This is sometimes considered a disadvantage and a reason for preferring type 2 machines, which are able to compute with uncountable structures such as real numbers. However, *internally* to the category the reals form a Cauchy-complete archimedean ordered field, on top of which it is perfectly possible to develop a version of compute analysisg. One gets an unusual variant which is a rich source of counter-examples.

3.5.2 Equilogical spaces

An equilogical space is a topological space with an equivalence relation [1]. We study equilogical spaces in some detail because they give us a general theory of computable maps between countably-based spaces. In Subsection 3.5.5 we relate equilogical spaces to Type Two Effectivity, which is another school of computability on general topological spaces.

[1]: Bauer et al. (1998), "Equilogical Spaces"

Recall that a topological space is ***countably based*** or ***2-countable*** if it has a countable topological basis. Equivalently, a space is countably based when it has a countable *subbasis*. We prefer to work with subbases because they simplify the treatment of computable maps between spaces. Thus we define a countably based space to be a pair $(X, (U_i)_{i\in\mathbb{N}})$ where

X is a topological space and $(U_i)_{i \in \mathbb{N}}$ is an enumeration of *subbasic* open sets. These generate the topology of X by taking finite intersections and arbitrary unions. While we usually omit an explicit mention of the subbasis $(U_i)_{i \in \mathbb{N}}$, we do insist that a countably based space always be given *together with* a particular subbasis. This allows us to avoid the axiom of choice.

The graph model $\mathbb{P}$ is a countably based space. We always take its subbasic open sets to be $\uparrow n = \{A \subseteq \mathbb{N} \mid n \in A\}$, for $n \in \mathbb{N}$.

A ***(countably based) equilogical space*** $(X, (U_i)_{i \in \mathbb{N}}, \equiv_X)$ is a countably based topological space $(X, (U_i)_{i \in \mathbb{N}})$ with an equivalence relation $\equiv_X$. We usually do not bother writing the subbasis $(U_i)_{i \in \mathbb{N}}$. The canonical quotient map $q_X : X \to X/{\equiv_X}$ maps each $x \in X$ to its equivalence class $[x]_X$. A morphism $f : (X, \equiv_X) \to (Y, \equiv_Y)$ is a map $f : X/{\equiv_X} \to Y/{\equiv_Y}$ between equivalence classes for which there exists a continuous $g : X \to Y$ such that

$$\begin{array}{ccc} X & \xrightarrow{g} & Y \\ {\scriptstyle q_X}\downarrow & & \downarrow{\scriptstyle q_Y} \\ X/{\equiv_X} & \xrightarrow[f]{} & Y/{\equiv_Y} \end{array}$$

commutes, i.e., $f([x]_X) = [g(x)]_Y$ for all $x \in X$. Morphisms compose as expected. The category of equilogical spaces and morphisms between them is denoted by Equ.

A countably based topological space X may be construed as an equilogical space $(X, =_X)$ with equality as the equivalence relation. A morphism $f : (X, =_X) \to (Y, =_Y)$ is the same thing as a continuous map $f : X \to Y$ so that we have a full and faithful embedding $\omega\mathsf{Top} \to \mathsf{Equ}$ of the category $\omega\mathsf{Top}$ of countably based spaces into the category Equ.

Let us show that Equ and $\mathsf{Asm}(\mathbb{P})$ are equivalent. A ***pre-embedding*** $e : X \to Y$ between topological spaces is a continuous map such that $e^{-1} : \mathcal{O}(Y) \to \mathcal{O}(X)$ is surjective. For T_0-spaces this is equivalent to e being an embedding. If $(U_i)_{i \in I}$ is a topological subbasis for Y and $e : X \to Y$ is a pre-embedding, then $(e^*(U_i))_{i \in I}$ is a topological subbasis for X.

Theorem 3.5.1 (Embedding Theorem for $\mathbb{P}$) *A space X may be pre-emebedded in $\mathbb{P}$ if, and only if, it is countably based.*

Proof. Here $\mathbb{P}$ is equipped with the Scott topology. If $e : X \to \mathbb{P}$ is a pre-embedding then the open sets $U_n = e^*(\uparrow n)$ form a countable subbasis for X.

Conversely, suppose $(U_n)_{n \in \mathbb{N}}$ is a countable subbasis for X. Define the map $e_X : X \to \mathbb{P}$ by

$$e_X(x) = \{n \in \mathbb{N} \mid x \in U_n\}.$$

We claim that e_X is a pre-embedding. It is continuous because $e_X^*(\uparrow n) = U_n$. Let $V \subseteq X$ be open. Then V is a union of finite intersections of subbasic opens,

$$V = \bigcup_i U_{n_{i,1}} \cap \cdots \cap U_{n_{i,k_i}}$$

Now

$$\begin{aligned} e_X^*\left(\bigcup_i \uparrow\{n_{i,1},\dots,n_{i,k_i}\}\right) &= \bigcup_i e_X^*(\uparrow\{n_{i,1},\dots,n_{i,k_i}\}) \\ &= \bigcup_i e_X^*(\uparrow n_{i,1}\cap\dots\cap\uparrow n_{i,k_i}) \\ &= \bigcup_i e_X^*(\uparrow n_{i,1})\cap\dots\cap e_X^*(\uparrow n_{i,k_i}) \\ &= \bigcup_i U_{n_{i,1}}\cap\dots\cap U_{n_{i,k_i}} \\ &= V, \end{aligned}$$

therefore e_X^* is surjective, as required. □

The pre-embedding $e_X : X \to \mathbb{P}$ is called the ***(subbasic) neighborhood filter*** because $e_X(x)$ is just the set of (indices of) subbasic neighborhoods of x. Henceforth $e_X : X \to \mathbb{P}$ will always denote the subbasic neighobrhood filter.

Theorem 3.5.2 (Extension Theorem for $\mathbb{P}$) *Suppose $e : X \to Y$ is a pre-embedding and $f : X \to \mathbb{P}$ continuous. Then f has a continuous extension $g : Y \to \mathbb{P}$ along e.*

Proof. Consider the map $g : Y \to \mathbb{P}$ defined by

$$g(y) = \bigcup_{U\in\mathcal{O}(Y)} \left\{\bigcap_{z\in e^*(U)} f(z) \mid y \in U\right\}.$$

It is continuous because

$$\begin{aligned} g^*(\uparrow n) &= \{y \in Y \mid n \in g(y)\} \\ &= \{y \in Y \mid \exists U.\, \mathcal{O}(Y) y \in U \wedge \forall z.\, e^*(U) n \in f(z)\} \\ &= \bigcup_{U\in\mathcal{O}(Y)} \{U \mid \forall z.\, e^*(U) n \in f(z)\}. \end{aligned}$$

Let us show that $g(e(x)) = f(x)$ for all $x \in X$. Consider the value

$$g(e(x)) = \bigcup_{U\in\mathcal{O}(Y)} \left\{\bigcap_{z\in e^*(U)} f(z) \mid e(x) \in U\right\}.$$

Because $f(x)$ appears in every intersection, each of them is contained in $f(x)$, which shows that $g(e(x)) \subseteq f(x)$. Suppose $n \in f(x)$. Because e is a pre-embedding there exists $W \in \mathcal{O}(Y)$ such that $f^*\uparrow n = e^*(W)$. If $z \in X$ and $e(z) \in W$ then $z \in e^*(W) = f^*(\uparrow n)$, hence $n \in f(z)$. The intersection $\bigcap\{f(z) \mid z \in X \wedge e(z) \in W\}$ contains n and so $n \in g(e(x))$. We proved that $f(x) \subseteq g(e(x))$, therefore $f(x) = g(e(x))$. □

The Embedding and Extension theorems now give us the desired equivalence [27].

[27]: Menni et al. (2002), "Topological and Limit-Space Subcategories of Countably-Based Equilogical Spaces"

Proposition 3.5.3 *The categories* Equ *and* $\mathsf{Asm}(\mathbb{P})$ *are equivalent.*

Proof. Suppose $(X, \equiv_X)$ is an equilogical space, and let $e_X : X \to \mathbb{P}$ be the subbasic neighborhood filter pre-embedding. Define the assembly $F(X) = (X/{\equiv_X}, \mathsf{E}_{F(X)})$ by $\mathsf{E}_{F(x)} = \{e_X(y) \mid x \equiv_X y\}$. To make F into a functor we define $F(f) = f$ for a morphism $f : (X, \equiv_X) \to (Y, \equiv_Y)$. If f

is tracked by $g : X \to Y$, then $F(f)$ is realized by a continuous extension of $e_Y \circ g : X \to \mathbb{P}$ along e_X, which exists by Extension Theorem 3.5.2.

The functor $G : \mathsf{Asm}(\mathbb{P}) \to \mathsf{Equ}$ is defined as follows. An assembly (S, E_S) is mapped to the equilogical space $G(S) = (X_S, \equiv_{G(S)})$ whose underlying space is the set $X_S = \{(x, A) \in S \times \mathbb{P} \mid A \in \mathsf{E}_S(x)\}$, equipped with the unique topology that makes the projection $p : X_S \to \mathbb{P}$, $p : (x, A) \mapsto A$, a pre-embedding. Explicitly, the open subsets of X_S are the inverse images $p^*(U)$ of open subsets $U \subseteq \mathbb{P}$. Let $\equiv_{G(S)}$ be the equivalence relation defined by

$$(x, A) \equiv_{G(S)} (y, B) \iff x = y.$$

A morphism $f : (S, \Vdash_S) \to (T, \Vdash_T)$ which is tracked by $B \in \mathbb{P}_\#$ is mapped to $G(f) : X_S/{\equiv_{G(S)}} \to X_T/{\equiv_{G(T)}}$ defined by $G(f)([(x, A)]_{G(S)}) = [(f(x), B \cdot A)]_{G(T)}$.

We leave the verification that F and G form an equivalence of categories as exercise. □

By restricting to the T_0-spaces we obtain another equivalence. Let Equ_0 be the full subcategory of Equ in which the underlying topological spaces are T_0-spaces.

Proposition 3.5.4 *The categories* Equ_0 *and* $\mathsf{Mod}(\mathbb{P})$ *are equivalent.*

Proof. We verify that the equivalence functors F and G from the proof of Proposition 3.5.3 restrict to Equ_0 and $\mathsf{Mod}(\mathbb{P})$. If $(X, \equiv_X)$ is an equilogical space whose underlying space X is T_0, then the pre-embedding $e : X \to \mathbb{P}$ is actually an embedding. Because it is injective the assembly $F(X)$ is modest. This shows that F restricts to a functor $\mathsf{Equ}_0 \to \mathsf{Mod}(\mathbb{P})$. To see that G restricts to a functor $\mathsf{Mod}(\mathbb{P}) \to \mathsf{Equ}_0$, observe that, for a modest assembly (S, E_S), the projection $X_S \to \mathbb{P}$ is an embedding, therefore X_S is a T_0-space. □

3.5.3 Computable countably based spaces

We have so far studied the *continuous* version of realizability over the graph model in which the realizers for morphisms may be arbitrary continuous maps. But what about the *mixed* version $\mathsf{Asm}(\mathbb{P}, \mathbb{P}_\#)$, is it also equivalent to a version of equilogical spaces? To see that the answer to the question is affirmative, we first need to define computable maps between countably based spaces.

Recall from Section 2.2 that an enumeration operator $g : \mathbb{P} \to \mathbb{P}$ is computable when its graph $\Gamma(g)$ is a c.e. set. By Embedding Theorem 3.5.1, every countably based T_0-space X can be embedded in $\mathbb{P}$, and every continuous map $f : X \to Y$ can be extended to an enumeration operator $g : \mathbb{P} \to \mathbb{P}$, so that the following diagram commutes:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle e_X}\downarrow & & \downarrow{\scriptstyle e_Y} \\ \mathbb{P} & \xrightarrow{g} & \mathbb{P} \end{array}$$

We can define a ***computable continuous map*** $f : X \to Y$ to be a continuous map for which there exists a computable enumeration operator $g : \mathbb{P} \to \mathbb{P}$ which makes the above diagram commute. This idea gives the following definition of computable continuous maps.

Definition 3.5.1 A continuous map $f : X \to Y$ between countably based spaces $(X, (U_i)_{i\in\mathbb{N}})$ and $(Y, (V_j)_{j\in\mathbb{N}})$ is ***computable*** when there exists a c.e. set $F \subseteq \mathbb{N} \times \mathbb{N}$ such that:

1. F is monotone in the first argument: if $A \subseteq B \ll \mathbb{N}$ and $\langle \ulcorner A \urcorner, m\rangle \in F$ then $(\ulcorner B \urcorner, m) \in F$.
2. F approximates f: if $(\ulcorner\{i_1, \ldots, i_n\}\urcorner, m) \in F$ then $f(U_{i_1} \cap \cdots \cap U_{i_n}) \subseteq V_m$.
3. F converges to f: if $f(x) \in V_m$ then there exist $i_1, \ldots, i_n$ such that $x \in U_{i_1} \cap \cdots \cap U_{i_n}$ and $(\ulcorner\{i_1, \ldots, i_n\}\urcorner, m) \in F$.

The relation F is called a ***c.e. realizer*** for f. We also say that F ***tracks*** f. The category of countably based spaces and computable continuous maps is denoted by $\omega\mathsf{Top}_{\#}$.

The category $\omega\mathsf{Top}_{\#}$ is well-defined. The identity map $\mathrm{id}_X : X \to X$ is tracked by the relation I_X, defined by

$$I_X = \{(\ulcorner\{i_1, \ldots, i_n\}\urcorner, m) \in \mathbb{N} \times \mathbb{N} \mid m \in \{i_1, \ldots, i_n\}\}.$$

The composition of computable maps $f : X \to Y$ and $g : Y \to Z$, which are tracked by F and G respectively, is again a computable map $g \circ f : X \to Z$ because it has a c.e. realizer H defined by

$$(\ulcorner\{j_1, \ldots, j_k\}\urcorner, \ell) \in H \iff$$
$$\exists i_1, \ldots, i_n . \mathbb{N}(\ulcorner\{i_1, \ldots, i_n\}\urcorner, \ell) \in G \wedge \textstyle\bigwedge_{s=1}^{n} (\ulcorner\{j_1, \ldots, j_k\}\urcorner, i_s) \in F.$$

The monotonicity condition in Definition 3.5.1 is redundant, for if F is an c.e. relation that satisfies the second and the third condition, then we can recover monotonicity by defining a new relation F' by

$$F' = \{(\ulcorner A \urcorner, m) \in \mathbb{N} \times \mathbb{N} \mid \textstyle\bigvee_{B\subseteq A} (\ulcorner B \urcorner, m) \in F\}.$$

It is easy to see that F' satisfies all three conditions and realizes the same function as f.

A point $x \in X$ is ***computable*** when the map $\{\star\} \to X$ which maps $\star$ to x is computable. This is equivalent to requiring that $e_X(x) = \{i \in \mathbb{N} \mid x \in U_i\}$ is a c.e. set.

Next, we prove effective versions of the Embedding and Extension Theorems.

Theorem 3.5.5 (Computable Embedding Theorem) *Every countably based space can be computably pre-embedded into $\mathbb{P}$.*

Proof. We just need to provide a c.e. realizer E_X for the neighborhood filter $e_X : X \to \mathbb{P}$. It is

$$E_X = \{(\ulcorner\{i_1, \ldots, i_n\}\urcorner, m) \in \mathbb{N} \times \mathbb{N} \mid m \in \{i_1, \ldots, i_n\}\}.$$

This is obviously a c.e. relation which is monotone in the first argument. The second condition for the c.e. realizer E_X is

$$(\ulcorner\{i_0, \ldots, i_n\}\urcorner, m) \in E_X \implies e_X(U_{i_1}, \ldots, U_{i_n}) \subseteq \uparrow m,$$

which clearly holds. Suppose $e_X(x) \in \uparrow m$. Then $x \in U_m$, and $(\ulcorner\{m\}\urcorner, m) \in E_X$, which proves the third condition. □

Theorem 3.5.6 (Computable Extension Theorem) *Let X and Y be countably based topological spaces and $f : X \to Y$ a computable map between them. Then there exists a computable map $g : \mathbb{P} \to \mathbb{P}$ such that the following diagram commutes:*

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle e_X}\downarrow & & \downarrow{\scriptstyle e_Y} \\ \mathbb{P} & \xrightarrow[g]{} & \mathbb{P} \end{array}$$

Proof. The maps e_X and e_Y are the computable embeddings from Embedding Theorem 3.5.5. Let F be a c.e. realizer for f. We define the map $g : \mathbb{P} \to \mathbb{P}$ by specifying its graph to be F, i.e.,

$$g(A) = \{m \in \mathbb{N} \mid \exists i_1, \ldots, i_n . \mathbb{N}(\ulcorner\{i_1, \ldots, i_n\}\urcorner, m) \in F\}.$$

All we have to show is that this choice of g makes the diagram commute. For any $x \in X$,

$$\begin{aligned} & m \in g(e_X(x)) \\ & \quad \Leftrightarrow \exists i_1, \ldots, i_n \in \mathbb{N}. \, x \in U_{i_1} \cap \cdots \cap U_{i_n} \wedge (\ulcorner\{i_1, \ldots, i_n\}\urcorner, m) \in F \\ & \quad \Leftrightarrow f(x) \in V_m \\ & \quad \Leftrightarrow m \in e_Y(f(x)). \end{aligned}$$

The second equivalence is implied from left to right by the second condition in Definition 3.5.1, and from right to the left by the third condition. □

We should point out that the computable continuous maps, as defined here, work at the level of open sets, i.e., a c.e. realizer F for $f : X \to Y$ operates on the (indices) of subbasic open sets. Therefore, F does not distinguish points that share the same neighborhoods, even though f might. This is not an issue with T_0-spaces in which points are distinguished by their neighborhoods.

3.5.4 Computable equilogical spaces

With a notion of computable maps between spaces at hand, we can define the computable equilogical spaces just like the ordinary ones, except that we replace continuous maps by their computable versions.

Definition 3.5.2 A morphism $f : (X, \equiv_X) \to (Y, \equiv_Y)$ between equi-

logical spaces is ***computable*** if there exists a computable continuous map $g : X \to Y$ which tracks f. The category of equilogical spaces and computable morphisms between them is denoted by $\mathsf{Equ}_\#$.

We check that we got the definition right by proving that $\mathsf{Equ}_\#$ is equivalent to $\mathsf{Asm}(\mathbb{P}, \mathbb{P}_\#)$.

Proposition 3.5.7 *The categories* $\mathsf{Asm}(\mathbb{P}, \mathbb{P}_\#)$ *and* $\mathsf{Equ}_\#$ *are equivalent.*

Proof. The proof goes just as the proof of Proposition 3.5.3 that Equ and $\mathsf{Asm}(\mathbb{P})$ are equivalent. The only difference is that we refer to Embedding Theorem 3.5.5 and Extension Theorem 3.5.6 in order to extend computable maps to computable enumeration operators. □

The category $\omega\mathsf{Top}_\#$ of countably based spaces and computable maps is embedded fully and faithfully into $\mathsf{Equ}_\#$. The embedding works as in the continuous case: a topological space X is mapped to the equilogical space $(X, =_X)$, and a computable continuous map $f : X \to Y$ is the same thing as a morphism $f : (X, =_X) \to (Y, =_Y)$.

3.5.5 Type Two Effectivity

A popular realizability model is Kleene-Vesley ***function realizability*** [18], also known as ***Type Two Effectivity (TTE)*** [6, 38]. As the names say, it is the model of realizability based on functions and type 2 machines.

[18]: Kleene et al. (1965), *The foundations of intuitionistic mathematics. Especially in relation to recursive functions.*

[6]: Brattka et al. (2021), *Handbook of Computability and Complexity in Analysis*

[38]: Weihrauch (2000), *Computable Analysis*

TTE is traditionally expressed as a theory of representations. There are actuallly three variations:

1. $\mathsf{Rep}(\mathbb{B}, \mathbb{B})$ is the *continuous* version in which maps are realized by continuous realizers.
2. $\mathsf{Rep}(\mathbb{B}, \mathbb{B}_\#)$ is the *relative* version.
3. $\mathsf{Rep}(\mathbb{B}_\#, \mathbb{B}_\#)$ is the *computable* version in which all realizers must be computable.

Mostly only the first two of these are used. Sometimes multi-valued representations are considered also, and for these we need to move to the larger category of assemblies $\mathsf{Asm}(\mathbb{B}, \mathbb{B}_\#)$.

Specifically, a representation (S, δ_S) over the Baire space is a partial surjection $\delta_S : \mathbb{B} \to S$. When $\delta_S(\alpha) = x$ we say that α is a δ_S***-name*** of x. A ***(continuously) realized map*** $f : (S, \delta_S) \to (T, \delta_T)$ is a map $f : S \to T$ for which there exists a partial continuous map $g : \mathbb{B} \rightharpoonup \mathbb{B}$ such that $f(\delta_S(\alpha)) = \delta_T(g(\alpha))$ for all $\alpha \in \mathsf{dom}(\delta_S)$. If the realizer g is computable we say that f is ***computably realized***. Recall that a computable g corresponds to a type 2 machine which converts a δ_S-name of x to a δ_T-name of $f(x)$.

In the case of equilogical spaces there was a straightforward way of turning a topological space into an equilogical space. The present situation is less obvious. One idea is to represent a topological space X by a representation $\delta_X : \mathbb{B} \rightharpoonup X$ for which δ_X is a quotient map. However, this is too weak a requirement. To see this, suppose $\delta_X : \mathbb{B} \rightharpoonup X$ and $\delta_Y : \mathbb{B} \rightharpoonup Y$ are representations of topological spaces with δ_X and δ_Y

topological quotient maps. Then a continuous map $f : X \to Y$ may be lifted to $g : \mathbb{B} \rightharpoonup Y$, as in the diagram

$$\begin{array}{ccc} \mathbb{B} & \overset{h?}{\dashrightarrow} & \mathbb{B} \\ {\scriptstyle \delta_X}\downarrow & \searrow^{g} & \downarrow{\scriptstyle \delta_Y} \\ X & \underset{f}{\longrightarrow} & Y \end{array}$$

because δ_X is a quotient map. But to make f into a morphism we need a continuous realizer $h : \mathbb{B} \rightharpoonup \mathbb{B}$ on the top line, which however might not exist. A stronger property is required.

Definition 3.5.3 Suppose X is a topological space and $\delta_X : \mathbb{B} \rightharpoonup X$ is a representation and a topological quotient map. Then δ_X is ***admissible*** if every continuous $g : \mathbb{B} \rightharpoonup X$ has a continuous lifting $h : \mathbb{B} \rightharpoonup \mathbb{B}$ such that $\delta_X(h(\alpha)) = g(\alpha)$ for all $\alpha \in \mathsf{dom}(g)$.

Admissible representations are a central concept in TTE because they are very well behaved. For example, if $\delta_X : \mathbb{B} \rightharpoonup X$ and $\delta_Y : \mathbb{B} \rightharpoonup Y$ are admissible, the continuous maps $f : X \to Y$ coincide with the realized maps.

The spaces which have admissible representations have been studied in depth [32]. We only mention a basic result whose proof is not too complicated.

[32]: Schröder (2021), "Handbook of Computability and Complexity in Analysis"

Proposition 3.5.8 *Every countably based T_0-space has an admissible representation.*

Proof. Suppose $(X, (U_i)_{i \in \mathbb{N}})$ is a countably based T_0-space, and let $e_X : X \to \mathbb{P}$ be the neighborhood filter. Define the representation $\delta_X : \mathbb{B} \rightharpoonup X$ by

$$\delta_X(\alpha) = x \iff e_X(x) = \{\alpha(n) \mid n \in \mathbb{N}\}.$$

In words, α is a δ_X-name for x when it enumerates the (indices of) subbasic open neighborhoods of x. Because X is a T_0-space, any α enumerates the subbasic neighborhood filter of at most one x, hence δ_X is single-valued. We leave admissibility of δ_X as an exercise. □

A relationship between equilogical spaces and TTE

In Subsection 2.8.3 we constructed an adjoint retraction

$$(\mathbb{P}, \mathbb{P}_\#) \underset{\iota}{\overset{\delta}{\rightleftarrows}} (\mathbb{B}, \mathbb{B}_\#)$$

where

$$\iota\, \alpha = \{\ulcorner a \urcorner \mid a \in \mathbb{N}^* \land a \sqsubseteq \alpha\}$$

and

$$\delta[A] = \{\alpha \in \mathbb{B} \mid A = \{n \in \mathbb{N} \mid \exists k \in \mathbb{N}.\, \alpha\, k = n + 1\}\}.$$

The induced functors $\widehat{\iota}$ and $\widehat{\delta}$ give an adjunction between equilogical spaces and TTE, see [3] for details.

[3]: Bauer (2002), "A Relationship between Equilogical Spaces and Type Two Effectivity"

3.6 The categorical structure of assemblies

In their everyday lives mathematicians use a limited set of constructions on sets: products, disjoint sums, subsets, quotient sets, images, function spaces, inductive and coinductive definitions, powersets, unions, intersections, and complements. For all of these, except powersets, there are analogous constructions of assemblies. Therefore, many familiar set-theoretic constructions carry over to assemblies.

3.6.1 Cartesian structure

A construction which makes a new set, space, or an algebraic structure from old ones is usually characterized by a *universal property* which determines it up to isomorphism. The universal property is shared among versions of the same construction in different categories. We start slowly with an easy one, the binary product.

Recall the definition of a ***(binary) product*** in a category: the product of objects S and T is an object P with morphisms $p_1 : P \to S$ and $p_2 : P \to T$, satisfying the following universal property: for all morphisms $f : U \to S$ and $g : U \to T$ there is a unique morphism $h : U \to P$ making the following diagram commute:

$$\begin{array}{ccccc} & & U & & \\ & \swarrow^{f} & \downarrow h & \searrow^{g} & \\ S & \xleftarrow{p_1} & P & \xrightarrow{p_2} & T \end{array}$$

The products (P, p_1, p_2) is determined uniquely up to a unique isomorphism. For suppose we had another product (Q, q_1, q_2) of S and T. By the universal property of P there is a map $h : P \to Q$ such that $p_1 \circ h = q_1$ and $p_2 \circ h = q_2$. Similarly, by the universal property of Q there is $k : Q \to P$ such that $q_1 \circ k = p_1$ and $q_2 \circ k = p_2$. Now $h \circ k$ satisfies

$$p_1 \circ h \circ k = q_1 \circ k = p_1 \qquad \text{and} \qquad p_2 \circ h \circ k = q_2 \circ k = p_2.$$

Since id_P also satisfies $p_1 \circ \mathrm{id}_P = p_1$ and $p_2 \circ \mathrm{id}_P = p_2$, it follows by the uniqueness condition of the universal property that $h \circ k = \mathrm{id}_P$. A similar argument shows that $k \circ h = \mathrm{id}_Q$, hence P and Q are isomorphic.

A category ***has binary products*** if every pair of objects has a binary product. In most cases we can actually provide an operation $\times$ which maps a pair of objects S, T to a specifically given product $S \times T$ with corresponding projections. The unique map h determined by f and g is denoted by $\langle f, g \rangle$.

In the category Set the product is just the usual cartesian product of sets. In assemblies we need to worry about the underlying types and

realizability relations. Let us verify that the product of assemblies S and T is the assembly

$$S \times T = (|S| \times |T|, \|S\| \times \|T\|, \Vdash_{S\times T})$$

with the realizability relation

$$p \Vdash_{S\times T} (x, y) \iff \mathtt{fst}\, q \Vdash_S x \land \mathtt{snd}\, q \Vdash_T y.$$

and the projection maps $\pi_1 : |S| \times |T| \to |S|$, $\pi_1 : (x, y) \mapsto x$, and $\pi_2 : |S| \times |T| \to |T|$, $\pi_2 : (x, y) \mapsto y$, which are realized by fst and snd, respectively. To see that $(S \times T, \pi_1, \pi_2)$ has the universal property, suppose $f : U \to S$ and $g : U \to T$ are realized by $\mathtt{f} \in \mathbb{A}'_{\|U\|\to\|S\|}$ and $\mathtt{g} \in \mathbb{A}'_{\|U\|\to\|T\|}$, respectively. There is a unique map $h : |U| \to |S| \times |T|$ for which $f = \pi_1 \circ h$ and $g = \pi_2 \circ g$, namely $h(u) = \langle f, g\rangle(u) = (f\,u, g\,u)$. We only need a realizer for h, and $\mathtt{h} = \langle u^{\|U\|}\rangle\, \mathtt{pair}\,(\mathtt{f}\,u)\,(\mathtt{g}\,u)$ does the job:

$$\begin{aligned}
\mathtt{u} \Vdash_U u &\implies \mathtt{f\,u} \Vdash_S f\,u \land \mathtt{g\,u} \Vdash_T g\,u \\
&\implies \mathtt{pair}\,(\mathtt{f\,u})\,(\mathtt{g\,u}) \Vdash_{S\times T} (f\,u, g\,u) \\
&\iff \mathtt{h\,u} \Vdash_{S\times T} h(u).
\end{aligned}$$

We may form n-ary products $S_1 \times \cdots \times S_n$ for $n \geq 1$ as nested binary products. The case $n = 0$ corresponds to the ***terminal object***, which is an object $\mathsf{1}$ such that for every object S there is exactly one morphism $S \to \mathsf{1}$. In the category of sets the terminal object is (any) singleton set, say $\mathsf{1} = \{\star\}$. Then $\nabla\mathsf{1}$ is the terminal assembly, since for any assembly S the only map $S \to \mathsf{1}$ is realized. We denote the terminal assembly as $\mathsf{1}$.

We may also ask whether $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ has infinite products. The answer depends on the underlying tpcas $(\mathbb{A}, \mathbb{A}')$. We state without proof that $\mathsf{Asm}(\mathbb{P}, \mathbb{P}_\#)$ and $\mathsf{Asm}(\mathbb{B}, \mathbb{B}_\#)$ have countable products, whereas $\mathsf{Asm}(\mathbb{K}_1)$ does not.

Exercise 3.6.1 Why does $\mathsf{Asm}(\mathbb{K}_1)$ not have countable products?

Products are a special case of categorical limits. Two other common kinds of limits are equalizers and pullbacks. An ***equalizer*** of a pair of morphisms $f, g : S \to T$ is an object E with a morphism $e : E \to S$ such that e equalizes f and g, which means that $f \circ e = g \circ e$, and the following universal property is satisfied: if $k : K \to S$ also equalizes f and g then there exists a unique morphism $i : K \to E$ such that $k = e \circ i$:

$$\begin{array}{ccccc}
E & \xrightarrow{e} & S & \underset{g}{\overset{f}{\rightrightarrows}} & T \\
{\scriptstyle i}\uparrow & \nearrow{\scriptstyle k} & & & \\
K & & & &
\end{array}$$

Think of E as the solution-set of equation $f\,x = g\,x$. Indeed, in the category of sets the equalizer of functions $f, g : S \to T$ is the subset $E = \{x \in S \mid f\,x = g\,x\}$ and $e : E \to S$ is the subset inclusion. In the category of assemblies we need to augment this with realizers. The

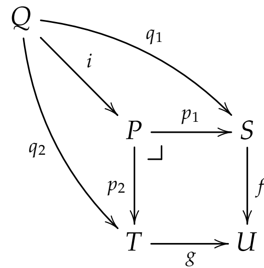
equalizer of $f, g : S \to T$ is

$$E = (\{x \in S \mid f\,x = g\,x\}, \|S\|, \Vdash_E) \tag{3.1}$$

where $\mathtt{x} \Vdash_E x$ if, and only if, $\mathtt{x} \Vdash_S x$. The map $e : |E| \to |S|$ is the subset inclusion, $e(x) = x$. It is realized by $\langle x^{\|S\|} \rangle\, x$. Clearly, e equalizes f and g.

Exercise 3.6.2 Verify that the above alleged equalizer has the correct universal property.

A ***pullback***, sometimes called ***fibered product***, is a combination of product and equalizer. Given morphisms $f : S \to U$ and $g : T \to U$, the pullback of f and g is an object P with morphisms $p_1 : P \to S$ and $p_2 : P \to T$ such that $f \circ p_1 = g \circ p_2$. Furthermore, if $q_1 : Q \to S$ and $q_2 : Q \to T$ are such that $f \circ q_1 = g \circ q_2$ then there is a unique $i : Q \to P$ which makes the following diagram commute:



The fact that P is a pullback is traditionally marked in a diagram with the "corner" symbol. In the category of assemblies the pullback of $f : S \to U$ and $g : T \to U$ is the assembly

$$P = (\{(x, y) \in S \times T \mid f\,x = g(y)\}, \|S\| \times \|T\|, \Vdash_P)$$

where $p \Vdash_P (x, y)$ if, and only if, $\mathtt{fst}\,p \Vdash_S x$ and $\mathtt{snd}\,p \Vdash_T y$.

Finite products, the terminal object, equalizers, and pullbacks are special cases of ***finite limits***. A category which has all finite limits is called ***cartesian*** or ***finitely complete***.[7]

7: We do not like much the still older terminology *left exact* or just *lex*.

Proposition 3.6.1 *The categories* $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *and* $\mathsf{Mod}(\mathbb{A}, \mathbb{A}')$ *are cartesian.*

Proof. It is well known that every finite limit may be constructed as a combination of a finite product and an equalizer, hence $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ is cartesian. It is easy to verify that finite products and equalizers of modest assemblies are again modest, therefore $\mathsf{Mod}(\mathbb{A}, \mathbb{A}')$ is cartesian. □

3.6.2 Cocartesian structure

Colimits are the dual of limits. In particular, the dual of products, terminal object, equalizers, and pullbacks are respectively coproducts, initial object, coequalizers, and pushouts. We study which of these exist in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$.

First we discuss (binary) coproducts of sets, also known as disjoint sums. For some reason there does not seem to be a well-established and practical notation for these, possibly because the related union operation is taken as primitive in set theory. The disjoint sum of sets S and T is usually defined as

$$S + T = (\{0\} \times S) \cup (\{1\} \times T).$$

The canonical injections $\iota_1 : S \to S + T$ and $\iota_2 : T \to S + T$ are the maps $x \mapsto (0, x)$ and $y \mapsto (1, x)$, respectively. A slight notational inconvenience arises when want to define a map $f : S + T \to U$ by cases $f_1 : S \to U$ and $f_2 : T \to U$. One possibility is to write

$$f\,u = \begin{cases} f_1(x) & \text{if } u = (0, x), \\ f_2(y) & \text{if } u = (1, y), \end{cases}$$

but this is seen rarely. In practice mathematicians prefer to assume, or shall we say *pretend*, that the sets S and T are disjoint and just write $S + T = S \cup T$. This allows us to get rid of the encoding by pairs,

$$f\,u = \begin{cases} f_1(u) & \text{if } u \in S, \\ f_2(u) & \text{if } u \in T. \end{cases}$$

Unlike people, computers do not pretend, and so as computer scientists we need notation that is actually correct. However, it is unnecessary encode the elements of a disjoin sum as pairs $(0, x)$ and $(1, y)$. Instead, we simply take the injections ι_1 and ι_2 as primitive *labels* that indicate which part of a disjoint sum we are referring to.[8] Thus, every element of $S + T$ is either of the form $\iota_1(x)$ for a unique $x \in S$, or $\iota_2(y)$ for a unique $y \in T$. In a specific case we may choose different, descriptive names for the injections.

8: If you feel the urge to really encode everything with sets, you can still define $\iota_1(x) = (0, x)$ and $\iota_2(y) = (1, y)$ but then forget the definition.

Definition by cases is a primitive concept involving disjoint sums which deserves its own notation, preferably one that fits on a single line. We may mimic Haskell and write

$$\texttt{case}\; e \;\texttt{of}\; \iota_1(x) \mapsto e_1 \mid \iota_2(y) \mapsto e_2.$$

Read this as "if e is of the form $\iota_1(x)$ then e_1, else if e is of the form $\iota_2(y)$ then e_2". The variables x and y are bound in e_1 and e_2, respectively. The definition of f above would be written as

$$f\,u = \texttt{case}\; u \;\texttt{of}\; \iota_1(x) \mapsto f_1(x) \mid \iota_2(y) \mapsto f_2(y),$$

or spanning several lines

$$\begin{aligned} f\,u = \texttt{case}\; u \;&\texttt{of} \\ &\iota_1(x) \mapsto f_1(x) \\ &\iota_2(y) \mapsto f_2(y). \end{aligned}$$

We shall use this notation. Let us mention that in Haskell ι_1 and ι_2 are called `Left` and `Right`, respectively.

In a general category a ***(binary) coproduct*** of objects S and T is an object C with morphisms $\iota_1 : S \to C$ and $\iota_2 : T \to C$ such that, for all morphisms $f : S \to U$ and $g : T \to U$ there exists a unique $h : C \to U$ such that the

following diagram commutes:

$$\begin{array}{ccccc} & & U & & \\ & \overset{f}{\nearrow} & \uparrow h & \overset{g}{\nwarrow} & \\ S & \xrightarrow[\iota_1]{} & C & \xleftarrow[\iota_2]{} & T \end{array}$$

Notice that we have exactly reversed all the morphisms with respect to the definition of products. We write the coproduct of S and T as $S + T$ when it is given as an operation, and the unique morphism h as $[f, g]$.

Whether assemblies $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ have binary coproducts is an interesting question. The answer seems to depend on the structure of the underlying tpcas.

Definition 3.6.1 A tpca $\mathbb{A}$ with ***sums*** is a tpca with a binary operation $+$ on the types such that, for all types s, t, and u there exist constants

$$\begin{aligned} \mathtt{left}_{s,t} &\in \mathbb{A}_{s \to (s+t)} \\ \mathtt{right}_{s,t} &\in \mathbb{A}_{t \to (s+t)} \\ \mathtt{case}_{s,t,u} &\in \mathbb{A}_{(s+t) \to (s \to u) \to (t \to u) \to u} \end{aligned}$$

satisfying, for all x, y, f, g of appropriate types,

$$\begin{aligned} \mathtt{left}_{s,t}\, x &\downarrow, \\ \mathtt{right}_{s,t}\, y &\downarrow, \\ \mathtt{case}_{s,t,u}\, (\mathtt{left}_{s,t}\, x)\, f\, g &\geq f\, x, \\ \mathtt{case}_{s,t,u}\, (\mathtt{right}_{s,t}\, y)\, f\, g &\geq g\, y. \end{aligned}$$

We say that the elements `left`, `right`, and `case` are *suitable* for sums when they satisfy these properties.

A ***sub-tpca with sums*** is a sub-tpca $\mathbb{A}'$ of $\mathbb{A}$ such that there exists `left`, `right`, `case` in $\mathbb{A}'$ suitable for sums in $\mathbb{A}$.

Proposition 3.6.2 *Suppose $\mathbb{A}$ is a tpca and $\mathbb{A}'$ its sub-tpca. The category* $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *has binary coproducts if, and only if, $\mathbb{A}$ is a tpca with sums and $\mathbb{A}'$ is its sub-tpca with sums.*

Proof. Suppose first that $\mathbb{A}$ has sums and that $\mathbb{A}'$ is a sub-tpca with sums. The coproduct of S and T is the assembly

$$S + T = (|S| + |T|, \|S\| + \|T\|, \Vdash_{S+T})$$

where $\Vdash_{S+T}$ is most easily defined in terms of the existence predicate E_{S+T}:

$$\begin{aligned} \mathsf{E}_{S+T}(u) = \mathtt{case}\ u\ \mathtt{of} & \\ & \iota_1(x) \mapsto \{\mathtt{left}\ \mathtt{x} \mid \mathtt{x} \Vdash_S x\} \\ & \iota_2(y) \mapsto \{\mathtt{right}\ \mathtt{y} \mid \mathtt{y} \Vdash_T y\}. \end{aligned}$$

That is, the realizers for $\iota_1(x)$ are of the form `left x` where $\mathtt{x} \Vdash_S x$, and the realizers for $\iota_2(y)$ are of the form `right y` where $\mathtt{y} \Vdash_T y$. The canonical

inclusions $\iota_1 : |S| \to |S| + |T|$ and $\iota_2 : |T| \to |S| + |T|$ are realized by $\mathtt{left}_{\|S\|,\|T\|}$ and $\mathtt{right}_{\|S\|,\|T\|}$, respectively. To see that $S + T$ has the required universal property, consider $f : S \to U$ and $g : T \to U$, realized by $\mathtt{f}$ and $\mathtt{g}$, respectively. The map $h = [f, g] : |S| + |T| \to |U|$, defined by

$$h(u) = \mathtt{case}\, u\, \mathtt{of}\, \iota_1(x) \mapsto f\,x \mid \iota_2(y) \mapsto g(y),$$

is realized by $\langle u^{\|S\|+\|T\|}\rangle\, \mathtt{case}\, u\, \mathtt{f}\, \mathtt{g}$. It is the unique morphism satisfying $h \circ \iota_1 = f$ and $h \circ \iota_2 = g$.

Conversely, suppose $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ has binary coproducts. For every type t, define the assembly

$$A_t = (\mathbb{A}_t, t, \Vdash_t)$$

with $r \Vdash_t q \Leftrightarrow r = q$. For types s and t let $s + t$ be the underlying type of the coproduct $A_s + A_t$,

$$s + t = \|A_s + A_t\|.$$

Let $\mathtt{left}_{s,t}$ and $\mathtt{right}_{s,t}$ be a realizers for the canonical inclusions $\iota_1 : A_s \to A_s + A_t$ and $\iota_2 : A_t \to A_s + A_t$, respectively.

Suppose s, t, and u are types. Define $a \in \mathbb{A}'_{s\to(s\to u)\to(t\to u)\to u}$ and $b \in \mathbb{A}'_{t\to(s\to u)\to(t\to u)\to u}$ by

$$a = \langle x^s\rangle\, \langle f^{s\to u}\rangle\, \langle g^{t\to u}\rangle\, f\, x \qquad \text{and} \qquad b = \langle x^s\rangle\, \langle f^{s\to u}\rangle\, \langle g^{t\to u}\rangle\, g\, x.$$

The map $x \mapsto a\,x$ is a morphism from A_s to $A_{(s\to u)\to(t\to u)\to u}$ because it is realized by a. Similarly, the map $y \mapsto b\,y$ is a morphism from A_t to $A_{(s\to u)\to(t\to u)\to u}$, realized by b. There is a unique morphism $h : A_s + A_t \to A_{(s\to u)\to(t\to u)\to u}$ such that $h(\iota(x)) = a\,x$ and $h(\iota(y)) = b\,y$ for all $x \in A_s$ and $y \in A_t$. There exists

$$\mathtt{case}_{s,t,u} \in \mathbb{A}'_{(s+t)\to(s\to u)\to(t\to u)\to u}$$

which realizes h. We claim that $\mathtt{left}_{s,t}$, $\mathtt{right}_{s,t}$, and $\mathtt{case}_{s,t,u}$ have the desired properties. It is obvious that $\mathtt{left}_{s,t}\, x{\downarrow}$ and $\mathtt{right}_{s,t}\, y{\downarrow}$ for all $x \in \mathbb{A}_s$, $y \in \mathbb{A}_t$. Next, because $\mathtt{case}_{s,t,u}$ realizes h, $\mathtt{left}_{s,t}$ realizes ι_1, and $h(\iota_1(x)) = a\,x$, we have $\mathtt{case}_{s,t,u}\,(\mathtt{left}_{s,t}\, x) = a\,x$, therefore

$$\mathtt{case}_{s,t,u}\,(\mathtt{left}_{s,t}\, x)\, f\, g \simeq a\,x\,f\,g \simeq f\,x$$

for all x, f, and g of relevant types. Similarly, $\mathtt{case}_{s,t,u}\,(\mathtt{right}_{s,t}\, y)\, f\, g \simeq g\,y$ holds as well. □

The obvious question to ask is when a tpca has sums. We do not know whether there is a tpca without sums, and we do not explore the question further. We satisfying ourselves with a sufficient condition that covers the instances we care about.

Definition 3.6.2 A tpca $\mathbb{A}$ has ***booleans*** when there is a type $\mathtt{bool}$, and for each type t elements

$$\mathtt{false}, \mathtt{true} \in \mathbb{A}_{\mathtt{bool}} \qquad \text{and} \qquad \mathtt{if}_t \in \mathbb{A}_{\mathtt{bool}\to t\to t\to t}$$

satisfying, for all $x, y \in \mathbb{A}_t$,

$$\mathtt{if}_t\,\mathtt{true}\,x\,y = x \qquad \text{and} \qquad \mathtt{if}_t\,\mathtt{false}\,x\,y = y.$$

We say that $\mathtt{false}$, $\mathtt{true}$, $\mathtt{if}_t$ are *suitable* for booleans in $\mathbb{A}$.

A sub-tpca with booleans is a sub-tpca $\mathbb{A}'$ of $\mathbb{A}$ such that there exists $\mathtt{false}$, $\mathtt{true}$, $\mathtt{if}_t$ in $\mathbb{A}'$ which are suitable for booleans in $\mathbb{A}$.

Proposition 3.6.3 *A tpca $\mathbb{A}$ has sums if, and only if, it has booleans. Furthermore, a sub-tpca $\mathbb{A}'$ is a sub-tpca with sums if, and only if, it is a sub-tpca with booleans.*

Proof. Suppose $\mathbb{A}$ has sums. Pick any type o, an element $\omega_o \in \mathbb{A}_o$, and define

$$\begin{aligned}
\mathtt{bool} &= o + o, \\
\mathtt{true} &= \mathtt{left}\,\omega_o, \\
\mathtt{false} &= \mathtt{right}\,\omega_o \\
\mathtt{if}_t &= \langle b^{\mathtt{bool}}\rangle\,\langle x^t\rangle\,\langle y^t\rangle\,\mathtt{case}_{o,t,t}\,b\,(\mathtt{K}_{o,t}\,x)\,(\mathtt{K}_{o,t}\,y)
\end{aligned}$$

It is easy to check that these satisfy the conditions from Definition 3.6.2.

Conversely, suppose $\mathbb{A}$ has booleans, and let s, t, and u be types. There exist $\omega_s \in \mathbb{A}_s$ and $\omega_t \in \mathbb{A}_t$. Define

$$\begin{aligned}
s + t &= \mathtt{bool} \times (s \times t) \\
\mathtt{left}_{s,t} &= \langle x^s\rangle\,\mathtt{pair}\,\mathtt{true}\,(\mathtt{pair}\,x\,\omega_t) \\
\mathtt{right}_{s,t} &= \langle y^t\rangle\,\mathtt{pair}\,\mathtt{false}\,(\mathtt{pair}\,\omega_s\,y) \\
\mathtt{case}_{s,t,u} &= \langle z^{s+t}\rangle\,\langle f^{s\to u}\rangle\,\langle g^{s\to u}\rangle \\
&\qquad \mathtt{if}\,(\mathtt{fst}\,z)\,(f\,(\mathtt{fst}\,(\mathtt{fst}\,z)))\,(g\,(\mathtt{snd}\,(\mathtt{fst}\,z)))
\end{aligned}$$

These have the required properties, as is easily checked. □

Proposition 3.6.4 *Every N-tpca has booleans and every sub-N-tpca is a sub-N-tpca with booleans.*

Proof. Let $\mathbb{A}$ be a N-tpca and $\mathbb{A}'$ its sub-N-tpca. Define

$$\begin{aligned}
\mathtt{bool} &= \mathtt{nat}, \\
\mathtt{false} &= \overline{0}, \\
\mathtt{true} &= \overline{1}, \\
\mathtt{if}_t &= \langle b^{\mathtt{bool}}\rangle\,\langle x^t\rangle\,\langle y^t\rangle\,\mathtt{rec}_t\,y\,(\langle n^{\mathtt{nat}}\rangle\,\langle z^t\rangle\,x)\,b
\end{aligned}$$

Again, it is easy to check that these have the desired properties. □

Finally, let us put all these together.

Proposition 3.6.5 *If $\mathbb{A}$ is a N-tpca and $\mathbb{A}'$ its sub-N-tpca then* $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *has binary coproducts.*

Proof. Combine Propositions 3.6.2 to 3.6.4. □

The other finite colimits are more easily dealt with. The initial object is the ***empty assembly***

$$\mathsf{0} = (\emptyset, o, \Vdash_{\mathsf{0}})$$

where o is any type.[9] Its universal property is that there is exactly one morphism $\mathsf{0} \to S$ for every assembly S. The property holds because there is a unique map $\emptyset \to S$, which is realized by $\mathsf{K}_{\|S\|,o}\, a$, where $a \in \mathbb{A}'_o$.

9: We need not specify $\Vdash_{\mathsf{0}}$ because there is only one relation between $\mathbb{A}_o$ and $\emptyset$.

A ***coequalizer*** of morphisms $f, g : S \to T$ is an object Q with a morphism $q : T \to Q$ that equalizes f and g, which means $q \circ f = q \circ g$, and has the following universal property: if $k : T \to K$ equalizes f and g then there is a unique morphism $i : Q \to K$ such that $k = i \circ q$:

$$\begin{array}{ccccc} S & \overset{f}{\underset{g}{\rightrightarrows}} & T & \xrightarrow{q} & Q \\ & & & \searrow^{k} & \downarrow{\scriptstyle i} \\ & & & & K \end{array}$$

In the category of sets the coequalizer is the quotient $Q = T/{\equiv}$ of T by the least equivalence relation $\equiv$ satisfying $f\,x \equiv g\,x$ for all $x \in S$. The map $q : T \to T/{\equiv}$ is the canonical quotient map which takes $y \in T$ to its equivalence class $[y]_{\equiv}$.

In $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ the coequalizer of $f, g : S \to T$ is the assembly $T/{\equiv}$ where $T/{\equiv}$ is the coequalizer of f and g computed in sets, as described above, $\|T/{\equiv}\| = \|T\|$, and $\Vdash_{T/\equiv}$ is defined by

$$\mathsf{y} \Vdash_X [z]_{\equiv} \iff \exists y \in T.\, \mathsf{y} \Vdash_T y \land y \equiv z.$$

The canonical quotient map $q : T \to T/{\equiv}$ is realized by $\langle \mathsf{y}^{\|T\|} \rangle\, \mathsf{y}$.

Pushouts, which are dual to pullbacks, exist in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ if coproducts do, because every finite colimit is a coequalizer of a finite coproduct, as recorded in the following proposition.

Exercise 3.6.3 Give an explicit description of the pushout of assemblies.

Proposition 3.6.6 $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *is cocartesian*[10] *if, and only if,* $\mathbb{A}$ *is a tpca with sums and* $\mathbb{A}'$ *a sub-tpca with sums.*

10: A category is ***cocartesian*** or ***finitely cocomplete*** if it has finite colimits.

Proof. Coequalizers and the initial object always exist. Therefore, all finite colimits exist, provided binary coproducts do. By Proposition 3.6.2 this is equivalent to the condition that $\mathbb{A}$ have sums and $\mathbb{A}'$ be a sub-tpca with sums. □

3.6.3 Monos and epis

Recall that f is a ***monomorphism (mono)*** when it can be canceled on the left: if $f \circ g = f \circ h$ then $g = h$. The dual notion is ***epimorphism (epi)***, which is a morphism that can be canceled on the right. In the category of

sets the monos and epis are precisely the injective and surjective maps, respectively.

Proposition 3.6.7 *A morphism in* $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *is mono if, and only if, it is mono as a map in* Set*, and likewise for epis.*

Proof. It is obvious that a morphism $f : S \to T$ is a mono in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ if it is mono in Set. Conversely, suppose f is mono in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$, and consider maps $g, h : U \to T$ in Set such that $f \circ g = f \circ h$. Define the assembly $U = (U, \|S\| \times \|S\|, \Vdash_U)$ with the realizability relation

$$p \Vdash_U u \iff \mathtt{fst}\, p \Vdash_S f\, u \wedge \mathtt{snd}\, p \Vdash_S g\, u.$$

The maps g and h are morphisms from U to S because they are realized by $\mathtt{fst}$ and $\mathtt{snd}$, respectively. Since f is mono as a morphism of assemblies, it follows that $g = h$.

Next we consider epis. Again, it is easy to see that a morphism $f : S \to T$ is epi if it is epi in Set. Conversely, suppose f is epi in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ and consider maps $g, h : T \to U$ in Set such that $g \circ f = h \circ f$. The maps g and h are morphisms $T \to \nabla U$ because they are both trivially realized. Since f is epi in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$, we may cancel it and obtain $g = h$. This shows that f is epi in Set. □

A ***mono-epi*** is a morphism $f : S \to T$ which is both mono and epi. In general such a morphism need not be an isomorphism. For example, a continuous bijection between topological spaces need not be a homeomorphism.

Corollary 3.6.8 *An assembly map* $f : S \to T$ *is mono-epi if, and only if, its underlying map is a bijection.*

Proof. This follows directly from Proposition 3.6.7 and the fact that in Set an epi-mono is the same thing as a bijection. □

It is easy to provide an epi-mono which is not an isomorphism. For example if S is a modest set with at least two different elements, then $\mathrm{id}_S : S \to S$ is realized as a morphism $\mathrm{id}_S : S \to \nabla S$, hence it is an epi-mono in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$. However, every morphism $\nabla S \to S$ is a constant map, therefore S is not isomorphic to ∇S.

Recall that $f : S \to T$ is a ***regular mono*** if there are $g, h : T \to U$ such that f is their equalizer. Regular monos are well behaved, and we can think of them as subspace embeddings. Given an assembly T and a subset $T' \subseteq T$, define the assembly $T' = (T', \|T\|, \Vdash_{T'})$ as the restriction of T, i.e., $\mathtt{x} \Vdash_{T'} x$ if, and only if $x \in T'$ and $\mathtt{x} \Vdash_T x$. The subset inclusion $\iota : T' \to T$ is a morphism of assemblies because it is realized by $\langle x^{\|T\|} \rangle\, x$, and it is a regular mono because it is the equalizer of the maps $g, h : T \to \nabla 2$, defined by

$$g\, x = 1 \qquad \text{and} \qquad h(x) = \begin{cases} 1 & \text{if } x \in T', \\ 0 & \text{otherwise.} \end{cases}$$

Even more, every regular mono is of this form, up to isomorphism. To see this, suppose $f : S \longrightarrow T$ is a regular mono, thus an equalizer of morphisms $g, h : T \longrightarrow U$. If we simply compute the equalizer of g and h again according to the recipe (3.1) from Subsection 3.6.1, we see that it is precisely the restriction of T to the subset

$$T' = \{x \in T \mid g\,x = h(x)\}.$$

Since both $\iota : T' \longrightarrow T$ and $f : S \longrightarrow T$ are equalizers of g and h, they are isomorphic.

The following characterization of regular monos is often useful.

Proposition 3.6.9 *A realized map $f : S \longrightarrow T$ is a regular mono if, and only if, f is injective and there exists $\mathtt{i} \in \mathbb{A}'_{\|T\| \to \|S\|}$ such that, for all $x \in S$ and $\mathtt{y} \in \mathbb{A}_{\|T\|}$, if $\mathtt{y} \Vdash_T f\,x$ then $\mathtt{i}\,\mathtt{y}$ is defined and $\mathtt{i}\,\mathtt{y} \Vdash_S x$.*

Proof. A regular mono $f : S \longrightarrow T$ is injective because it is mono. There exist realized maps $g, h : T \longrightarrow U$ such that f is their equalizer. Let $e : E \longrightarrow T$ be the equalizer of g and h, as computed in (3.1):

$$\begin{aligned} E &= (\{y \in T \mid g(y) = h(y)\}, \|T\|, \Vdash_E), \\ \mathtt{y} \Vdash_E y &\iff \mathtt{y} \Vdash_T y, \\ e(y) &= y. \end{aligned}$$

Because e equalizes g and h, there is a realized map $i : E \longrightarrow S$ such that $e = f \circ i$. We claim that any realizer $\mathtt{i} \in \mathbb{A}'_{\|T\| \to \|S\|}$ of i has the desired property. Suppose $x \in S$, $\mathtt{y} \in \mathbb{A}_{\|T\|}$, and $\mathtt{y} \Vdash_T f\,x$. Because $g(f\,x) = h(f\,x)$, $f\,x \in E$ and so $\mathtt{y} \Vdash_E f\,x$. Then $\mathtt{i}\,\mathtt{y}$ is defined and $\mathtt{i}\,\mathtt{y} \Vdash_S i(f\,x)$. This is what we want because $f(i(f\,x)) = e(f\,x) = f\,x$, from which $i(f\,x) = x$ follows by injectivity of f.

Conversely, suppose $f : S \longrightarrow T$ is injective, realized by $\mathtt{f}$, and $\mathtt{i}$ is as in statement of the proposition. Let $T' = \{y \in T \mid \exists x.\, S f\,x = y\}$. It suffices to show that f is isomorphic to the inclusion $\iota : T' \longrightarrow T$, where T' is the restriction of T to T'. The map $j : S \longrightarrow T'$ defined by $j(x) = f\,x$ is realized by $\mathtt{f}$. The map $i : T' \longrightarrow S$, defined by $i(f\,x) = x$, is well defined because f is injective, and is realized by $\mathtt{i}$. Clearly, j and i are inverses of each other and $f = \iota \circ j$. □

We repeat the story for ***regular epis***, which are those morphisms that are coequalizers. They are the well behaved epis which can be thought of as quotient maps. In fact, if $f : S \longrightarrow T$ is a regular epi, we say that T is a ***quotient*** of S.

The match between regular epis and quotients is precise in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$. Note that that in the construction of coequalizers, as described above, we may start with an arbitrary equivalence relation: given an assembly T and an equivalence relation $\equiv$ on T, define the ***quotient assembly*** $T/{\equiv} = (T/{\equiv}, \|T\|, \Vdash_{T/\equiv})$ whose realizability relation satisfies $\mathtt{x} \Vdash_{T/\equiv} [y]$ if, and only if, $\mathtt{x} \Vdash_T x$ and $x \equiv y$ for some $x \in T$. The quotient map $q : T \longrightarrow T/{\equiv}$ is realized by $\langle x^{\|T\|} \rangle\, x$, and is a coequalizer of $g, h : V \longrightarrow T$ where[11]

11: You should convince yourself that V is the ***kernel pair*** of q, i.e., the pullback of q with itself.

$$
\begin{aligned}
V &= (\{(x, y) \in T \times T \mid x \equiv y\}, \|T\| \times \|T\|, \Vdash_V), \\
p \Vdash_V (x, y) &\iff \texttt{fst}\, p \Vdash_T x \wedge \texttt{snd}\, p \Vdash_T y, \\
g(x, y) &= x, \\
h(x, y) &= y.
\end{aligned}
$$

Every regular epi is isomorphic to one of this form. To see this, suppose $f : T \longrightarrow U$ is a coequalizer of $g, h : U \longrightarrow T$. Let $\equiv$ be the least equivalence relation on T such that $g\, x = h(x)$ for all $x \in U$. Then f is isomorphic to $q : T \longrightarrow T/{\equiv}$ because $g : T \longrightarrow T/{\equiv}$ is the coequalizer of g and h according to Subsection 3.6.2.

There is another characterization of regular epis which is used often.

Proposition 3.6.10 *in* $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *a morphism* $f : T \longrightarrow U$ *is a regular epi if, and only if, there exists* $\texttt{i} \in \mathbb{A}'_{\|U\| \to \|T\|}$ *such that, whenever* $\texttt{y} \Vdash_U y$ *then* $\texttt{i}\,\texttt{y}\downarrow$ *and there is* $x \in f^*(y)$ *such that* $\texttt{i}\,\texttt{y} \Vdash_T x$*.*

Proof. Suppose first that we have a regular epi $f : T \longrightarrow U$ which is a coequalizer of $g, h : S \longrightarrow T$. Let $q : T \longrightarrow T/{\equiv}$ be the coequalizer of g and h, as computed in Subsection 3.6.2. Because f and q both equalize g and h, there exists a unique isomorphism $i : U \longrightarrow T/{\equiv}$ such that $q = i \circ f$. Let $\texttt{i} \in \mathbb{A}'_{\|U\| \to \|T\|}$ be a realizer for i. We claim that it has the required properties. If $\texttt{y} \Vdash_U y$ then $\texttt{i}\,\texttt{y}\downarrow$ and $\texttt{i}\,\texttt{y} \Vdash_{T/\equiv} i(y)$. Because f is surjective there exists $x' \in T$ such that $f(x') = y$, from which we get $i(y) = i(f(x')) = q(x') = [x']$. Since $\texttt{i}\,\texttt{y} \Vdash_{T/\equiv} [x']$ there is $x \in T$ such that $\texttt{i}\,\texttt{y} \Vdash_T x$ and $x \equiv x'$. The element x is the one we are looking for because $x \equiv x'$ implies $f\, x = f(x') = y$.

Conversely, suppose $\texttt{i}$ is as in the statement of the proposition, and let $\texttt{f}$ be a realizer for f. To show that $f : T \longrightarrow U$ is a coequalizer, define the equivalence relation $\equiv$ on T by

$$x \equiv y \iff f\, x = f(y).$$

It suffices to show that f is isomorphic to $q : T \longrightarrow T/{\equiv}$. In one direction we have the map $j : T/{\equiv} \longrightarrow U$ defined by $j([x]) = f\, x$, realized by $\texttt{f}$. In the other direction we have $i : U \longrightarrow T/{\equiv}$ defined by $i(f\, x) = [x]$, which is well defined because f is surjective. The map i is realized by $\texttt{i}$. It is obvious that i and j are inverses of each other and that $q = i \circ f$ holds. □

3.6.4 Regular structure

In the category of sets every function $f : S \longrightarrow T$ may be factored as $f = e \circ m$ where e is epi and m is mono. Similarly, a continuous map between topological spaces may be factored into a regular epi and a mono.

In assemblies a realized map $f : S \longrightarrow T$ factors as

$$\begin{array}{ccc} S & \xrightarrow{\quad f \quad} & T \\ {\scriptstyle q}\searrow & & \nearrow{\scriptstyle i} \\ U & \xrightarrow[b]{} & V \end{array}$$

where q is a regular epi, b is a mono-epi and i is a regular mono. Indeed, we may take $U = S/{\equiv}$, where $x \equiv y \iff f\,x = f(y)$, and $q : S \longrightarrow S/{\equiv}$ the canonical quotient map. The assembly V is the restriction of T to the subset $V = \{y \in T \mid \exists x.\, S f\,x = y\}$ so that $\|V\| = \|T\|$ and $\mathsf{y} \Vdash_V y \Leftrightarrow \mathsf{y} \Vdash_T y$ for all $y \in V$. The map $i : V \longrightarrow T$ is the subset inclusion. Finally, $b : U \longrightarrow V$ is characterized by $b([x]_{\equiv}) = f\,x$. It is realized by the same realizers as f.

In the above factorization we call U the ***image*** of f and V the ***stable image*** of f. The reason for the terminology is revealed in **??**, where the stable image is related to stable propositions.

The factorization is unique up to isomorphism. Suppose we had another factorization $f = i' \circ b' \circ q'$ where i', b' and q' are regular mono, mono-epi, and regular-epi, respectively. We claim that there are unique isomorphisms j and k which make the following diagram commute:

$$\begin{array}{ccccc} S & & \xrightarrow{\qquad f \qquad} & & T \\ & {\scriptstyle q}\searrow & & \nearrow{\scriptstyle i} & \\ {\scriptstyle q'} & U & \xrightarrow{b} & V & {\scriptstyle i'} \\ & {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle k} & \\ & U' & \xrightarrow[b']{} & V' & \end{array}$$

Without loss of generality we may assume that q' is a canonical quotient map $q' : S \longrightarrow S/{\equiv'}$ for an equivalence relation $\equiv'$ on S, and that $i' : V' \longrightarrow T$ is a subset inclusion.

First we show that $\equiv$ and $\equiv'$ coincide. If $x \equiv y$ then $i'(b'(q'(x))) = f\,x = f(y) = i'(b'(q'(y)))$, and since $i' \circ b'$ is a mono $q'(x) = q'(y)$ and $x \equiv' y$ follows. Conversely, if $x \equiv' y$ then $q'(x) = q'(y)$ and $f\,x = i'(b'(q'(x))) = i'(b'(q'(y))) = f(y)$, hence $x \equiv y$.

Next we verify that V and V' are equal. If $y \in V$ then $f\,x = y$ for some $x \in S$. Because $y = f\,x = i'(b'(q'(x)))$ and i' is a subset inclusion, $b'(q'(x)) = f\,x = y$, which shows that $y \in V'$. Conversely, if $y \in V'$ then there is $x \in S$ such that $y = i'(b'(q'(x))) = f\,x$, but then $y \in V$.

Good factorization properties of $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ lead to another important feature of assemblies.

Proposition 3.6.11 *The category* $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *is* **regular** *which means that*

1. *it is cartesian,*
2. *every morphism can be factored as a composition of a regular epi and a mono, and*
3. *the pullback of a regular epi is a regular epi.*

Proof. The first item was proved in Proposition 3.6.1. For the second item, take the above factorization $f = i \circ b \circ q$ and notice that q is a regular epi and $i \circ b$ a mono. Lastly, suppose $q : T \to T/{\equiv}$ is a regular epi, where we assumed without loss of generality that it is a quotient by an equivalence relation. Let $f : S \to T/{\equiv}$ be realized by $\mathtt{f} \in \mathbb{A}'_{\|S\| \to \|T\|}$. The pullback of q is the map $r : P \to S$, as in the diagram

$$\begin{array}{ccc} P & \longrightarrow & T \\ {\scriptstyle r}\downarrow & \lrcorner & \downarrow{\scriptstyle q} \\ S & \xrightarrow[f]{} & T/{\equiv} \end{array}$$

where

$$P = (\{(x, y) \in S \times T \mid f\,x \equiv y\}, \|S\| \times \|T\|, \Vdash_P),$$

$\mathtt{pair}\,\mathtt{x}\,\mathtt{y} \Vdash_P (x, y)$ if, and only if, $\mathtt{x} \Vdash_S x$ and $\mathtt{y} \Vdash_T y$ and $r : (x, y) \mapsto x$. Let us use Proposition 3.6.10 to show that r is regular epi. The realizer $\mathtt{i} = \langle x^{\|S\|} \rangle\, \mathtt{pair}\, x\, (\mathtt{f}\, x)$ satisfies the conditions of the proposition: if $x \in S$ and $\mathtt{x} \Vdash_S x$ then $\mathtt{i}\,\mathtt{x} = \mathtt{pair}\,\mathtt{x}\,(\mathtt{f}\,\mathtt{x})$ is defined. There is $y \in T$ such that $f\,x = [y]_{\equiv}$, hence $r(x, y) = x$, and also $\mathtt{i}\,\mathtt{x} \Vdash_P (x, y)$. □

The regular structure of assemblies is important for at least two reasons: it gives a well-defined notion of an image of a realized map, and it provides an interpretation of existential quantifiers, cf. **??**.

3.6.5 Cartesian closed structure

If S and T are objects in a category, we may form the *set* $\mathsf{Hom}(S, T)$ of morphisms with domain S and codomain T. Sometimes $\mathsf{Hom}(S, T)$ carries additional structure that turns it into an object of the category. For example, in the category of partially ordered sets and monotone maps, the set $\mathsf{Hom}(P, Q)$ of monotone maps between $(P, \leq_P)$ and $(Q, \leq_Q)$ is partially ordered by $f \leq g \Leftrightarrow \forall x. P\, f\, x \leq_Q g\, x$. The following definition explains what it means for an object to correspond to the the set of morphisms.

Definition 3.6.3 An ***exponential*** of objects S and T is an object E with a morphism $e : E \times S \to T$, such that for every $f : U \times S \to T$ there exists a *unique* $\hat{f} : U \to E$ such that the following diagram commutes:

$$\begin{array}{ccc} E \times S & & \\ {\scriptstyle \hat{f} \times \mathrm{id}_S}\uparrow & \searrow^{e} & \\ U \times S & \xrightarrow[f]{} & T \end{array}$$

A category with finite products in which all exponentials exist is a ***cartesian closed category (ccc)***.

The exponential is determined uniquely up to isomorphism. When given as an operation, we denote the exponential of S and T by T^S, and sometimes by $S \to T$. The map $e : T^S \times S \to T$ is called the

evaluation morphism. The map $\hat{f} : U \to T^S$ is called the ***transpose*** of $f : U \times S \to T$.

Let us explain how exponentials work in the category of sets. The exponential of sets S and T is the set

$$T^S = \{f \mid f \text{ is a function from } S \text{ to } T\}$$

and the evaluation map is $e(f, x) = f\,x$. The transpose of $f : U \times S \to T$ is $\hat{f}(z)(x) = f(z, x)$. This way the diagram commutes because $e((\hat{f} \times \mathrm{id}_S)(z, x)) = e(\hat{f}(z), x) = \hat{f}(z)(x) = f(z, x)$. The transpose $\hat{f}$ is the only map satisfying this property.

Proposition 3.6.12 *The categories* $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ *and* $\mathsf{Mod}(\mathbb{A}, \mathbb{A}')$ *are cartesian closed.*

Proof. We prove that $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ has exponentials. The same construction works for modest sets. Suppose S and T are assemblies. Define the assembly

$$T^S = (\{f : S \to T \mid f \text{ is realized}\}, \|S\| \to \|T\|, \Vdash_{S \to T})$$

where

$$\mathtt{f} \Vdash_{S\to T} f \iff \forall x.\, S \forall \mathtt{x} \in \|S\|.\, (\mathtt{x} \Vdash_S x \Rightarrow \mathtt{f}\,\mathtt{x}{\downarrow} \land \mathtt{f}\,\mathtt{x} \Vdash_T f\,x).$$

None of this is surprising because we just copied the definition of realized maps. The evaluation map $e : T^S \times S \to T$ is $e(f, x) = f\,x$, which is realized by

$$\mathtt{e} = \langle p^{(\|S\| \to \|T\|) \times \|S\| \to \|T\|} \rangle\,(\mathtt{fst}\,p)\,(\mathtt{snd}\,p).$$

The transpose of $f : U \times S \to T$, realized by $\mathtt{f}$, is the map $\hat{f}(z)(x) = f(z, x)$, which is realized by

$$\hat{\mathtt{f}} = \langle z^{\|U\|} \rangle\, \langle x^{\|S\|} \rangle\, \mathtt{f}\,(\mathtt{pair}\,z\,x). \qquad \square$$

The passage from $f : U \times S \to T$ to its transpose $\hat{f} : U \to T^S$ has an inverse. To every $g : U \to T^S$ we may assign $\check{g} : U \times S \to T$, defined by $\check{g}(z, x) = g(z)(x)$. If g is realized by $\mathtt{g}$ then $\check{g}$ is realized by $\check{\mathtt{g}} = \langle p^{\|U\| \times \|S\|} \rangle\, \mathtt{g}\,(\mathtt{fst}\,p)\,(\mathtt{snd}\,p)$. It is easy to check that $\check{\hat{f}} = f$ and $\hat{\check{g}} = g$. The operation $f \mapsto \hat{f}$ is also known as ***currying*** and its inverse $g \mapsto \check{g}$ as ***uncurrying***.

3.6.6 The interpretation of λ-calculus in assemblies

Currying and uncurrying are useful operations, but the above notation with “hats and checks” is not very practical. We may take better advantage of the cartesian closed structure of $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ by interpreting the λ-calculus in it. The types of the λ-calculus are the assemblies, where the product and function types are interpreted as products and exponentials of assemblies, respectively. The unit type is the terminal assembly $\mathsf{1}$. The expressions are those of the λ-calculus, except that we write the

projections as π_1 and π_2 instead of `fst` and `snd`, respectively. In addition if T is an assembly and $a \in T$ is an element for which there exits a realizer $\mathtt{a} \in \mathbb{A}'_{\|T\|}$ then a is a primitive constant of type T.

Suppose e is an expression of type T and the freely occurring variables of e are among $x_1^{S_1}, \ldots, x_n^{S_n}$. We prefer to write the list of variables as $x_1 : S_1, \ldots, x_n : S_n$, which we abbreviate as $\overline{x} : \overline{S}$, and call it a ***typing context*** for e. The expression with the typing context determines a realized map

$$[\![\overline{x} : \overline{S} \mid e : T]\!] : S_1 \times \cdots \times S_n \to T,$$

which we abbreviate to $[\![e]\!]$ when no confusion may arise. We define the meaning of $[\![e]\!]$ inductively on the structure of e as follows, where $a = (a_1, \ldots, a_n) \in S_1 \times \cdots \times S_n$:

1. A primitive constant $b \in T$ which is realized by $\mathtt{b} \in \mathbb{A}'_{\|T\|}$ is interpreted as the constant map

$$[\![\overline{x} : \overline{S} \mid b : T]\!](a) = b,$$

which is realized[12] by $\langle x^{\|S_1\| \times \cdots \times \|S_n\|} \rangle\, \mathtt{b}$.

2. A variable $x_i^{S_i}$ is interpreted as the i-th projection

$$[\![\overline{x} : \overline{S} \mid x_i : S_i]\!](a) = a_i.$$

which of course is realized.

3. A λ-abstraction $\lambda y{:}U.\, e$ of type $U \to T$ is interpreted as the realized map $[\![\lambda y{:}U.\, e]\!] : S_1 \times \cdots \times S_n \to T^U$ that is obtained as the transpose of

$$S_1 \times \cdots \times S_n \times U \xrightarrow{[\![\overline{x}:\overline{S}, y:U \mid e:T]\!]} T$$

4. The interpretation of an application $e_1\, e_2$, where e_1 has type $U \to T$ and e_2 has type U, is the map

$$S_1 \times \cdots \times S_n \xrightarrow{\langle [\![e_1]\!], [\![e_2]\!] \rangle} T^U \times U \xrightarrow{\mathsf{ev}} T$$

where ev is the evaluation map.

5. The interpretation of a pair (e_1, e_2) of type $T \times U$ is the map

$$S_1 \times \cdots \times S_n \xrightarrow{\langle [\![e_1]\!], [\![e_2]\!] \rangle} T \times U$$

6. A projection $\pi_1(e)$ of type T, where e has type $T \times U$, is the map

$$S_1 \times \cdots \times S_n \xrightarrow{[\![e]\!]} T \times U \xrightarrow{\pi_1} T$$

The second projection π_2 is treated analogously.

12: Had we allowed as primitive constants *all* elements of T, we would face a difficulty here, because we could not exhibit a computable realizer for the constant map.

This definition shows that we may freely use the λ-calculus to define realized maps. Although the definition tells us exactly how to compute the realizers from the expressions, the idea is to *not* do that. We have verified once and for all that any map defined by λ-calculus is realized, and in most cases we do not care which specific realizer is used.

When e is a closed expression of type S and the typing context $\overline{x} : \overline{S}$ is an

empty list, the meaning of e is a realized map

$$[\![\cdot \mid e : 1 \to T]\!]$$

which amounts to the same thing as an element of T. This element has a computable realizer, i.e., one in $\mathbb{A}'_{\|T\|}$, because the corresponding realized map does.

We may further simplify the notation by allowing *patterns* in λ-abstraction, a technique commonly used in functional programming languages. Instead of

$$\lambda p{:}S \times T. \; \ldots$$

we write

$$\lambda(x, y){:}S \times T. \; \ldots$$

and replace each occurrence of $\mathtt{fst}\, p$ by x, of $\mathtt{snd}\, p$ by y, and all other occurrences of p by (x, y). For instance, we would write

$$\lambda(f, g){:}(T \to U) \times (S \to T).\, \lambda x{:}S.\, f\,(g\,x)$$

instead of

$$\lambda p{:}(T \to U) \times (S \to T).\, \lambda x{:}S.\, (\mathtt{fst}\, p)\,(\mathtt{snd}\, p)\, x.$$

It is also useful to write the definition of a function as

$$f\, x_1 \, \ldots \, x_n = e$$

instead of

$$f = \lambda x_1 \, \ldots \, x_n .\, e.$$

When $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ has coproducts, and in most cases of interest it does, the λ-calculus may be extended further to encompass binary sums. If S and T are assemblies, viewed as types, then we have the following expressions:

1. If e is an expressions of type S then $\iota_1^{S,T}(e)$ is an expression of type $S + T$. It is interpreted as the composition

$$S_1 \times \cdots \times S_n \xrightarrow{[\![e]\!]} S \xrightarrow{\iota_1} S + T$$

where ι_1 is the canonical inclusion. The expression $\iota_2^{S,T}(e)$ is treated similarly. We usually omit supscripts S, T on ι_1 and ι_2.

2. If e_1 has type $S + T$, and e_2 and e_3 have type U, then

$$\mathtt{case}\, e_1\, \mathtt{of}\, \iota_1(x) \mapsto e_2 \mid \iota_2(y) \mapsto e_3$$

is an expression of type U, with x bound in e_2 and y bound in e_3. It is interpreted as the composition

$$S_1 \times \cdots \times S_n \xrightarrow{[\![e_1]\!]} S + T \xrightarrow{[[\![e_2]\!],[\![e_3]\!]]} U$$

The following equations hold:

$$
\begin{aligned}
(\mathtt{case}\, \iota_1(e)\, \mathtt{of}\, \iota_1(x) \mapsto e_1 \mid \iota_2(y) \mapsto e_2) &= e_1[e/x], \\
(\mathtt{case}\, \iota_2(e)\, \mathtt{of}\, \iota_1(x) \mapsto e_1 \mid \iota_2(y) \mapsto e_2) &= e_2[e/y], \\
(\mathtt{case}\, e_1\, \mathtt{of}\, \iota_1(x) \mapsto e_2[\iota_1(x)/z] \mid \iota_2(y) \mapsto e_2[\iota_2(y)/z]) &= e_2[e_1/z], \\
e[(\mathtt{case}\, e_1\, \mathtt{of}\, \iota_1(x) \mapsto e_2 \mid \iota_2(y) \mapsto e_3)/z] &= \\
&\mathtt{case}\, e_1\, \mathtt{of} \\
&\qquad \iota_1(x) \mapsto e[e_2/z] \\
&\qquad \iota_2(y) \mapsto e[e_3/z]
\end{aligned}
$$

We conclude by using the cartesian closed structure of $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ to derive the distributive law

$$(S + T) \times U \cong S \times U + T \times U.$$

Let us use the λ-calculus to write down the isomorphisms explicitly. The isomorphism from left to right is

$$
\begin{aligned}
f = \lambda(a, b){:}(S + T) \times U.\, &\mathtt{case}\, a\, \mathtt{of} \\
&\qquad \iota_1(x) \mapsto \iota_1(x, b) \\
&\qquad \iota_2(y) \mapsto \iota_2(y, b)
\end{aligned}
$$

and its inverse is

$$
\begin{aligned}
g = \lambda c{:}(S \times U) + (T \times U).\, &\mathtt{case}\, c\, \mathtt{of} \\
&\qquad \iota_1(x, b) \mapsto (\iota_1(x), b) \\
&\qquad \iota_2(y, b) \mapsto (\iota_2(y), b)
\end{aligned}
$$

We compute

$$
\begin{aligned}
g(f(a, b)) &= g(\mathtt{case}\, a\, \mathtt{of}\, \iota_1(x) \mapsto \iota_1(x, b) \mid \iota_2(y) \mapsto \iota_2(y, b)) \\
&= \mathtt{case}\, a\, \mathtt{of} \\
&\qquad \iota_1(x) \mapsto g(\iota_1(x, b)) \\
&\qquad \iota_2(y) \mapsto g(\iota_2(y, b)) \\
&= \mathtt{case}\, a\, \mathtt{of} \\
&\qquad \iota_1(x) \mapsto (\iota_1(x), b) \\
&\qquad \iota_2(y) \mapsto (\iota_2(y), b) \\
&= (a, b)
\end{aligned}
$$

and

$$
\begin{aligned}
f(g(c)) &= f(\mathtt{case}\, c\, \mathtt{of}\, \iota_1(x, b) \mapsto (\iota_1(x), b) \mid \iota_2(y, b) \mapsto (\iota_2(y), b)) \\
&= \mathtt{case}\, c\, \mathtt{of} \\
&\qquad \iota_1(x, b) \mapsto f(\iota_1(x), b) \\
&\qquad \iota_2(y, b) \mapsto f(\iota_2(y), b) \\
&= \mathtt{case}\, c\, \mathtt{of} \\
&\qquad \iota_1(x, b) \mapsto \iota_1(x, b) \\
&\qquad \iota_2(y, b) \mapsto \iota_2(y, b) \\
&= c.
\end{aligned}
$$

This proof works in any cartesian closed category with binary coproducts. In particular, it works in $\mathsf{Asm}(\mathbb{A}, \mathbb{A}')$. Notice how we need not worry about the underlying realizers for the isomorphisms f and g. You are invited to redo the proof by drawing the relevant commutative diagrams and using the universal properties of products, coproducts, and exponentials.

3.6.7 Projective assemblies

An object P is ***(regular) projective*** when for every regular epi $e : A \to B$ and $P : A \to B$ there is $\overline{f} : P \to A$ such that $f = e \circ \overline{f}$:

$$\begin{array}{ccc} & & A \\ & \overset{\overline{f}}{\nearrow} & \downarrow e \\ B & \xrightarrow[f]{} & P \end{array}$$

We say that P has *the lifting property* with respect to morphisms, because every f "lifts" to $\overline{f}$, as in the diagram.[13] Which assemblies are projective?

13: In a general category an object is called *projective* if it has the lifting property with respect to *all* epis, and *regular projective* if it has the liftiing property with respect to *regular* epis. However, we are only interested in the regular projective objects, so we drop the qualifier "regular".

Definition 3.6.4 An assembly S is ***partitioned*** if each element has precisely one realizer: if $r \Vdash_S x$ and $r \Vdash_S y$ then $x = y$.

Proposition 3.6.13 *A partitioned assemly is projective.*

Proof. Suppose that P is a partitioned assembly, $e : A \to B$ is a regular epi and $f : P \to B$ is realized by $\mathtt{f} \in \mathbb{A}'_{\|P\| \to \|B\|}$. By Proposition 3.6.10 there is $\mathtt{i} \in \mathbb{A}'_{\|B\| \to \|A\|}$ such that whenever $\mathtt{z} \Vdash_B z$ then $\mathtt{i}\,\mathtt{z} \Vdash_A z$ for some $y \in e^*(z)$. Because $e : |A| \to |B|$ is surjective, there exists by the axiom of choice a map $\overline{f} : |P| \to |A|$ such that $e(\overline{f}\,x) = f(x)$ for all $x \in |P|$. The map $\overline{f}$ is realized by $\langle x^{\|P\|} \rangle\, \mathtt{i}\,(\mathtt{f}\,x)$ because P is partitioned. ☐

Proposition 3.6.14 *Every assembly is a quotient of a partitioned assembly.*

Proof. Given an assembly S, let P be the partitioned assembly whose underlying type is $\|P\| = \|S\|$, the underlying set is the extension of $\Vdash_S$,

$$|P| = \{(\mathtt{x}, x) \in \mathbb{A}_{\|S\|} \times |S| \mid \mathtt{x} \Vdash_S x\},$$

and the existence predicate is $\mathsf{E}_P(x, \mathtt{x}) = \{\mathtt{x}\}$. The first projection $e : |P| \to |S|$, $e : (x, \mathtt{x}) \mapsto x$ is realized by $\mathtt{fst}$ and is a regular epi by Proposition 3.6.10. It is called the ***canonical cover*** of S. ☐

Exercise 3.6.4 Is canonical cover functorial?

The following notion is useful when one has to compute with concrete realizers.

Definition 3.6.5 An assembly S has ***canonical realizers*** when there exists $\mathtt{c} \in \mathbb{A}'_{\|S\| \to \|S\|}$ such that, for all $x \in |S|$ and $\mathtt{x} \in \|S\|$, if $\mathtt{x} \Vdash_S x$ then $\mathtt{c}\,\mathtt{x}{\downarrow}$, $\mathtt{c}\,\mathtt{x} \Vdash_S x$, and $\mathtt{c}\,(\mathtt{c}\,\mathtt{x}) = \mathtt{c}\,\mathtt{x}$. We say that $\mathtt{c}$ ***computes canonical realizers*** for S.

Not every assembly has canonical realizers. In fact, having them is equivalent to projectivity, as shown by the following characterization of projective assemblies.

Theorem 3.6.15 *The following are equivalent for an assembly S:*

1. *S is projective.*
2. *S the canonical cover is split.*[14]
3. *S has canonical realizers.*
4. *S is isomorphic to a partitioned assembly.*

14: A morphism $f : X \to Y$ is ***split*** if there is $g : Y \to X$ such that $f \circ g = \mathrm{id}_Y$.

Proof. To show that the first statement implies the second one, let S be a projective assembly and $e : P \to S$ the regular epimorphism constructed in Proposition 3.6.14. Because S is projective, id_S lifts along e to an assembly map $d : S \to P$ such that $e \circ d = \mathrm{id}_S$, hence e is split.

If the canonical cover $e : P \to S$ is split by $d : S \to P$ then any realizer of d computes canonical realiers for S.

If $\mathtt{c}$ computes canonical realizers for S then S is isomorphic to the partitioned assembly Q, defined by $|Q| = |S|$, $\|Q\| = \|S\|$ and

$$\mathtt{y} \Vdash_Q x \iff \exists \mathtt{x} \in \|S\| .\, \mathtt{x} \Vdash_S x \wedge \mathtt{y} = \mathtt{c}\,\mathtt{x}.$$

Finally, the fourth statement implies the first one because projectivity is preserved by isomorphism and partitioned assemblies are projective by Proposition 3.6.13. □

Exercise 3.6.5 Show that the full subcategory on the projective modest sets is equivalent to the category of *sets of realizers*, whose objects are pairs $(|S|, \|S\|)$ where $\|S\|$ is a type and $|S| \subseteq \mathbb{A}_{\|S\|}$. A morphism $f : (|S|, \|S\|) \to (|T|, \|T\|)$ is a map $f : |S| \to |T|$ that has a realizer $\mathtt{f} \in \mathbb{A}'_{\|S\| \to \|T\|}$ satisfying $\mathtt{f}\,\mathtt{x}{\downarrow}$ and $\mathtt{f}\,\mathtt{x} \in |T|$ for all $\mathtt{x} \in |S|$.

4 Realizability and logic

The idea that the elements of a set are represented by values of a datatype is familiar to programmers. In the previous chapter we expressed the idea mathematically in terms of realizability relations and assemblies. Programmers are less aware of, but still use, the fact that realizability carries over to logic as well: a logical statement can be validated by realizers.

4.1 The set-theoretic interpretation of logic

Let us first recall how the usual interpretation of classical first-order logic works. A *predicate* on a set S is a Boolean function $S \to 2$, where $2 = \{\bot, \top\}$ is the Boolean algebra on two elements. The Boolean algebra structure carries over from 2 to predicates, e.g., the conjunction of $p, q : S \to 2$ is computed element-wise as

$$(p \wedge q)x = px \wedge qx,$$

and similarly for other predicates. With this much structure we can interpret the propositional calculus. The quantifiers $\exists$ and $\forall$ can be interpreted too, because 2 is complete: given a predicate $p : S \times T \to 2$, define $\exists_S p : T \to 2$ and $\forall_S p : T \to 2$ by

$$(\exists_S p)y = \bigvee_{x \in S} p(x, y) \qquad \text{and} \qquad (\forall_S p)y = \bigwedge_{x \in S} p(x, y).$$

Categorical logic teaches us that the essential characteristic of the quantifiers is not their relation to infima and suprema, but rather an *adjunction*: $\exists_S p$ is the least predicate on T such that $p(x, y) \leq (\exists_S p)y$ for all $x \in S$, $y \in T$, and $\forall_S p$ is the largest predicate on T such that $(\forall_S p)y \leq p(x, y)$ for all $x \in S$, $y \in T$. The adjunction will carry over to the realizability logic, but completeness will not.

4.2 Realizability predicates

In a category a predicate on an object S is represented by a mono with codomain S. These form a preorder, where $u : T \rightarrowtail S$ is below $t : U \rightarrowtail S$, written $u \leq t$ or somewhat less precisely $U \leq T$, whhen u factors through t, i.e., there exists a morphism $f : U \to T$ such that

$$\begin{array}{ccc} & S & \\ {\scriptstyle u}\nearrow & & \nwarrow{\scriptstyle t} \\ U & \xrightarrow{f} & T \end{array}$$

commutes. Such an f is unique if it exists and is a mono.[1] If $u \leq t$ and $t \leq u$ we say that u and t are *isomorphic* and write $u \equiv t$. The induced partial order $\mathsf{Sub}(S) = \mathsf{Mono}(S)/\cong$ of *subobjects* can be used if one cares about antisymmetry, which we do not.

1: Such basic category-theoretic observations are excellent exercises. You should prove them yourself.

In the case of assemblies $\mathsf{Mono}(S)$ forms a *Heyting prealgebra*, which is enough to interpret *intuitionistic* propositional calculus, and there is enough additional structure to interpret the quantifiers too.

However, rather than working with the Heyting prealgebra of monos, we shall replace it with an equivalent one that expresses the predicates as maps into Heyting prelagebras.

Definition 4.2.1 A ***realizability predicate*** on an assembly S is given by a type $\|p\|$ and a map $p : |S| \to \mathcal{P}(\mathbb{A}_{\|p\|})$.

It is customary to write $\mathtt{r} \Vdash p x$ instead of $\mathtt{r} \in p x$ and to read this as "$\mathtt{r}$ realizes $p x$". Realizability predicates are more informative than Boolean predicates. The latter only express truth and falshood, whereas the former provide *computatational evidence* for validity of statements.

The set $\mathsf{Pred}(S) = \mathcal{P}(\mathbb{A}_{\|p\|})^{|S|}$ of all realizability predicates on S is a preorder for the ***entailment*** relation $\vdash$, defined as follows. Given $p, q \in \mathsf{Pred}(S)$, we define $p \vdash q$ to hold when there exists $\mathtt{i} \in \mathbb{A}'_{\|S\| \to \|p\| \to \|q\|}$ such that whenever $\mathtt{x} \Vdash_S x$ and $\mathtt{r} \Vdash p x$ then $\mathtt{i}\,\mathtt{x}\,\mathtt{r}{\downarrow}$ and $\mathtt{i}\,\mathtt{x}\,\mathtt{r} \Vdash q x$. Thus, $\mathtt{i}$ converts computational evidence of $p x$ to computational evidence of $q x$. Note that $\mathtt{i}$ recieves as input both a realizer for x and the evidence of $p x$.

Exercise 4.2.1 Verify that $\vdash$ is reflexive and transitive.

Theorem 4.2.1 *The preorders* $\mathsf{Mono}(S)$ *and* $\mathsf{Pred}(S)$ *are equivalent.*

Proof. Given $p \in \mathsf{Pred}(S)$, define the assembly S_p by

$$\begin{aligned} |S_p| &= \{x \in |S| \mid \exists \mathtt{r} \in \mathbb{A}_{\|p\|} .\, \mathtt{r} \in p x\}, \\ \|S_p\| &= \|S\| \times \|p\|, \\ q \Vdash_{S_p} x &\iff \mathtt{fst}\, q \Vdash_S x \land \mathtt{snd}\, q \Vdash p x. \end{aligned}$$

The subset inclusion $|S_p| \subseteq |S|$ is realized by $\mathtt{fst}$. The corresponding assembly map $i_p : S_p \to S$ is called the ***extension*** of p.

It is easy to check that $p \vdash q$ is equuvalent to $i_p \leq i_q$, therefore the assignment $p \mapsto S_p$ consistutes a monotone embedding $\mathsf{Pred}(S) \to \mathsf{Mono}(S)$. We still have to show that it is essentially surjective, i.e., that every mono $u : U \rightarrowtail S$, realized by $\mathtt{u}$, is equivalent to the extension of a predicate. Define the predicate p_u on S by $\|p_u\| = \|U\|$ and

$$\mathtt{r} \Vdash p_u\, x \iff \exists y \in U .\, u\, y = x \land \mathtt{y} \Vdash_U y.$$

We claim that i_{p_u} and u are isomorphic monos. The injection $u : |U| \to |S|$ restricts to a bijection $u : |U| \to |S_{p_u}|$, which is realized as a morphism $U \to S_{p_u}$ by $\langle y^{\|U\|} \rangle\, \mathtt{pair}\, (\mathtt{u}\, y)\, y$. Its inverse $u^{-1} : |S_{p_u}| \to |U|$ is realized by snd. □

4.3 The Heyting prealgebra of realizability predicates

Recall that a ***Heyting prealgebra*** $(H, \vdash)$ is a preorder (reflexive and transitive) with elements $\bot$, $\top$ and binary operations $\wedge$, $\vee$, and $\Rightarrow$ governed by the following rules of inference:[2]

$$\frac{}{\bot \vdash p} \qquad \frac{}{p \vdash \top} \qquad \frac{r \vdash p \qquad r \vdash q}{r \vdash p \wedge q} \qquad \frac{p \vdash r \qquad q \vdash r}{p \vee q \vdash r} \qquad \frac{r \wedge p \vdash q}{r \vdash p \Rightarrow q}$$

2: These "fractions" are inference rules stating that the bottom statement follows from the conjunction of the top ones. The double line indicates a two-way rule which additionally states that the bottom statement implies the conjunction of the top ones.

We say that elements $p, q \in H$ are ***equivalent***, written $p \dashv\vdash q$, if $p \vdash q$ and $q \vdash p$. A ***Heyting algebra*** is a Heyting prealgebra in which equivalence is equality, which just means that the preorder is also antisymmetric.

In a Heyting prealgebra there may be many smallest elements, but they are all equivalent to $\bot$. Similarly, the binary infima and suprema may exist in many copies, all of which are equivalent.

Proposition 4.3.1 *The preorder* $\mathsf{Pred}(S)$ *is a Heyting prealgebra.*

Proof. Define the predicates $\bot, \top : S \to \mathcal{P}(\mathbb{A}_{\mathtt{unit}})$ on an assembly S by

$$\bot x = \emptyset \qquad \text{and} \qquad \top x = \mathbb{A}_{\mathtt{unit}}.$$

That is, $\bot$ is realized by nothing and $\top$ by everything. It is easy to check that $\bot \vdash p \vdash \top$ for all $p \in \mathsf{Pred}(S)$.

For predicates p and q on S, let $p \wedge q$ be the predicate with $\|p \wedge q\| = \|p\| \times \|q\|$ and

$$\mathtt{r} \Vdash (p \wedge q)\, x \iff \mathtt{fst}\,\mathtt{r} \Vdash p \wedge \mathtt{snd}\,\mathtt{r} \Vdash q.$$

It is customary to write $p\, x \wedge q\, x$ instead of $(p \wedge q)\, x$, which we shall do henceforth, including for other connectives. Let us verify that $p \wedge q$ is the infimum of p and q. If $r \vdash p$ and $r \vdash q$ are witnessed by a and b, respectively, then $r \vdash p \wedge q$ is witnessed by $\langle x^{\|S\|}\rangle\, \langle u^{\|r\|}\rangle\, \mathtt{pair}\, (\mathsf{a}\, x\, u)\, (\mathsf{b}\, x\, u)$. Conversely, if c witnesses $r \vdash p \wedge q$ then $\langle x^{\|S\|}\rangle\, \langle u^{\|r\|}\rangle\, \mathtt{fst}\, (\mathsf{c}\, x\, u)$ witnesses $r \vdash p$ and $\langle x^{\|S\|}\rangle\, \langle u^{\|r\|}\rangle\, \mathtt{snd}\, (\mathsf{c}\, x\, u)$ witnesses $r \vdash q$.

Next, define $p \vee q$ to be the predicate with $\|p \wedge q\| = \|p\| + \|q\|$ and

$$\begin{aligned} \mathtt{r} \Vdash px \vee qx \iff & (\exists \mathtt{u}.\, \mathtt{r} = \mathtt{left}\,\mathtt{u} \wedge \mathtt{u} \Vdash px) \vee \\ & (\exists \mathtt{v}.\, \mathtt{r} = \mathtt{right}\,\mathtt{v} \wedge \mathtt{v} \Vdash qx). \end{aligned}$$

If a and b witness $p \vdash r$ and $q \vdash r$, respectively, then $p \vee q \vdash r$ is witnessed by

$$\langle x^{\|S\|}\rangle\, \langle w^{\|p\|+\|q\|}\rangle\, \mathtt{case}\, w\, (\mathsf{a}\, x)\, (\mathsf{b}\, x).$$

Conversely, if c witnesses $p \vee q \vdash r$ then $p \vdash r$ and $q \vdash r$ are witnessed by $\langle x^{\|S\|}\rangle\, \langle u^{\|p\|}\rangle\, \mathsf{c}\, x\, (\mathtt{left}\, u)$ and $\langle x^{\|S\|}\rangle\, \langle v^{\|q\|}\rangle\, \mathsf{c}\, x\, (\mathtt{right}\, v)$, respectively.

Finally, define $p \Rightarrow q$ to be the predicate with $\|p \Rightarrow q\| = \|p\| \to \|q\|$ and

$$\mathtt{r} \Vdash (px \Rightarrow qx) \iff \forall \mathtt{u} \in \mathbb{A}_{\|p\|}.\, \mathtt{u} \Vdash px \Rightarrow \mathtt{r}\,\mathtt{u} \Vdash qx.$$

That is, $\mathtt{r}$ maps realizers for px to realizers for qx. Note that $\mathtt{r} \in \mathbb{A}_{\|p\| \to \|q\|}$ and *not* $\mathtt{u} \in \mathbb{A}'_{\|p\| \to \|q\|}$, so we have an example of entailment $\vdash$ and implication $\Rightarrow$ not "being the same thing" (about which students of logic often wonder about). □

Exercise 4.3.1 Finish the proof by checking that the definition of $\Rightarrow$ validates the inference rules for implication.

In intuitionistic logic ***negation*** $\neg p$ is an abbreviation for $p \Rightarrow \bot$. The following holds:

$$r \Vdash \neg px \iff \neg \exists \mathsf{s} \in \mathbb{A}_{\|p\|} .\, \mathsf{s} \Vdash px,$$
$$r \Vdash \neg\neg px \iff \exists \mathsf{s} \in \mathbb{A}_{\|p\|} .\, \mathsf{s} \Vdash px.$$

4.4 Quantifiers

Categorical logic teaches us that the existential and universal quantifiers are defined as the left and right adjoints to weakening. Let us explain what this means.

Weakening along the projection $\pi_1 : S \times T \longrightarrow S$ is an operation which maps a mono $V \rightarrowtail S$ to the mono $V \times T \rightarrowtail S \times T$. This is a monotone map from $\mathsf{Mono}(S)$ to $\mathsf{Mono}(S \times T)$. Existential quantification is its left adjoint, i.e., a monotone map $\exists_T : \mathsf{Mono}(S \times T) \longrightarrow \mathsf{Mono}(S)$ such that, for all monos $U \rightarrowtail S \times T$ and $V \rightarrowtail S$,

$$U \leq V \times T \iff \exists_T U \leq V.$$

Similarly, universal quantification is the right adjoint:

$$V \times T \leq U \iff V \leq \forall_T U.$$

From the above adjunctions the usual laws of inference for the existential and universal quantifiers follow. We verify that such adjoints for the Heyting prealgebras of realizability predicates.

Weakening along $\pi_1 : S \times T \longrightarrow S$ takes $p \in \mathsf{Pred}(S)$ to the predicate $p \times T \in \mathsf{Pred}(S \times T)$ defined by

$$\|p \times T\| = \|p\| \qquad \text{and} \qquad (p \times T)(x, y) = px,$$

where $x \in S$ and $y \in T$. We claim that the left adjoint $\exists_T$ maps $q \in \mathsf{Pred}(S \times T)$ to $\exists_T q \in \mathsf{Pred}(S)$ with $\|\exists_T q\| = \|T\| \times \|q\|$ and

$$\mathtt{r} \Vdash (\exists_T q)(x) \iff \exists y \in |T| .\, \mathtt{fst}\, Rr \Vdash_T y \wedge \mathtt{snd}\, \mathtt{r} \Vdash q(x, y).$$

Let us verify that we have a left adjoint to weakening. Suppose $p \in \mathsf{Pred}(S \times T)$ and $q \in \mathsf{Pred}(S)$. If $p \leq q \times T$ is witnessed by $\mathtt{f}$ then $\exists_T p \leq q$ is witnessed by $\langle x^{\|S\|} \rangle \, \langle e^{\|T\| \times \|p\|} \rangle \, \mathtt{f}\, (\mathtt{pair}\, x\, (\mathtt{fst}\, e))\, (\mathtt{snd}\, e)$. Conversely, if $\exists_T p \leq q$ is witnessed by $\mathtt{g}$ then $p \leq q \times T$ is witnessed by $\langle r^{\|S\| \times \|T\|} \rangle \, \langle s^{\|p\|} \rangle \, \mathtt{g}\, (\mathtt{fst}\, r)\, (\mathtt{pair}\, (\mathtt{snd}\, r)\, s)$.

Similarly, the universal quantifier $\forall_T$ maps $q \in \mathsf{Pred}(S \times T)$ to $\forall_T q \in \mathsf{Pred}(S)$ with $\|\forall_T q\| = \|T\| \to \|q\|$ and

$$\mathtt{r} \Vdash (\forall_T q)(x) \iff \forall y \in |T|.\, \forall \mathtt{y} \in \mathbb{A}_{\|T\|}.\, \mathtt{y} \Vdash_T y \Rightarrow \mathtt{r}\,\mathtt{y} \Vdash q(x, y).$$

To verify that this is the right adjoint, suppose $p \in \mathsf{Pred}(S \times T)$ and $q \in \mathsf{Pred}(S)$. If $q \times T \le p$ is witnessed by $\mathtt{f}$ then $q \le \forall_T p$ is witnessed by $\langle x^{\|S\|}\rangle\, \langle r^{\|q\|}\rangle\, \langle y^{\|T\|}\rangle\, \mathtt{f}\,(\mathtt{pair}\, x\, y)\, r$. Conversely, if $q \le \forall_T p$ is witnessed by $\mathtt{g}$ then $q \times T \le p$ is witnessed by $\langle s^{\|S\| \times \|T\|}\rangle\, \langle r^{\|q\|}\rangle\, \mathtt{g}\,(\mathtt{fst}\, s)\, r\, (\mathtt{snd}\, s)$.

It is customary to write $\exists y \in T.\, q(x, y)$ and $\forall y \in T.\, q(x, y)$ instead of $\exists_T q$ and $\forall_T q$.

Exercise 4.4.1 Show that the usual inference rules for quantifiers follow form them being adjoint to weakneing.

4.5 Substitution

We have so far ignored the most basic logical operation of all, which is *substitution*. Terms may be substituted into terms and into formulas.

Substitution of terms into terms is interpreted as composition. Think of a term $s(x)$ of type S with a free variable x of type T as a map $s : T \to S$. Given $t : U \to T$, construed as a term $t(y)$ of type T with a free variable y of type U, the substitution of $t(y)$ for x in $s(x)$ yields $s(t(y))$, which is just $(s \circ t)(y)$, so $s \circ t : U \to S$.

Substitution of terms into predicates corresponds to composition too, by an analogous argument. Think about how it works in Set. Given a predicate $p : T \to 2$ on a set S and a term $t : S \to T$, the composition $p \circ t : S \to 2$ corresponds to substituting t into p. Indeed, if we replace y with $t(x)$ in the formula $p(x)$ we obtain $p(t(y))$, which is just $(p \circ t)(y)$.

Exercise 4.5.1 You may have heard the slogan "substitution is pullback". Explain how the slogan arises when predicates are viewed as subsets, rather than maps into 2

Let us introduce a notation for substitution. Given assembly maps $t : U \to T$ and $s : T \to S$, and $p \in \mathsf{Pred}(T)$, define the ***substitution*** of t into s and p to be precomposition with t:

$$t^* s = s \circ t : U \to S \qquad \text{and} \qquad t^* p = p \circ t \in \mathsf{Pred}(U).$$

We still have some work to do, namely check that substitution is functorial and that it commutes with the logical connective and the quantifiers. The former guarantees that the identity substitution and compositions of substitutions act in the expected way, and the latter that substituting preserves the logical structure of a formula. Functoriality means that

$$\mathrm{id}_S^* s = s \qquad \text{and} \qquad (u \circ t)^* s = t^*(u^* s),$$

which is of course the case.[3]

3: Substitution is *contravariant* because it reverses the order of composition.

Proposition 4.5.1 *Realizability predicates and substitution constitute a functor*

$$\mathsf{Asm}(\mathbb{A}, \mathbb{A}')^{\mathrm{op}} \longrightarrow \mathsf{Heyt}$$

from (the opposite category of) assemblies to the category of Heyting prealgebras. Moreover, substitution preserves the quantifiers.

Proof. We already checked that substitution is functorial, but we still need to verify that it yields a homomorphism of Heyting prealgebras, i.e.,

$$\begin{aligned} t^* \top &= \top, \\ t^* \bot &= \bot, \\ t^*(p \wedge q) &= t^* p \wedge t^* q, \\ t^*(p \vee q) &= t^* p \vee t^* q, \\ t^*(p \Rightarrow q) &= t^* p \Rightarrow t^* q. \end{aligned}$$

These hold because they are defined pointwise. For instance, given $p, q \in \mathsf{Pred}(T)$, define $\wedge' : \mathcal{P}(\mathbb{A}_{\|p\|}) \times \mathcal{P}(\mathbb{A}_{\|q\|}) \longrightarrow \mathcal{P}(\mathbb{A}_{\|p\| \times \|q\|})$ by

$$A \wedge' B = \{\mathtt{r} \in \mathbb{A}_{\|p\| \times \|q\|} \mid \mathtt{fst}\,\mathtt{r} \in A \wedge \mathtt{snd}\,\mathtt{r} \in B\},$$

and observe that $p \wedge q = \wedge' \circ \langle p, q \rangle$ therefore, for any $t : U \longrightarrow T$,

$$t^*(p \wedge q) = \wedge' \circ \langle p, q \rangle \circ t = \wedge' \circ \langle p \circ t, q \circ t \rangle = t^* p \wedge t^* q.$$

The other connectives are dealt with analogously.

Preservation of quantifier amounts to checking the ***Beck-Chevalley conditions***:

$$\begin{aligned} t^*(\exists_T p) &= \exists_T (t^* p), \\ t^*(\forall_T p) &= \forall_T (t^* p). \end{aligned}$$

□

Exercise 4.5.2 Verify the Beck-Chevalley conditions for $\forall_T$ and $\exists_T$.

4.6 Equality

Equality qua binary predicate on an object S is represented by the diagonal morphism $\Delta : S \longrightarrow S \times S$. In assemblies this is the map $\Delta x = (x, x)$, which is realized by $\langle x^{\|S\|} \rangle\, \mathtt{pair}\, x\, x$. We claim that the corresponding realizability predicate $\mathrm{eq} \in \mathsf{Pred}(S \times S)$ is given by $\|e\| = \mathtt{unit}$ and

$$\star \Vdash \mathrm{eq}(x, y) \iff x = y.$$

To verify the claim, we need to factor Δ and $i_{\mathrm{eq}} : S_{\mathrm{eq}} \to S$ from Theorem 4.2.1 through each other. We have

$$
\begin{aligned}
|S_{\mathrm{eq}}| &= \{(x, y) \in |S| \times |S| \mid x = y\}, \\
i_{\mathrm{eq}}(x, y) &= (x, y), \\
\|S_{\mathrm{eq}}\| &= \|S\| \times \|S\| \times \mathtt{unit}, \\
q \Vdash_{S_{\mathrm{eq}}} x &\Leftrightarrow \mathtt{fst}\, q \Vdash_S x \land \mathtt{snd}\, q = \star.
\end{aligned}
$$

Thus Δ factors through i_{eq} via $x \mapsto (x, x)$ and i_{eq} through Δ via $(x, y) \mapsto x$, each of which is easily seen to be realized.

4.7 Summary of realizability logic

We summarize the realizability interpretation of intuitionistic logic for easy lookup.

A realizability predicate p on an assembly S is given by a type $\|p\|$ and a map $p : |S| \to \mathcal{P}(\|p\|)$. The collection of all realizability predicates on S is denoted $\mathsf{Pred}(S)$. Given $p, q \in \mathsf{Pred}(S)$, p entails q, written $p \vdash q$, when there is $\mathtt{i} \in \mathbb{A}'_{\|S\| \to \|p\| \to \|q\|}$ such that

$$\forall x \in |S|.\, \forall \mathtt{x} \in \|S\|.\, \forall \mathtt{r} \in \|p\|.\, \mathtt{r} \Vdash_S x \land \mathtt{r} \Vdash p \Rightarrow \mathtt{i}\,\mathtt{x}\,\mathtt{r} \Vdash qx.$$

The entailment relation is a preorder.

The underlying types of connectives are computed as follows, where $p, q \in \mathsf{Pred}(S)$ and $r \in \mathsf{Pred}(T \times S)$:

$$
\begin{aligned}
\|\bot\| &= \mathtt{unit}, \\
\|\top\| &= \mathtt{unit}, \\
\|s = t\| &= \mathtt{unit}, \\
\|p \land q\| &= \|p\| \times \|q\|, \\
\|p \lor q\| &= \|p\| + \|q\|, \\
\|p \Rightarrow q\| &= \|p\| \to \|q\|. \\
\|\forall x \in T.\, r(x, -)\| &= \|T\| \to \|r\|, \\
\|\exists x \in T.\, r(x, -)\| &= \|T\| \times \|r\|.
\end{aligned}
$$

We write $\mathtt{r} \Vdash px$ instead of $\mathtt{r} \in px$. The Heyting prealgebra structure on $\mathsf{Pred}(S)$ is as follows, where $p, q \in \mathsf{Pred}(S)$, $r \in \mathsf{Pred}(T \times S)$, and

$s, t \in |S|$:

$$
\begin{aligned}
\mathtt{r} \Vdash \bot &\Leftrightarrow \bot \\
\mathtt{r} \Vdash \top &\Leftrightarrow \top \\
\mathtt{r} \Vdash s = t &\Leftrightarrow s = t \land \mathtt{r} = \star \\
\mathtt{r} \Vdash px \land qx &\Leftrightarrow \mathtt{fst}\,\mathtt{r} \Vdash p \land \mathtt{fst}\,\mathtt{r} \Vdash q \\
\mathtt{r} \Vdash px \lor qx &\Leftrightarrow (\exists \mathtt{u}.\, \mathtt{r} = \mathtt{left}\,\mathtt{u} \land \mathtt{u} \Vdash px) \lor \\
& (\exists \mathtt{v}.\, \mathtt{r} = \mathtt{right}\,\mathtt{v} \land \mathtt{v} \Vdash qx). \\
\mathtt{r} \Vdash (p \Rightarrow q) &\Leftrightarrow \forall \mathtt{s} \in \mathbb{A}_{\|p\|}.\, \mathtt{s} \Vdash p \Rightarrow \mathtt{r}\,\mathtt{s} \Vdash q \\
\mathtt{r} \Vdash \exists y \in T.\, p(x, y) &\Leftrightarrow \exists y \in |T|.\, \mathtt{fst}\,\mathtt{r} \Vdash_T y \land \mathtt{snd}\,\mathtt{r} \Vdash p(x, y) \\
\mathtt{r} \Vdash \forall y \in T.\, p(x, y) &\Leftrightarrow \forall y \in |T|.\, \forall \mathtt{y}.\, \mathtt{y} \Vdash_T y \Rightarrow \mathtt{r}\,\mathtt{y} \Vdash q(x, y).
\end{aligned}
$$

We have the following soundness theorem.

Theorem 4.7.1 *The realizability interpretation of intuitionistic logic is sound.*

Proof. We already proved the theorem when we checked that the predicates form Heyting prealgebras and that the quantifiers validate the desired inference rules. □

Let us spell out what the theorem says: if a formula ϕ is intuitionistically provable then there exists $\mathtt{r} \in \mathbb{A}'_{\|\phi\|}$ such that $\mathtt{r} \Vdash \phi$. Moreover, the realizer $\mathtt{r}$ can be constructed from the proof of ϕ, one just has to follow the inference steps and apply the corresponding realizability constructions.

The above clauses are reminiscent of the Curry-Howard correspondence between logic and type theory. The similarity is not accidental, as both realizability and Martin-Löf type theory aim to formalize the Brouwer-Heyting-Kolmogorov explanation of intuitionistic logic. Let us not forget, however, that realizability was the first such rigorous explanation.

4.8 Classical and decidable predicates

Some classes of predicates are of special interest.

4.8.1 Classical predicates

A formula ϕ is ***classical***[4] or ***¬¬-stable*** when $\neg\neg\phi \Rightarrow \phi$. (We do not require $\phi \Rightarrow \neg\neg\phi$ because it holds anyway.)

4: The terminology is non-standard but is vindicated below by the fact that $\nabla 2$ classifies the classical predicates. In any case, almost any terminology seems better than "¬¬-stable".

Proposition 4.8.1 *A predicate $p \in \mathsf{Pred}(S)$ is classical if, and only if, there exists $\mathtt{q} \in \mathbb{A}'_{\|S\| \to \|p\|}$ such that, if $\mathtt{x} \Vdash_S x$ and $px \neq \emptyset$ then $\mathtt{q}\,\mathtt{x} \Vdash px$.*

Exercise 4.8.1 Prove Proposition 4.8.1. More precisely, you need to show that the stated condition is equivalent to $\top \vdash \neg\neg p \Rightarrow p$.

Thus a predicate is classical when its realizers can be computed at will, when they exist. An important fact about assemblies is the following.

Proposition 4.8.2 *Equality on an assembly is classical.*

Proof. Apply Proposition 4.8.1 with $\mathtt{r} = \mathtt{K}\,\star$. □

It is useful to have a simple syntactic criterion for recognizing classical formulas. Say that a formula is ***almost negative***[5] when it has one of the following forms:

- $\bot, \top, s = t$,
- $\phi \land \psi$ where ϕ and ψ are almost negative,
- $\forall x \in S.\, \phi$ where ϕ is almost negative,
- $\phi \Rightarrow \psi$ where ψ is almost negative.

5: A ***negative*** formula is like an almost negative one, except that the antecedent in an implication is required to be negative, too.

Proposition 4.8.3 *An almost negative formula is classical.*

Exercise 4.8.2 Prove Proposition 4.8.3 *without* resorting to realizability. You should just give suitable intuitionistic proofs. For extra credit, do it in a proof assistant.

The classical predicates can be used to interpret classical logic.

Exercise 4.8.3 Consider the following preorders:

- the sub-preorder $\mathsf{RegMono}(S) \subseteq \mathsf{Mono}(S)$ on the *regular* monos,
- the sub-preorder $\mathsf{Pred}_{\neg\neg}(S) \subseteq \mathsf{Pred}(S)$ on the classical predicates,
- the powerset $\mathcal{P}(|S|)$ ordered by $\leq$.

Show that these preorders are equivalent, and hence complete Boolean algebras. For extra credit, explain in what sense the equivalence is *functorial*.

4.8.2 Decidable predicates

Recall that a formula ϕ is ***decidable***[6] when $\phi \lor \neg\phi$.

6: Every decidable statement is classical, but the converse cannot be shown intuitionistically. If you are confused, you might be thinking of the equivalence between excluded middle ("all predicates are decidable") and double-negation elimination ("all predicates are ¬¬-stable").

Proposition 4.8.4 *A predicate $p \in \mathsf{Pred}(S)$ is decidable if, and only if, there exists $\mathtt{d} \in \mathbb{A}'_{\|S\| \to \mathtt{bool}}$ such that, for all $x \in |S|$ and $\mathtt{x} \in \|S\|$, if $\mathtt{x} \Vdash_S x$ then $\mathtt{d}\,\mathtt{x}{\downarrow}$ and*

$$\mathtt{d}\,\mathtt{x} = \begin{cases} \mathtt{true} & \text{if } px \neq \emptyset, \\ \mathtt{false} & \text{if } px = \emptyset. \end{cases}$$

Proof. If $\mathtt{r} \Vdash p \lor \neg p$ then we may take

$$\mathtt{d} = \langle \mathtt{x}^{\|S\|} \rangle\, \mathtt{case}\,(\mathtt{r}\,\mathtt{x})\,\mathtt{true}\,\mathtt{false}.$$

Conversely, suppose $\mathtt{d}$ with the stated property exists. Because a decidable predicate is stable, by Proposition 4.8.1 there exists $\mathtt{q} \in \mathbb{A}'_{\|S\| \to \|p\|}$ such that, if $\mathtt{x} \Vdash_S x$ and $px \neq \emptyset$ then $\mathtt{r}\,\mathtt{x} \Vdash px$. Now the realizer

$$\langle _^{\mathtt{unit}} \rangle\, \langle \mathtt{x}^{\|S\|} \rangle\, \mathtt{if}\,(\mathtt{d}\,\mathtt{x})\,(\mathtt{left}(\mathtt{q}\,\mathtt{x}))\,(\mathtt{right}(\langle _^{\|S\|} \rangle\, \mathtt{K}))$$

witnesses $\top \vdash p \lor \neg p$. □

The realizer $\mathtt{d}$ from Proposition 4.8.4 can be thought of as a decision procedure for p, although what that means depends on the underlying model of computation. If $\mathbb{A}$ retracts to $\mathbb{K}_1$, see Subsection 2.8.2, then $\mathtt{d}$ does indeed correspond to an algorithm in the usual sense of the word. But a topological model of computation, such as the graph model or Kleene's second algebra, $\mathtt{d}$ amounts to having a *topological separation* of the underlying space into two disjoint open sets.

4.8.3 Predicates classified by two-element assemblies

Consider a two-element assembly T with $|T| = \{0, 1\}$, like we did in Subsection 3.2.5. Say that a predicate $p \in \mathsf{Pred}(S)$ is ***classified*** by an assembly map $c : S \to T$, called the ***characteristic map*** of p, when

$$\top \vdash \forall x \in S.\, px \Leftrightarrow cx = 1$$

is realized. Because the above is a negative formula, it is $\neg\neg$-stable and so its realizers are uninformative. In other words, the statement is realized if, and only if,

$$\forall x \in |S|.\, px \neq \emptyset \Leftrightarrow cx = 1.$$

Exercise 4.8.4 Show that a predicate has at most one characteristic map.

For a specific choices of two-element assemblies we obtain known classes of predicates.

Proposition 4.8.5 $\nabla 2$ *and* $\mathbb{2}$ *classify the classical and decidable predicates, respectively.*

Proof. Let us show that $\nabla 2$ classifies the classical predicates on an assembly S. Given an assembly map $f : S \to \nabla 2$, defined $p_f \in \mathsf{Pred}(S)$ by $\|p_f\| = \mathtt{unit}$ and

$$\mathtt{r} \Vdash p_f x \iff f x = 1.$$

Obviously, p_f is $\neg\neg$-stable. Conversely, given a classical $p \in \mathsf{Pred}(S)$, define $f_p : S \to \nabla 2$ by

$$f_p x = \begin{cases} 1 & \text{if } px \neq \emptyset, \\ 0 & \text{if } px = \emptyset. \end{cases}$$

The map f_p is realized because every map into a constant assembly is. We claim that $f \mapsto p_f$ and $p \mapsto f_p$ form a bijective correspondence. It is easy to check that $f = f_{p_f}$ for all $f : S \to \nabla 2$. For the other direction, we need to show that a classical $p \in \mathsf{Pred}(S)$ satisfies $p \dashv\vdash p_{f_p}$. One direction is easy, and other one not much harder with the help of Proposition 4.8.1.

It remains to be checked that $\mathbb{2}$ classifies the decidable predicates. The hard part of the proof was already done in Proposition 4.8.4, and we leave the rest as an exercise. □

The ***law of excluded middle*** states that all predicates are decidable. It is never realized.

Proposition 4.8.6 *There exists a non-decidable predicate.*

Proof. Consider the predicate classified by $\mathrm{id}_{\nabla 2}$. More precisely, it is the predicate $p \in \mathsf{Pred}(\nabla 2)$ defined by $\|p\| = \mathtt{unit}$ and $p0 = \emptyset$, $p1 = \mathbb{A}_{\mathtt{unit}}$. It is not decidable because every assembly map $\nabla 2 \to \mathbb{2}$ is constant, since $\mathbb{2}$ is modest. □

The ***law of double negation*** states that all predicates are classical. Because it is inter-derivable with excluded middle, it cannot be valid in realizability logic.

Exercise 4.8.5 Say that a predicate $p \in \mathsf{Pred}(S)$ is ***semidecidable*** if it is classified by the Rosolini dominance Σ^0_1. Suppose $\mathtt{r} \in \mathbb{A}'_{\|S\| \to \mathtt{nat} \to \mathtt{bool}}$ is such that whenever $\mathtt{x} \Vdash_S x$ and $n \in \mathbb{N}$ then $\mathtt{r}\,\mathtt{x}\,\overline{n} \in \{\mathtt{true}, \mathtt{false}\}$. Define $q_{\mathtt{r}} \in \mathsf{Pred}(S)$ by $\|q_{\mathtt{r}}\| = \mathtt{unit}$ and

$$\star \Vdash q_{\mathtt{r}}\, x \iff \exists n \in \mathbb{N}.\, \mathtt{r}\,\mathtt{x}\,\overline{n} = \mathtt{true}.$$

Show that a predicate is semidecidable if, and only if, it is equivalent to some $q_{\mathtt{r}}$.

Exercise 4.8.6 Is there a pca $\mathbb{A}$ such that in $\mathsf{Asm}(\mathbb{A})$ the decidable and semidecidable predicates coincide?

One might ask whether there is a two-element assembly Ω which classifies *all* predicates, as that would imply that assemblies form a topos. Alas, this is not the case.

Proposition 4.8.7 *If a predicate is classified by a two-element assembly then it is ¬¬-stable.*

Proof. If a predicate is classified by T then it is also classified by $\nabla 2$, therefore it is ¬¬-stable. □

The point is that not all predicates are ¬¬-stable.

Exercise 4.8.7 Give an example of a realizability predicate which is not ¬¬-stable.

Realizability and type theory 5

In everyday mathematics *parametrized* constructions are commonplace. For example, when a mathematical text says “consider a continuous map $f : [a,b] \to \mathbb{R}$”, there is an implicit use of the parametrized set $[a,b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$, where a and b are the parameters. And whenever in algebra we say “the cyclic group $\mathbb{Z}_n$”, that is not a single group, but a *family* of groups parameterized by $n \in \mathbb{N}$.

The language of such parameterized constructions is ***(dependent) type theory***. It is applicable in many settings, including realizability. In this chapter we shall give an interpretation of type theory in terms of families of assemblies.

5.1 Families of sets

To set the scene, let us review the set-theoretic model of type theory. A ***family of sets*** is a map $A : I \to \mathsf{Set}$ from an ***index set*** I to the class of all sets. We say that A *is indexed by* I or that it is *a family over base* I. Let $\mathsf{Fam}(I)$ be the class of all families indexed by I.

Each $\mathsf{Fam}(I)$ is a category whose objects are the families indexed by I. A morphism $f : A \to B$, where $A, B \in \mathsf{Fam}(I)$, is a ***map of families*** $f : A \to B$, which is a family of maps $f_i : A_i \to B_i$, parameterized by $i \in I$. Such maps are composed index-wise.

Exercise 5.1.1 Recall the definition of the ***slice category*** Set/I: an object is a map $a : A \to I$ with codomain I, and a morphism a map $f : A \to B$ such that $b \circ f = a$:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & {\scriptstyle a}\searrow \quad \swarrow{\scriptstyle b} & \\ & I & \end{array}$$

A map into I is called a ***display map*** over the ***base*** I, and its domain the ***total space***. (The terminology is inspired by a geometric picture of a bundle over a space.)

For each $i \in I$ we define the ***fiber*** of a at i to be the inverse image $a^*\{i\} = \{x \in A \mid ax = i\}$. Thus a display map a over I yields an I-indexed family $i \mapsto a^*\{i\}$ of fibers. Conversely, a family $A : I \to \mathsf{Set}$ determines the display map, namely the first projection $\Sigma_I A \to I$.

Verify that the passages between $\mathsf{Fam}(I)$ and Set/I constitute an equivalence of categories. As a first step you should determine how the equivalence acts on morphisms.

A map $r : J \to I$, induces ***reindexing*** $r^* : \mathsf{Fam}(I) \to \mathsf{Fam}(J)$ by precomposition, $r^*A = A \circ r$. Reindexing is contravariantly functorial,

$$\mathrm{id}_I^* A = A \qquad \text{and} \qquad (s \circ r)^* A = r^*(s^* A).$$

Therefore, families and reindexings form a functor $\mathsf{Set}^{\mathsf{op}} \longrightarrow \mathsf{Cat}$.

5.1.1 Products and sums of families of sets

The two fundamental operations on families of sets are the cartesian products and sums. Given a family $A \in \mathsf{Fam}(I)$, its ***product*** $\Pi_I A$ and ***sum*** $\Sigma_I A$ are respectively the sets

$$\Pi_I A = \{u : I \to \textstyle\bigcup_{i \in I} A_i \mid \forall i \in I.\, u i \in A_i\},$$
$$\Sigma_I A = \{(i, a) \mid i \in I \land a \in A_i\}.$$

The elements of the product are called ***choice maps***.

We shall need their generalized forms, where the product and the sum is taken with respect to a reindexing, as follows. Consider $r : J \to I$ and $A \in \mathsf{Fam}(J)$. For $K \subseteq I$ write[1] $r^* K = \{j \in J \mid r j \in K\}$, and define the ***product*** $\Pi_r A \in \mathsf{Fam}(I)$ and the ***sum*** $\Sigma_r A \in \mathsf{Fam}(I)$ by

$$(\Pi_r A)_i = \{u : r^*\{i\} \to \textstyle\bigcup_{j \in r^*\{i\}} A_j \mid \forall j \in r^*\{i\}.\, u j \in A_j\} \tag{5.1}$$
$$(\Sigma_J A)_i = \{(j, x) \mid j \in r^*\{i\} \land x \in A_j\}.$$

1: Careful, the notation r^* is used both for $r^* : \mathcal{P}(I) \to \mathcal{P}(J)$ and $r^* : \mathsf{Fam}(I) \to \mathsf{Fam}(J)$. The coincidence is not accidental.

Exercise 5.1.2 Show that $\Pi_r : \mathsf{Fam}(J) \to \mathsf{Fam}(I)$ is a functor by providing a suitable action on the morphisms, and similarly for Σ_r.

The distinguishing feature of products and sums is that they are adjoint to reindexing,

$$\Sigma_r \dashv r^* \dashv \Pi_r.$$

Concretely, the above amounts to having isomorphisms, natural in $A \in \mathsf{Fam}(J)$ and $B \in \mathsf{Fam}(I)$,

$$\mathsf{Hom}_{\mathsf{Fam}(I)}(\Sigma_r A, B) \cong \mathsf{Hom}_{\mathsf{Fam}(J)}(A, r^* B)$$

and

$$\mathsf{Hom}_{\mathsf{Fam}(I)}(B, \Pi_r A) \cong \mathsf{Hom}_{\mathsf{Fam}(J)}(r^* B, A).$$

We spell out the second isomorphism and leave the first one as an exercise. Given a map of families $f : B \to \Pi_r A$, define $\hat{f} : r^* B \to A$ by

$$\hat{f}_j x = f_{rj} x j, \tag{5.2}$$

and given $g : r^* B \to A$ define $\check{g} : B \to \Pi_r A$ by

$$\check{g}_i x j = g_j x. \tag{5.3}$$

It is easy to see that $f \mapsto \hat{f}$ and $g \mapsto \check{g}$ are inverses of each other. Checking naturality is less pleasant but instructive.

Exercise 5.1.3 Complete the verification of $\Sigma_r \dashv r^* \dashv \Pi_r$.

5.1.2 Type theory as the internal language

Having identified the relevant set-theoretic structure, we can now interpret the language of type theory in set theory.

Contexts

In type theory the index sets are called ***contexts***. In practice they are not arbitrary sets (although they can be), but are rather built up by introduction of new parameters in an inductive fashion:

- the empty context is the singleton[2] $1 = \{\star\}$,
- given a context Γ and a family of sets $A \in \mathsf{Fam}(\Gamma)$, the ***extended context*** is the sum $\Sigma_\Gamma A$.

2: A family of sets which does not depend on any parameters is just a fixed set, so an element of $\mathsf{Fam}(1)$. If you think the empty context should be $\emptyset$, consider what $\mathsf{Fam}(\emptyset)$ is like.

By iterating context extension we obtain a ***telescope***

$$\Sigma_{\Sigma_{\cdots \Sigma_{\Sigma_1 \cdots A_{n-2}} A_{n-1}}} A_n.$$

Such a nested sum is unwieldy, so we write it as

$$x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n.$$

where $x_1, \ldots, x_n$ are distinct variable names. This way we may access the components of the telescope by referring to the variables, rather than having to use iterated projections. The elements of a telescope are tuples[3] $(a_1, \ldots, a_n)$ such that $a_i \in A_{(a_1,\ldots,a_{i-1})}$ for $i = 1, \ldots, n$. Once again, "context" is a synonym for "set", but in practice we use telescopes.

3: To be quite precise, the elements are nested pairs $(((\star, a_1), a_2) \cdots, a_n)$ but we might as well use these as the definition of n-tuples $(a_1, \ldots, a_n)$.

Example 5.1.1 Suppose a mathematical text says

"Consider a continuous $f : [a, b] \to \mathbb{R}$ bounded by $M \in \mathbb{R}$."

What precisely is the context? It is implied that $a, b \in \mathbb{R}$, so at first we might think that the context is

$$a{:}\mathbb{R}, \quad b{:}\mathbb{R}, \quad f{:}[a, b] \to \mathbb{R}, \quad M{:}\mathbb{R}.$$

However, there are also three *hypotheses*, namely that $a < b$, that f is continuous, and that f is bounded by M. Mathematical tradition would have us ignore these, because it demands that proofs and logical statements be considered second-class. Indeed, notice how the text introduces names a, b, f, M for all the entities *except* the hypotheses, and even these notes refer to theorems by mere unmemorable numbers, as if it were forbidden to name them. The correct context is, written

vertically for readability,

$$\begin{aligned}
&a : \mathbb{R}, \\
&b : \mathbb{R}, \\
&p : (a < b), \\
&f : [a, b] \to \mathbb{R}, \\
&q : \mathsf{continuous}(f), \\
&M : \mathbb{R}, \\
&r : \forall x \in [a, b].\, f(x) \leq M
\end{aligned}$$

However, we now face a difficulty: in what sense are logical formulas, such as $a < b$ and $\mathsf{continuous(f)}$ families of sets? They must be, if they are to appear in contexts. We shall resolve the matter in **??**.

Type families and their elements

When type theory is used to talk about set theory, we prefer to say ***type*** and ***type family*** instead of "set" and "set family", and write

$$\Gamma \vdash A \ \mathsf{type}$$

for $A \in \mathsf{Fam}(\Gamma)$. The ***elements*** of A are its choice maps. We write

$$\Gamma \vdash t : A$$

when t is such a choice map.

Example 5.1.2 To see why it makes sense to call the choice maps "elements", we translate the statement

> "$(a + b)/2$ *is an element of the closed interval* $[a, b]$."

to the type-theoretic terminology. First, the text expects us to guess that a and b are reals such that $a < b$, so the context is

$$a{:}\mathbb{R}, \quad b{:}\mathbb{R}, \quad p{:}(a < b).$$

Over this context we define a type family C of closed intervals by[4]

$$C_{(a,b,p)} = [a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}.$$

The mid-point is map m which assigns to each interval its mid-point, $m(a, b, p) = (a + b)/2$, which of course is just a choice map for C.

4: Dragging along the argument p seems a little bureaucratic. In practice we would of course drop it, and in a proof assistant we might use one of several mechanisms that hide it.

Dependent products and sums

If a family is indexed by a telescope with parameters $x_1, \ldots, x_n$, we may wish to form the cartesian product with respect to just x_n, which is accomplished by taking the product (5.1) along a suitable reindexing. Suppose $\Gamma, x{:}A \vdash B \ \mathsf{type}$ and let $p : (\Gamma, x{:}A) \to \Gamma$ be the first projection $p(\gamma, a) = \gamma$. Define the ***product*** of B to be the type family $\Gamma \vdash \Pi_p B \ \mathsf{type}$.

Unfolding the definitions shows that, for $\gamma \in \Gamma$,

$$\begin{aligned}(\Pi_p B)_\gamma &= \{u : p^*\{\gamma\} \to \textstyle\bigcup_{\delta \in p^*\{\gamma\}} B_\delta \mid \forall \delta \in p^*\{\gamma\}.\, u\delta \in B_\delta\} \\ &\cong \{u : A_\gamma \to \textstyle\bigcup_{a \in A_\gamma} B_{(\gamma,a)} \mid \forall a \in A_\gamma.\, ua \in B_{(\gamma,a)}\},\end{aligned}$$

which is precisely the desired parameterized version of cartesian product.

A similar line of though show that the ***sum*** of a family $\Gamma, x{:}A \vdash B$ type is the family $\Gamma \vdash \Sigma_p B$, where p is as above. It is the parameterized version of the disjoint sum, or coproduct, of a family:

$$\begin{aligned}(\Sigma_p B)_\gamma &= \{(\delta, b) \mid \delta \in \gamma^*\{\gamma\} \wedge b \in B_\delta\} \\ &\cong \{(a, b) \mid a \in A_\gamma \wedge b \in B_{(\gamma,a)}\}.\end{aligned}$$

Exercise 5.1.4 Explicitly write down the isomorphisms appearing in the above calculations of $\Pi_p B$ and $\Sigma_p B$. Does it matter which of the two isomorphic versions of products and sums we use?

5.2 Families of assemblies

The interpretation of type theory in assemblies proceeds much as in sets, we only need to make sure that the set-theoretic maps are realized. However, we must first define what a family of assemblies is.

Defining a family of assemblies to be a collection of assemblies indexed by (the underlying set of) an assembly almost works, we just have to additionally require that all the assemblies in the family share the same underlying type.[5]

5: This is so because tpcas are simply typed. By making all the assemblies in the family share the same type, we facilitate the construction of its product, whose underlying type may then be the (non-dependent) function type, see **??**.

Definition 5.2.1 A ***(uniform) family of assemblies*** $S : I \to \mathsf{Asm}(\mathbb{A}, \mathbb{A}')$ is given by an ***index assembly*** I, an ***underlying type*** $\|S\|$, and for each $i \in |I|$ an assembly S_i such that $\|S_i\| = \|S\|$.

The qualifier "uniform" refers to the fact that all the members share the same underlying type. We drop it because we only ever consider uniform families. We write $\mathsf{Fam}_{\mathbb{A},\mathbb{A}'}(I)$ or just $\mathsf{Fam}(I)$ for the collection of all families of assemblies indexed by I. For everything to work out, maps between families of assemblies have to be uniformly realized.

Definition 5.2.2 A ***(uniform) map of families*** $f : A \to B$ between families of assemblies $A, B \in \mathsf{Fam}(I)$ is given by a family of maps $(f_i : |A_i| \to |B_i|)_{i \in |I|}$ for which there exists $\mathtt{f} \in \mathbb{A}'_{|I| \to |A| \to |B|}$ such that, for all i, $\mathtt{i}$, x, $\mathtt{x}$,

$$\mathtt{i} \Vdash_I i \wedge \mathtt{x} \Vdash_{A_i} x \implies \mathtt{f}\,\mathtt{i}\,\mathtt{x} \Vdash_{B_i} f x.$$

The definition endows each $\mathsf{Fam}(I)$ with the structure of a category. An assembly map $r : J \to I$ induces a ***reindexing*** $r^* : \mathsf{Fam}(I) \to \mathsf{Fam}(J)$, defined by

$$r^* A = A \circ r.$$

There is nothing here to be realized, but we shall use realizers for r in the construction of products and sums.

Exercise 5.2.1 Verify that reindexing is contravariantly functorial.

5.2.1 Products and sums of families of assemblies

Consider a family of assemblies $S \in \mathsf{Fam}(J)$. It has an associated family of underling sets $|S| \in \mathsf{Fam}(J)$, defined by $j \mapsto |S_j|$. Given an assembly map $r : J \to I$, $i \in |I|$, and $u \in (\Pi_r |S|)_i$, say that u is ***realized*** by $\mathtt{u} \in \mathbb{A}_{\|J\| \to \|S\|}$ when, for all $j \in r^*\{i\}$ and $\mathtt{j} \in \mathbb{A}_{\|J\|}$,

$$\mathtt{j} \Vdash_J j \implies \mathtt{u}\,\mathtt{j} \Vdash_{S_i} u j.$$

When this is the case, we write $\mathtt{u} \Vdash_{(\Pi_r S)_i} u$. Now define the ***product*** $\Pi_r S \in \mathsf{Fam}(I)$ to be the family whose realizability relation at $i \in |I|$ is $\Vdash_{(\Pi_r S)i}$ and

$$\|\Pi_r S\| = \|J\| \to \|S\|,$$
$$|(\Pi_r S)_i| = \{u \in (\Pi_r |S|)_i \mid \exists \mathtt{u} \in \mathbb{A}_{\|J\| \to \|S\|} .\, \mathtt{u} \Vdash_{(\Pi_r S)_i} u\}.$$

Define the ***sum*** $\Sigma_r S \in \mathsf{Fam}(J)$ to be the family

$$\|\Sigma_r S\| = \|J\| \times \|S\|,$$
$$|(\Sigma_r S)_i| = (\Sigma_r |S|)_i,$$
$$\mathtt{r} \Vdash_{(\Sigma_r S)_i} (j, x) \Leftrightarrow \mathtt{fst}\,\mathtt{r} \Vdash_J j \wedge \mathtt{snd}\,\mathtt{r} \Vdash_{S_j} x.$$

Notice how at the level of underlying types the dependency on the parameter disappears because we required the families to be uniform.

Exercise 5.2.2 Verify the adjunctions

$$\Sigma_r \dashv r^* \dashv \Pi_r.$$

Half of work has been done in Subsection 5.1.1. You still need to check that the map of families $\hat{f}$ defined in (5.2) is realized when f is realized, and similarly for $\check{g}$ defined in (5.3).

Products and sums along arbitrary reindexings are perhaps a bit un-intuitive. For better understanding we spell out the non-parameterize sum and products. Given a family $S \in \mathsf{Fam}(I)$, its ***product*** $\Pi_I S$ is the assembly

$$\|\Pi_I S\| = \|I\| \to \|S\|,$$
$$|\Pi_I S| = \{u \in \Pi_{|I|} |S| \mid \exists \mathtt{u} \in \mathbb{A}_{\|I\| \to \|S\|} .\, \mathtt{u} \Vdash_{\Pi_I S} u\},$$
$$\mathtt{u} \Vdash_{\Pi_I S} u \Leftrightarrow \forall i \in |I|, \mathtt{i} \in \mathbb{A}_{\|I\|} .\, \mathtt{i} \Vdash_I i \Rightarrow \mathtt{u}\,\mathtt{i} \Vdash_{S_i} u i.$$

The ***sum*** $\Sigma_I S$ is the assembly

$$\|\Sigma_I S\| = \|I\| \times \|S\|,$$
$$|\Sigma_I S| = \Sigma_{|I|} |S|,$$
$$\mathtt{r} \Vdash_{\Sigma_I S} (i, x) \Leftrightarrow \mathtt{fst}\,\mathtt{r} \Vdash_I i \wedge \mathtt{snd}\,\mathtt{r} \Vdash_{S_i} x.$$

These are indeed just the familiar set-theoretic constructions embellished with realizers.

5.2.2 Contexts of assemblies

Contexts of assemblies are built as iterated sums, the same was as contexts of sets. Thus a telescope of assemblies

$$\Gamma = (x_1 : S_1, \ldots, x_n : S_n)$$

is the assembly

$$\begin{aligned} \|\Gamma\| &= \|S_1\| \times \cdots \times \|S_n\|, \\ |\Gamma| &= \{(a_1, \ldots, a_n) \mid \forall i \leq n.\, a_i \in |(S_i)_{(a_1,\ldots,a_{i-1})}|\} \\ \mathtt{r} \Vdash_\Gamma (a_1, \ldots, a_n) &\Leftrightarrow \forall i \leq n.\, \mathtt{proj}_{n,i}\, \mathtt{r} \Vdash_{(S_i)_{(a_1,\ldots,a_{i-1})}} a_i, \end{aligned}$$

where $\mathtt{proj}_{n,i}$ is the i-th projection from an n-tuple.

5.3 Propositions as assemblies

In Example 5.1.1 we placed a hypothesis into the context, but for this to make sense in needs to be a type family. How can a proposition be a type?

In classical logic a predicate ϕ on a set A is a Boolean function $\phi : S \to \{\bot, \top\}$. If we encode the truth values $\bot$ and $\top$ with sets, ϕ becomes a family of sets, which is what we want. A choice that works well is to take $\bot$ to be $\emptyset$ and $\top$ to be $\{\star\}$. An even better idea is to allow any singleton set to represent truth, they are all isomorphic anyhow. We may do the same with assemblies.

Definition 5.3.1 An assembly is a ***proposition*** if all of its elements are equal.[6] A ***predicate*** on an assembly S is a family of assemblies $P \in \mathsf{Fam}(S)$ such that Px is a predicate for all $x \in |S|$.

6: Even though "all elements are equal" is equivalent to "empty or singleton", the choice of wording matters once we bring in realizability logic, where excluded middle is not valid.

To see that the definition works, note that the following logical operations can be expressed with standard constructions on assemblies:

$$\begin{aligned} \bot &= \mathbb{0}, \\ \top &= \mathbb{1}, \\ P \wedge Q &= P \times Q, \\ P \Rightarrow Q &= P \to Q,, \\ \forall_S R &= \Pi_S R. \end{aligned} \tag{5.4}$$

For instance, $P \times Q$ is inhabited if, and only if, both P and Q are, which is precisely how conjunction is supposed to be. Similarly, the product $\Pi_S R$ has a choice map only if[7] all the assemblies Rx are inhabited, which again characterizes the universal quantifier.

7: We did *not* say "if and only if" because in assemblies choice maps must also be realized. It is quite possible that there is no realized choice map for a family of assemblies, even though every one of its member is inhabited. Even so, $\Pi_S R$ satisfies the rules of universal quantification, and so can be used as such.

Disjunction and the existential quantifier present a bit of a challenge. How about

$$P \vee Q = P + Q \qquad \text{and} \qquad \exists_S R = \Sigma_S R\,?$$

The definitions seem to work: $P + Q$ is inhabited if, and only if, P or Q is inhabited; and $\Sigma_S R$ is inhabited if, and only if, there is $x \in |S|$ for which Px has an element.[8] The problem is that $P + Q$ and $\Sigma_S R$ need not be propositions because they may have more than one element. There are two ways to fix the deficiency.

8: We are a bit sloppy with these statements because we should also think about realizers.

Firstly, we can force any assembly P to become a proposition $\|P\|$ by quotienting it with respect to the trivial equivalence relation. Disjunction and existential quantification are then defined as

$$\begin{aligned} P \vee Q &= \|P \vee Q\|\,, \\ \exists_S R &= \|\Sigma_S R\|\,. \end{aligned} \tag{5.5}$$

This option is explored in Subsection 5.3.1.

Secondly, why do we insist that propositions must have at most one element? We could use *all* assemblies as if they were propositions, with the empty assembly representing falsehood and the inhabited assemblies truth. The correspondence (5.4) still holds and can even be extended to

$$\begin{aligned} P \vee Q &= P + Q, \\ \exists_S R &= \Sigma_S R. \end{aligned}$$

This approach is taken in Martin-Löf type theory and goes by the name ***propositions as types***.

Which definition of $\vee$ and $\exists$ should we adopt? Mathematical practice uses both. The truncated sum $\|\Sigma_S R\|$ is a form of *abstract existence* because its element, when it exists, reveals no specific $x \in |S|$ for which Rx is inhabited; whereas $\Sigma_S R$ is *concrete existence* because its elements provide witnesses. Similarly, $\|P + Q\|$ states that one or the other disjunct holds without revealing which one, whereas an element of $P + Q$ makes a specific choice of one or the other.

Judicious use of propositional truncation thus makes it possible to formalize aspects of mathematical practice that are not easily incorporated into traditional first-order logic, which provides only abstract existence and disjunction, nor into traditional Martin-Löf type theory, which provides only concrete existence and disjunction.

5.3.1 Propositional truncation of an assembly

Propositional truncation works both for sets and assemblies. The propositional truncation $\|A\|$ of a set A is the quotient $A/{\sim}$ by the full relation $\sim$ on A. The quotient map takes an element $x \in A$ to its equivalence class $[x] = A$. Thus, if A has an element then $\|A\| = \{A\}$ and if A is empty then $\|A\| = \emptyset$. A category theorist would postulate $\|A\|$ as the coequalizer

$$A \times A \underset{\pi_2}{\overset{\pi_1}{\rightrightarrows}} A \xrightarrow{[-]} \|A\|$$

This construction works in categories other than sets. It takes an assembly S to its ***propositional truncation*** $\|S\|$ where[9]

$$\begin{aligned} \| \|S\| \| &= \|S\|, \\ | \|S\| | &= \||S|\| , \\ \mathtt{r} \Vdash_{\|S\|} \xi &\Leftrightarrow \exists y \in |S|.\, \mathtt{r} \Vdash_S y. \end{aligned}$$

9: It is too late to disentangle the conflicting notations for propositional truncation and the underlying type. We could insert parentheses, $\|(\|S\|)\| = \|S\|$ and $|(\|S\|)| = \|(|S|)\|$, but that just ruins the fun.

Like any worthy construction, propositional truncation has a universal property stemming from the above coequalizer diagram: if P is a proposition then for every assembly map $f : S \to P$ there is a unique $\bar{f} : \|S\| \to P$ such that the following diagram commutes:

$$\begin{array}{ccc} S & \xrightarrow{|-|} & \|S\| \\ & \searrow^{f} & \downarrow \bar{f} \\ & & P \end{array}$$

Exercise 5.3.1 Verify that the universal property of propositional truncation holds with respect to families of assemblies. Given a family of assemblies $S \in \mathsf{Fam}(\Gamma)$, a predicate $P \in \mathsf{Fam}(\Gamma)$ and a map of families $f : S \to P$, there is a unique map of families $\bar{f} : \|S\| \to P$ such that $\bar{f} \circ |-| = f$.

Moreover, explain what it means for propositional truncation to be functorial with respect to reindexing, then show that it is.

5.3.2 Realizability predicates and propositions

In Chapter 4 we gave a realizability interpretation of logic, with its own notions of propositions and predicates. We show that it is equivalent to truncated logic of the present chapter.

Let $\mathsf{Pred}'(S)$ be the class of predicates on S in the sense of the present chapter, i.e., families of assemblies $P \in \mathsf{Fam}(S)$ such that $\forall u, v \in |Px|.\, u = v$ for all $x \in |S|$. We endow $\mathsf{Pred}'(S)$ with a preorder $\vdash'$, defined by

$$P \vdash' Q \iff \text{there is a map of families } P \to Q.$$

In fact, there is at most one map of families $P \to Q$.

Exercise 5.3.2 Verify that the preoder $\mathsf{Pred}'(S)$ forms a Heyting prealgebra with the operations given by (5.4) and (5.5).

Theorem 5.3.1 *The Heyting prealgebras* $(\mathsf{Pred}(S), \vdash)$ *and* $(\mathsf{Pred}'(S), \vdash')$ *are equivalent.*

Proof. The equivalence takes a realizability predicate $p \in \mathsf{Pred}(S)$ to the

family $P_p \in \mathsf{Fam}(S)$, defined by

$$\|P_p\| = \|p\|,$$
$$|P_p x| = 1,$$
$$\mathtt{r} \Vdash_{P_p x} \star \Leftrightarrow \mathtt{r} \Vdash px.$$

In the opposite direction, a predicate $P \in \mathsf{Pred}'(S)$ is taken to the realizability predicate $p_P \in \mathsf{Pred}(S)$, defined by

$$\|p_P\| = \|P\|,$$
$$\mathtt{r} \Vdash p_P x \Leftrightarrow \exists \xi \in Px.\, \mathtt{r} \Vdash_{Px} \xi.$$

To finish the proof, one would have to verify that $p \to P_p$ and $P \to p_P$ are monotone, that $p \dashv\vdash p_{P_p}$ and $P \dashv\vdash' P_{p_P}$, and that they preserve the Heyting prealgebra structure. □

Exercise 5.3.3 In the proof of Theorem 5.3.1 the passage from P to realizability predicate p_P works for an arbitrary family $P \in \mathsf{Fam}(S)$. Therefore, the equivalence extends to a pair of functors

$$\mathsf{Pred}(S) \underset{T}{\overset{I}{\rightleftarrows}} \mathsf{Fam}(S)$$

Show that I is full and faithful, T is its left adjoint, and that $I \circ T$ is naturally isomorphic to propositional truncation. How does the adjunction interact with reindexing?

The realizability logic in Chapter 4 gave the simple definition of quantifiers where one of the parameters of a two-parameter predicate was quantified over. We can improve on that by defining quantification with respect to reindexing.

An assembly map $r : J \to I$ induces a monotone map $r^* : \mathsf{Pred}(J) \to \mathsf{Pred}(I)$ which acts by precomposition, $r^* p = p \circ r$. The ***universal quantification*** of $p \in \mathsf{Pred}(J)$ ***along*** r is the realizability predicate $\forall_r p \in \mathsf{Pred}(I)$, defined by

$$\|\forall_r p\| = \|J\| \to \|p\|,$$
$$\mathtt{u} \Vdash (\forall_r p) i \Leftrightarrow \forall j \in r^*\{i\}.\, \forall \mathtt{j} \in \|J\|.\, \mathtt{j} \Vdash_J j \Rightarrow \mathtt{u}\,\mathtt{j} \Vdash pj.$$

Define the ***existential quantification along*** r to be the realizability predicate $\exists_r p \in \mathsf{Pred}(I)$,

$$\|\exists_r p\| = \|J\| \times \|p\|,$$
$$\mathtt{r} \Vdash (\exists_r p) i \Leftrightarrow \exists j \in |J|.\, \mathtt{fst}\,\mathtt{r} \Vdash_J j \wedge \mathtt{snd}\,\mathtt{r} \Vdash pj.$$

The quantifiers from Section 4.4 arise as quantification along a projection $T \times S \to S$.

Exercise 5.3.4 Verify that the equivalence of $\mathsf{Pred}(S)$ and $\mathsf{Pred}'(s)$ preserves the quantifiers as well.

Henceforth we shall use one or the other formulation of realizability

predicates, whichever is more appropriate in the situation at hand.

5.4 Identity types

In Section 4.6 we defined equality on an assembly S as as predicate on $S \times S$. The corresponding family of assemblies $\mathsf{Id}_S \in \mathsf{Fam}(S \times S)$ is given by

$$\|\mathsf{Id}_S\| = \mathtt{unit} \qquad \text{and} \qquad \mathsf{Id}_S(x, y) = \begin{cases} \mathbb{1} & \text{if } x = y, \\ \mathbb{0} & \text{if } x \neq y. \end{cases}$$

It is called the ***identity type*** of S.

If we compose Id_S with the diagonal map $S \to S \times S$ we get the family $x \mapsto \mathsf{Id}_S(x, x)$ which has an element refl_S, namely $\mathsf{refl}_S(x) = \star$, realized by $\langle x^{\|S\|} \rangle\, \star$.

The identity type in Martin-Löf type theory satisfies the following elimination principle. Suppose Γ is a context and $\Gamma \vdash S\ \mathsf{type}$ a type over it.

5.4.1 UIP and equality reflection

5.5 Inductive and coinductive types

5.6 Universes

5.6.1 Universes of propositions

The universe of decidable propositions.

The universe of semi-decidable propositions.

The universe of stable propositions.

The universe of propositions.

5.6.2 The universe of modest sets

5.6.3 The universe of small assemblies

6 The internal language at work

6.1 Epis and monos

6.2 The axiom of choice

6.3 Heyting arithmetic

6.4 Countable objects

6.5 Markov's principle

6.6 Church's thesis and the computability modality

6.7 Aczel's presentation axiom

6.8 Continuity principles

6.8.1 Brouwer's continuity

6.8.2 Kreisel-Lacombe-Schönfield-Ceitin continuity

6.9 Brouwer's compactness principle

Bibliography

[1] A. Bauer, L. Birkedal, and D.S. Scott. "Equilogical Spaces". Submitted for publication. 1998 (cited on page 54).

[2] Andrej Bauer. "The Realizability Approach to Computable Analysis and Topology". PhD thesis. Carnegie Mellon University, 2000 (cited on page 44).

[3] Andrej Bauer. "A Relationship between Equilogical Spaces and Type Two Effectivity". In: *Mathematical Logic Quarterly* 48.S1 (2002), pp. 1–15 (cited on page 62).

[4] Andrej Bauer and Jens Blanck. "Canonical Effective Subalgebras of Classical Algebras as Constructive Metric Completions". In: *Journal of Universal Computer Science* 16.18 (2010), pp. 2496–2522 (cited on page 44).

[5] Jens Blanck. "Domain representability of metric spaces". In: *Annals of Pure and Applied Logic* 83 (1997), pp. 225–247 (cited on page 44).

[6] Vasco Brattka and Peter Hertling, eds. *Handbook of Computability and Complexity in Analysis*. Theory and Applications of Computability. Springer, 2021 (cited on page 60).

[7] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. Vol. 97. London Mathematical Society Lecture Notes Series. Cambridge University Press, 1987 (cited on page 54).

[8] Alonzo Church. "A set of postulates for the foundation of logic". In: *Annals of Mathematics, Series 2* 33 (1932), pp. 346–366 (cited on page 21).

[9] Alonzo Church and J. B. Rosser. "Some Properties of Conversion". In: *Transactions of the American Mathematical Society* 39.3 (1936). Available at http://www.jstor.org/pss/1989762, pp. 472–482 (cited on page 23).

[10] M. Davis. *Computability and Unsolvability*. Reprinted in 1982 by Dover Publications. McGraw-Hill, 1958 (cited on pages 4, 8).

[11] Yuri L. Eršov. "Handbook of Computability Theory". In: vol. 140. Studies in Logic and the Foundations of Mathematics. Springer, 1999. Chap. Theory of numberings, pp. 473–503 (cited on page 54).

[12] K. Gödel. "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes". In: *Dialectica* 12 (1958), pp. 280–287 (cited on page 35).

[13] H. Goldstein, J. von Neumann, and A. Burks. *Report on the mathematical and logical aspects of an electronic computing instrument*. Tech. rep. Princeton Institute of advanced study, 1947 (cited on page 5).

[14] C.A. Gunter and D.S. Scott. "Semantic Domains". In: *Handbook of Theoretical Computer Science*. Ed. by J. van Leeuwen. Elsevier Science Publisher, 1990 (cited on page 26).

[15] J. D. Hamkins and A. Lewis. "Infinite time Turing machines". In: *Journal of Symbolic Logic* 65.2 (2000), pp. 567–604 (cited on page 15).

[16] Jr. H.R. Rogers. *Theory of Recursive Functions and Effective Computability*. 3rd. MIT Press, 1992 (cited on page 4).

[17] S.C. Kleene. "On the Interpretation of Intuitionistic Number Theory". In: *Journal of Symbolic Logic* 10 (1945), pp. 109–124 (cited on page 44).

[18] S.C. Kleene and R. Vesli. *The foundations of intuitionistic mathematics. Especially in relation to recursive functions.* Studies in Logic and The Foundations of Mathematics. Amsterdam: North-Holland, 1965 (cited on page 60).

[19] Steephen Kleene. "Recursive predicates and quantifiers". In: *Transactions of the AMS* 53.1 (1943), pp. 41–73 (cited on page 7).

[20] B. A. Kušner. *Lectures on Constructive Mathematical Analysis*. Vol. 60. Translations of Mathematical Monographs. American Mathematical Society, 1984 (cited on page 54).

[21] Peter Lietz. "From Constructive Mathematics to Computable Analysis via the Realizability Interpretation". PhD thesis. Technischen Universität Darmstadt, 2004 (cited on page 42).

[22] J. Longley. "Realizability Toposes and Language Semantics". PhD thesis. Edinburgh University, 1995 (cited on page 28).

[23] John Longley. "Realizability Toposes and Language Semantics". PhD thesis. University of Edinburgh, 1994 (cited on pages 37, 39–41, 52–54).

[24] John Longley. "Matching typed and untyped realizability". In: *Electronic Notes in Theoretical Computer Science* 23.1 (1999). https://doi.org/10.1016/S1571-0661(04)00105-7, pp. 74–100 (cited on pages 4, 37).

[25] John Longley. "Unifying typed and untyped realizability". Available at http://homepages.inf.ed.ac.uk/jrl/Research/unifying.txt. 1999 (cited on pages 4, 31).

[26] John Longley. "Computability structures, simulations and realizability". In: *Mathematical Structures in Computer Science* 24.1 (2014). https://doi.org/10.1017/S0960129513000182 (cited on page 4).

[27] Matías Menni and Alex K. Simpson. "Topological and Limit-Space Subcategories of Countably-Based Equilogical Spaces". In: *Mathematical Structures in Computer Science* 12.6 (2002), pp. 739–770 (cited on page 56).

[28] P. Odifreddi. *Classical Recursion Theory*. Vol. 125. Studies in logic and the foundations of mathematics. North-Holland, 1989 (cited on page 4).

[29] Jaap van Oosten. *Realizability: An Introduction To Its Categorical Side*. Vol. 152. Studies in logic and the foundations of mathematics. Elsevier, 2008 (cited on page 31).

[30] G. Plotkin. "$\mathbb{T}^\omega$ as a Universal Domain". In: *Journal of Computer and System Sciences* 17 (1978), pp. 209–236 (cited on page 26).

[31] Gordon Plotkin. "LCF Considered as a Programming Language". In: *Theoretical Computer Science* 5 (1977). Available at http://homepages.inf.ed.ac.uk/gdp/publications/LCF.pdf, pp. 223–255 (cited on page 35).

[32] Matthias Schröder. "Handbook of Computability and Complexity in Analysis". In: Springer, 2021. Chap. Admissibly represented spaces and QCB-spaces, pp. 305–346 (cited on page 61).

[33] Dana S. Scott. "Continuous Lattices". In: *Lecture Notes in Mathematics 274*. Springer, 1972, pp. 97–136 (cited on page 25).

[34] D.S. Scott. "Data Types as Lattices". In: *SIAM Journal of Computing* 5.3 (1976), pp. 522–587 (cited on page 18).

[35] D. Spreen. "On Effective Topological Spaces". In: *The Journal of Symbolic Logic* 63.1 (1998), pp. 185–221 (cited on page 54).

[36] J.V. Tucker and J.I. Zucker. "Computable Functions and Semicomputable Sets on Many-Sorted Algebras". In: *Handbook of Logic in Computer Science, Volume 5*. Ed. by S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum. Oxford: Clarendon Press, 2000 (cited on page 44).

[37] Alan Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 2.42 (1937). Available at http://www.scribd.com/doc/2937039/Alan-M-Turing-On-Computable-Numbers, pp. 230–265 (cited on pages 4, 5).

[38] Klaus Weihrauch. *Computable Analysis*. Berlin: Springer, 2000 (cited on pages 44, 60).

[39] N. Šanin. *Constructive Real Numbers and Constructive Function Spaces*. Vol. 21. Translations of Mathematical Monographs. American Mathematical Society, 1968 (cited on page 54).