Computation Theory

Andrew Pitts, Tobias Kohn
University of Cambridge, UK

© Andrew Pitts, Tobias Kohn
University of Cambridge, UK

Version of 28th April 2022

Contents

1 Introduction **9**

1.1 Algorithmically undecidable problems . . . 9

1.1.1 Hilbert's Entscheidungsproblem . . . 10

1.1.2 Decision problems . . . 10

1.2 From the Halting Problem to the *Entscheidungsproblem* . . . 12

1.3 An Applied Example of the Halting Problem* . . . 13

1.4 Provability and Truth* . . . 15

2 Register Machines **17**

2.1 Register Machines, Informally . . . 17

2.2 Graphical Representation . . . 19

2.3 Partial Functions . . . 20

2.3.1 Computable Functions . . . 21

3 Coding Programs as Numbers **23**

3.1 Numerical Coding of Pairs . . . 23

3.2 Numerical Coding of Lists . . . 24

3.3 Numerical Coding of Programs . . . 25

4 Universal Register Machine **27**

4.1 High-level specification . . . 27

4.2 Overall Structure of U's Program . . . 28

4.3 The Program for U . . . 30

5 The Halting Problem **33**

5.1 Computable Functions . . . 34

5.1.1 Enumerating Computable Functions . . . 34

5.1.2 An Uncomputable Function . . . 35

5.1.3 (Un)decidable Sets of Numbers . . . 35

6 Turing Machines **37**

6.1 Turing Machines . . . 37

6.2 Turing Machine Computation . . . 39

6.3 Computable Functions . . . 42

7 Notions of Computability **43**

7.1 Computable Partial Functions . . . 43

8 Partial Recursive Functions **47**

8.1 Primitive Recursion . . . 48

8.2 Minimisation . . . 50

8.3 Partial Recursive Functions . . . 51

8.4 Ackermann's Function . . . 53

9 Lambda-Calculus **55**

9.1 Notions of Computability . . . 55

9.2 Introduction . . . 56

9.3 λ-Terms . . . 58

9.4 α-Equivalence . . . 60

9.5 β-Reduction . . . 61

9.5.1 Substitution . . . 61

9.5.2 Reduction . . . 61

9.5.3 β-Conversion $M =_\beta N$ . . . 62

9.6 β-Normal Forms . . . 64

9.7 Applications* . . . 65

9.7.1 Compiler Transformations . . . 65

9.7.2 Macros . . . 66

10 Lambda-Definable Functions **67**

10.1 Encoding Data in λ-Calculus 67

10.2 λ-Definable Functions 69

10.2.1 Representing Basic Functions 70

10.2.2 Representing Composition 70

10.2.3 Representing Predecessor 71

10.3 Primitive Recursion 71

10.3.1 Curry's Fixed Point Combinator $\mathbf{Y}$ 72

10.3.2 Representing Minimization 74

10.4 λ-Definability 75

Preface

These notes are designed to accompany twelve lectures on computation theory for Part IB of the Computer Science Tripos at the University of Cambridge. The aim of this course is to introduce several apparently different formalisations of the informal notion of *algorithm*, to show that they are equivalent, and to use them to demonstrate that there are uncomputable functions and algorithmically undecidable problems.

At the end of the course you should:

- be familiar with the register machine, Turing machine and λ-calculus models of computability;
- understand the notion of coding programs as data, and of a universal machine;
- be able to use diagonalisation to prove the undecidability of the Halting Problem;
- understand the mathematical notion of partial recursive function and its relationship to computability.

The prerequisite for taking this course is the Part IA course Discrete Mathematics. This incaration of the Computation Theory course builds on previous lecture notes by Ken Moody, Glynn Winskel, Larry Paulson and Andrew Pitts. It contains some material that *everyone who calls themselves a computer scientist should know*. It is also a prerequisite for the Part IB course on Complexity Theory.

An exercise sheet, any corrections to the notes and additional material can be found on the course web page: `https://www.cl.cam.ac.uk/teaching/2122/CompTheory/`.

1 Introduction

Where we set the (historic) stage and introduce one of the main protagonists: the halting problem. *You should leave with a good intuitive understanding of the halting problem and decision problems in general.*

1.1 Algorithmically undecidable problems

Computers cannot solve all mathematical problems, even if they are given unlimited time and working space. Three famous examples of computationally unsolvable problems are sketched in this lecture.

- Hilbert's Entscheidungsproblem
- The Halting Problem
- Hilbert's 10th Problem

Historic background. Around 300 BC, Euclid of Alexandria not only collected the mathematical knowledge and wisdom of his time, but set out to formally prove the known theorems. He established a set of basic definitions and basic "truths" (the *axioms*), which lay the groundwork for proving all other propositions and theorems. For over two thousand years, the rigor of geometric proofs was virtually unmatched by anything else.

In the 19th century, the development of set theory and modern logic led to a revolution. For the first time it was possible to precisely define what a number is and thus start to establish axioms and proofs as rigorous as in geometry. Similar to Euclid's programme of collecting and proving the mathematical wisdom of its time, the mathematicians of the 19th century started on a quest to systematically prove all mathematical theorems by strictly adhering to the established ground rules of logic. With his programme of open problems and questions, *David Hilbert* was on the forefront of this: in the spirit of this "Euclidean" programme, he formulated a set of most pressing open questions to answer.

The later discovery that this programme of systematically collecting and proving all mathematical theorems is impossible was a severe blow to the mathematical community. However, it gave very strong impetus to the development of new areas of research, with computer science being one of the most prominent ones.

1.1.1 Hilbert's Entscheidungsproblem

Hilbert's Entscheidungsproblem

Is there an algorithm which when fed any statement in the formal language of first-order arithmetic, determines in a finite number of steps whether or not the statement is provable from Peano's axioms for arithmetic, using the usual rules of first-order logic?

Such an algorithm would be useful. For example, by running it on

$$\forall k > 1 : \exists p, q : (2k = p + q \wedge prime(p) \wedge prime(q))$$

(where $prime(p)$ is a suitable arithmetic statement that p is a prime number) we could solve *Goldbach's Conjecture* ("every even integer strictly greater than two is the sum of two primes"), a famous open problem in number theory.

We have been searching for a counterexample to Goldbach's conjecture for years. Computer programs checked every possibility in a wide range of integers to see whether there is one even integer that could not be written as the sum of two prime numbers. If one such example was found, we would know that there is no proof for the conjecture. On the other hand, if the conjecture was true, checking numbers would never yield a definitive answer. In a way, what we really want to know is: does the program that looks for a counterexample to Goldbach's conjecture ever *halt*, or does it keep on running forever because there is no such counterexample?

The Entscheidungsproblem was posed by Hilbert at the 1928 International Congress of Mathematicians. The problem was actually stated in a more ambitious form, with a more powerful formal system in place of first-order logic. In 1928, Hilbert believed that such an algorithm could be found. A few years later (in 1935/36) he was proved wrong by the work of Church and Turing, as we will see.

Note that the algorithm in the Entscheidungsproblem does not need to actually yield a proof if the statement is provable, but only needs to tell whether the statement is provable or not. Compare this to, e.g., the famous example of the *Four Colour Theorem*, which was proven in the 1970s using a computer. Although this computer assisted proof showed that the theorem is provable, many mathematicians did not accept it as a valid *mathematical* proof.

1.1.2 Decision problems

Entscheidungsproblem means "decision problem". Given

- a set S whose elements are finite data structures of some kind,
- a property P of elements of S (i.e. a function $P : S \to \{\bot, \top\}$),

the associated **decision problem** is to determine whether an element $s \in S$ has property P or not:

Decision problem

Find an **algorithm** which terminates with result 0 or 1 when fed an element $s \in S$ and yields result 1 when fed s if and only if s has property P.

At the time Hilbert posed the *Entscheidungsproblem*, there was no precise definition of “algorithm”, just examples, such as:

- procedure for multiplying numbers in decimal place notation;
- procedure for extracting square roots to any desired accuracy;
- Euclid’s algorithm for finding highest common factors.

These examples share some common features:

- a **finite** description of the procedure in terms of elementary operations,
- it is **deterministic** (the next step is uniquely determined if there is one),
- the procedure may not terminate on some input data, but we can recognise when it does terminate and what the **result** is.

In 1935/36 Turing in Cambridge and Church in Princeton independently gave negative solutions to Hilbert’s *Entscheidungsproblem*. The first step was to give a precise, mathematical definition of “algorithm”. Turing’s definition used *Turing Machines*, Church’s definition was based on *Lambda-Calculus*. Then one can regard *algorithms as data* on which algorithms can act and reduce the problem to the *Halting Problem*.

Halting Problem

The **Halting Problem** is a decision problem where

- the set S consists of all pairs (A, D), where A is an algorithm and D is a datum on which it is designed to operate;
- the propery P holds for (A, D) if the algorithm A, when applied to datum D, eventually produces a result (that is, eventually **halts** – we write $A(D) \downarrow$ to indicate this).

Turing and Church’s work shows that *the Halting Problem is undecidable*, that is, there is no algorithm H such that for all $(A, D) \in S$:

$$H(A, D) = \begin{cases} 1 & \text{if } A(D) \downarrow \\ 0 & \text{otherwise.} \end{cases}$$

For instance, you cannot build a compiler that detects infinite loops in all programs. There is at least one program for which the compiler could not decide whether it will loop forever or not.

Informal proof by contradiction. If there where such an H, let C be the algorithm:

> "given input A, compute $H(A, A)$; if $H(A, A) = 0$ then return 1, else loop forever."

So:

$$\begin{aligned} \forall A : (C(A) \downarrow \leftrightarrow\ H(A, A) = 0) & \qquad \text{since } H \text{ is total} \\ \forall A : (H(A, A) = 0\ \leftrightarrow\ \neg A(A) \downarrow) & \qquad \text{definition of } H \\ \forall A : (C(A) \downarrow \leftrightarrow\ \neg A(A) \downarrow) & \end{aligned}$$

Taking A to be C, we get $C(C) \downarrow \leftrightarrow \neg C(C) \downarrow$, which is a contradiction! □

1.2 From the Halting Problem to the *Entscheidungsproblem*

The final step in Turing/Church's proof of undecidability of the *Entscheidungsproblem* is that they constructed an algorithm encoding instances (A, D) of the Halting Problem as arithmetic statements $\Phi_{A,D}$ with the property:

$$\Phi_{A,D} \text{ is provable } \leftrightarrow A(D) \downarrow$$

Thus any algorithm deciding provability of arithmetic statements could be used to decide the Halting Problem—so no such algorithm exists.

With hindsight, a positive solution of the *Entscheidungsproblem* would be too good to be true. However, the algorithmic unsolvability of some decision problems is much more surprising. A famous example is *Hilbert's 10th Problem*.

Hilbert's 10th Problem

Give an algorithm which, when started with any *Diophantine equation*, determines in a finite number of operations whether or not there are natural numbers satisfying the equation.

This is one of a number of important open problems listed by Hilbert at the International Congress of Mathematicians in 1900.

A Diophantine equation is of the form:

$$p(x_1, x_2, \ldots, x_n) = 0$$

where p is a polynomial in unknowns $x_1, \ldots, x_n$ with coefficients from $\mathbb{N} = \{0, 1, 2, \ldots\}$. These equations are named after Diophantus of Alexandria (c. 250 AD). An example is problem 7 from his third book: "*find three whole numbers x_1, x_2 and x_3 such that the product of any two added to the third is a square*." In modern notation: find $x_1, x_2, x_3 \in \mathbb{N}$ for which there exists $x, y, z \in \mathbb{N}$ with:

$$\left(x_1x_2 + x_3 - x^2\right)^2 + \left(x_2x_3 + x_1 - y^2\right)^2 + \left(x_3x_1 + x_2 - z^2\right)^2 = 0$$

Note that we actually want all of the three terms $x_i x_j + x_k - x^2$ to be zero at the same time, i.e. $A = 0 \wedge B = 0 \wedge C = 0$. By first squaring the three terms and then adding them up, none of the A^2, B^2, and C^2 could ne negative, and hence the overall sum is only zero if each of A, B, and C is zero.

Hilbert's 10$^{\text{th}}$ Problem was posed in 1900, but only solved in 1970: Y. Matijasevič, J. Robinson, M. Davis and H. Putnam showed it is *undecidable* by reducing it to the Halting Problem. Their orginal proof used Turing machines. A later, simpler proof used Minsky and Lambek's *register machines* – we will use them in this course to begin with and return to Turing and Church's formulations of the notion of "algorithm" later.

1.3 An Applied Example of the Halting Problem*

If you are a programmer, you might be more comfortable expressing algorithms in the form of program code. Then the following explanation might help you understand the argumentation behind the *Halting Problem*. However, if you are more interested in the mathematical formalism, you can savely skip this section.

You have probably already written programs which did not terminate because of some sort of infinite loop. Would it not be nice if the compiler somehow warned you beforehand that your program might run forever and never properly terminates? As it turns out, it is impossible to write a program that could decide for all possible programs whether they halt in finite time or not. Let us consider this with a concrete example.

The Collatz conjecture. The Collatz conjecture is based on the following simple algorithm: pick any natural number n to start with. If n is even, divide it by two, if it is odd, multiply it by three and add one. Python and ML functions implementing this algorithm are given below. In order to avoid later confusion with the term *function* in other context, we will just refer to this as a *program* – for our purposes it does not matter if you implement it as a function inside a program or if you think of a program that can be run with command line arguments.

```python
def collatz(n):
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3*n + 1
```

```sml
fun collatz(n) =
  if n == 1
    then 1
    else if n % 2 == 0
      then collatz(n / 2)
      else collatz(3*n + 1)
```

According to the Collatz conjecture, you will always reach the number 1 at some point. While we have not found any number n to start with that would not end up at 1, it is still an open question: does the program `collatz(n)` halt in finite time for all possible integers n, or is there an n for which it runs forever?

As long as our program `collatz(n)` halts in finite time for an n, it is easy to see that it halts. But if there is an n_∞ for which `collatz(n)` never stops, how do we determine that? How do you discern between an extremely long running time and infinite running time? Just running the program obviously won't do and we might want to use more elaborate schemes. Perhaps the numbers n go into a cycle (where 1 never occurs), so we could track the n's and check if we ever go back to an n that has already occurred before. But if the n are ever increasing without repetition, then this will not work, either. So we might try to think of some sort of static analysis or come up with an even more complex program.

However, even if we finally managed to write a program that could decide with certainty for all n whether the `collatz` program terminates, there will be another program f for which we do not succeed. This is what Church and Turing's answer to *Halting Problem* says: no matter how good our program is, there is always at least one combination of a program f and some input n, for which we cannot decide whether $f(n)$ halts in finite time or not.

Another informal proof. Let us assume that we have written a program `check_halt(a, d)` that takes some program code a and the input/arguments d with which to run the program in a. If for some given input d, $a(d)$ halts in finite time, then `check_halt(a, d)` returns *true*, otherwise it returns *false*. To determine if the `collatz` program above halts for $n = 12345$, say, you would then call:

```
check_halt("def collatz(n):\n  while n != 1: ...", [12345])
```

We now construct a new program C that takes some program code as its argument a. It then checks whether the program in a would halt if it was given the string a as its argument. If the program in a halts, then it checks again, being caught in an infinite loop. If the program does not halt, it returns *true*.

```
def C(a: str):
    while check_halt(a, a):
        pass
    return True
```

Obviously, `C("def f(): f()")` would return *true* but never halt for `C("def g(): return 123")`.

What happens now if we feed C with its own program code? That is, we run:

```
code = """
def C(a: st):
    while check_halt(a, a):
        pass
    return True
"""

C(code)
```

If C – when run with its own program code – halts (i.e. `check_halt(C, C)` is *true*), then C obviously checks again and never halts. However, if C does *not* halt, then `check_halt(C, C)` is *false* and the program halts and returns *true*.

We are caught in a circle here: if C halted when fed its own program code, then it would not halt, and if C did not halt, it would halt. This is obviously a logical contradiction and we therefore conclude that there cannot be a function `check_halt(a, d)` in the first place. In other words: there cannot be an algorithm that determines for all possible combinations of programs and inputs whether the program would halt with the given input.

Code and data From our modern point of view, there is certainly nothing spectacular or noteworthy about the idea of having a value (a text string) that represents the program (code). With some ASCII or unicode encoding, we can even easily understand how the program might be expressed as a (potentially very large) integer number. Likewise, executable binaries encode a program as a sequence of bytes, essentially forming a very long integer number as well.

However, in the early 20th century, when Turing, Church, and Gödel were working on these problems, the idea of expressing a formula, computation or algorithm as an integer number was absolutely revolutionary! It is really this insight that code can be expressed as data that lies at the heart of their proofs.

1.4 Provability and Truth*

Although strongly related, *provable* and *true* are not exactly the same thing. While we would expect from any sensible and correct system that all provable theorems are true, there might be true theorems, which are not provable. In fact, it was one of the open problems whether all true theorems (in mathematics) are provable. The answer was given by *Kurt Gödel* in his famous *incompleteness theorem*: if you have a consistent set of formulas (theorems) that is strong enough to express arithmetic, then there are always formulas which could (consistently) be either true or false, i.e. are not provable within the system.

This might become clearer if we look at a concrete example. The fifth axiom in Euclid's geometry (also known as the *parallel postulate*) basically says that given a line and a point not on it, there is at most one line *parallel* to the given line such that the point lies on the parallel line.

For centuries, mathematicians felt that this postulate should not be an axiom, but that you could prove it based on the other axioms. However, all attempts of finding a proof failed. Finally, it was discovered that it does not necessarily have to be true in the first place. If you consider what is now called "Euclidean geometry", i.e. geometry on a flat plane, then the axiom holds. But if you consider geometry on a sphere, say, it no longer holds even though the rest of the axioms of geometry still make sense.

Picture two ships, say, both starting somewhere on the equator, some distance apart, and heading North. Clearly, the two ships have parallel courses. Yet, at the North pole, their trajectories will meet. The same holds for any two starting points and headings. In fact, on the surface of a sphere, given a line and a point (not on the line), any second line with the point on it will meet the first line somewhere. There are no "truly" parallel lines, altogether.

The existence of consistent geometries with and without the parallel axiom shows that this axiom is indeed independent of the other axioms, and thus cannot be proven (or disproven) within the system. Gödel's work showed that there is in fact an infinite number of such independent theorems or statements. In other words: we can never find a finite set of axioms such that every true statement in our system can be proven.

Thus, whether a statement is true or not is an *extrinsic* property, something that depends on how you apply the logical system or theory (e.g., do you apply the theoremy of geometry to a flat plane or the surface of a sphere). But whether a statement is provable or not is an *intrinsic* property. Accordingly, the Entscheidungsproblem does not ask for an algorithm that decides whether a statement is true, but whether it is provable.

2 Register Machines

Register machines provide us with an accessible and simple model of what we mean by a "computer program", without being bogged down by questions of the exact implementation. With its registers of infinite size, it is a purely theoretical construct, but exactly what we need to then think about the limitations of computability (as opposed to limitations imposed by concrete hardware implementations).

2.1 Register Machines, Informally

A register machine operates on natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ stored in (idealised) registers using the following "elementary operations":

- addition of 1 to the contents of a register,
- test of whether the contents of a register is 0,
- subtraction of 1 from the contents of a register if it is non-zero,
- jumps ("goto"),
- conditionals ("if then else")

Definition 1: Register machine

A *register machine* is specified by:

- finitely many registers $R_0, R_1, \dots R_n$ (each capable of storing a natural number);
- a program consisting of a finite list of instructions of the form *label: body*, where for $i = 0, 1, 2, \dots$ the $(i+1)^{\text{th}}$ instruction has label L_i. The instruction *body* takes one of three forms:

$R^+ \to L'$	add 1 to contents of register R and jump to instruction labelled L'
$R^- \to L', L''$	if contents of R is larger than 0, then subtract 1 from it and jump to L', else jump to L''
`HALT`	stop executing instructions

Example 1: We consider a small register machine with three registers and a program consisting of five instructions. On the right hand side you find a sample "trace" from running the machine. The program was started with R_0 containing 0, R_1 containing 1, R_2 containing 2, and ended with R_0 containing 3 and the other registers being zero. At each point in time, the machine is in a specific *configuration*, represented by the numbers in a line of the trace.

Register: R_0, R_1, R_2

Program:

$L_0 \; : \; R_1^- \to L_1, L_2$
$L_1 \; : \; R_0^+ \to L_0$
$L_2 \; : \; R_2^- \to L_3, L_4$
$L_3 \; : \; R_0^+ \to L_2$
$L_4 \; : \;$ `HALT`

ℓ	R_0	R_1	R_2
0	0	1	2
1	0	0	2
0	1	0	2
2	1	0	2
3	1	0	1
2	2	0	1
3	2	0	0
2	3	0	0
4	3	0	0

■

Register machine configuration A register machine **configuration** is a tuple:

$$c = (\ell, r_0, \dots, r_n)$$

where L_ℓ is the current label and r_i is the current content of register R_i. We will write "$R_i = x$ [in configuration c]" to mean that $c = (\ell, r_0, \dots, r_n)$ with $r_i = x$. An initial configuration is given by $c_0 = (0, r_0, \dots, r_n)$ where r_i is the initial content of register R_i.

Definition 2: Register machine computation

A **computation** of a register machine is a (finite or infinite) sequence of configurations

$$c_0, c_1, c_2, \dots$$

where

- c_0 is an initial configuration,
- each $c = (\ell, r_0, \dots, r_n)$ in the sequence determines the next configuration in the sequence (if any) by carrying out the program instruction labelled L_ℓ with registers containing $r_0, \dots, r_n$.

Halting For a finite computation $c_0, c_1, \dots, c_m$, the last configuration $c_m = (\ell, r_0, \dots)$ must be a *halting configuration*, i.e. ℓ must satisfy:

either: the ℓ^{th} instruction in the program has the body `HALT` (a “proper halt”);

or: ℓ is greater than the number of instructions in the program, so that there is no instruction labelled L_ℓ (an “erroneous halt”).

N.B. a program can always be modified (without affecting its computations) to turn all erroneous halts into proper halts by adding extra `HALT` instructions to the list with appropriate labels.

Note that computations may never halt. For example, the following register machine with one register R_0 has only infinite computation sequences of the form $(0, r), (0, r+1), (0, r+2), \dots$.

$$\begin{array}{lcl} L_0 & : & R_0^+ \to L_0 \\ L_1 & : & \texttt{HALT} \end{array}$$

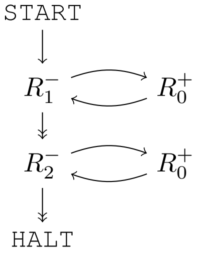
2.2 Graphical Representation

A register machine can be represented by a graph with one node (vertex) for each instruction. The arcs (edges) represent jumps between instructions and thereby replace the labels. Because the sequential ordering of instructions is lost, we need to indicate the initial instruction with `START`.

program code	**graphical representation**
$R^+ \to L$	$R^+ \to [L]$
$R^- \to L, L'$	$\begin{tikzcd}[row sep=small] & {[L]} \\ R^- \arrow[ur] \arrow[dr, two heads] & \\ & {[L']} \end{tikzcd}$
`HALT`	`HALT`
L_0	`START` $\longrightarrow [L_0]$

Example 2: Program with R_0, R_1, R_2:

$L_0 \; : \; R_1^- \to L_1, L_2$
$L_1 \; : \; R_0^+ \to L_0$
$L_2 \; : \; R_2^- \to L_3, L_4$
$L_3 \; : \; R_0^+ \to L_2$
$L_4 \; : \; \texttt{HALT}$

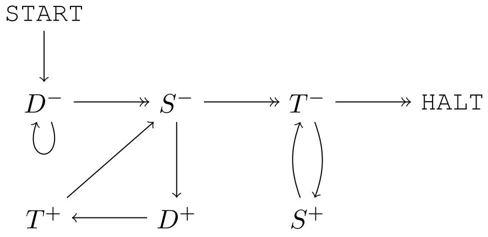


Claim: Starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$. ■

Example 3: We cannot read the contents of a register without emptying it. Copying a register to another thus requires a short program. In order to copy the contents of S to D ($D := S$, where S stands for *source* and D for *destination*), we first empty register D, then make two copies of S's contents, one in register D and another one in a previously unused temporary register T. We finally move the contents of T back to S.

Program with registers D, S, T:

$L_0 \; : \; D^- \to L_0, L_1$
$L_1 \; : \; S^- \to L_2, L_4$
$L_2 \; : \; D^+ \to L_3$
$L_3 \; : \; T^+ \to L_1$
$L_4 \; : \; T^- \to L_5, L_6$
$L_5 \; : \; S^+ \to L_4$
$L_6 \; : \; \texttt{HALT}$



■

2.3 Partial Functions

Register machine computation is *deterministic*: in any non-halting configuration, the next configuration is uniquely determined by the program. So the relation between initial and final register contents defined by a register machine program is a *partial function*.

Definition 3: Partial function

A *partial function* from a set X to a set Y is specified by any subset $f \subseteq X \times Y$ satisfying

$$(x, y) \in f \wedge (x, y') \in f \to y = y'$$

for all $x \in X$ and $y, y' \in Y$.

The cartesian product $X \times Y$ is the set of all ordered pairs $X \times Y = \{(x, y) | x \in X \wedge y \in Y\}$.

The definition of a partial function is to say that for all $x \in X$ there is *at most* one $y \in Y$ with $(x, y) \in f$. For a (total) function, in contrast, each $x \in X$ has *exactly* one $y \in Y$ with $(x, y) \in f$.

Notation:

- $f(x) = y$ means $(x, y) \in f$,
- $f(x) \downarrow$ means $\exists y \in Y\ (f(x) = y)$ (i.e. $f(x)$ is defined),
- $f(x) \uparrow$ means $\neg \exists y \in Y\ (f(x) = y)$ (i.e. $f(x)$ is undefined),
- $X \rightharpoonup Y$ is the set of all partial functions from X to Y,
- $X \rightarrow Y$ is the set of all (total) functions from X to Y.

Definition 4: Total function

A partial function from a set X to a set Y is *total* if it satisfies

$$f(x) \downarrow$$

for all $x \in X$.

2.3.1 Computable Functions

In essence, a partial function is computable if you can design a register machine that computes the partial function. We thereby require that the register machine fully implements the partial function: if the partial function is undefined for some input x, the register machine will not halt (and produce a meaningful result) for that input x, either.

Definition 5: Computable

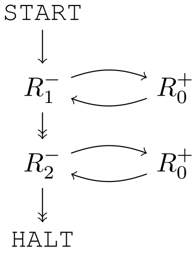
$f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is (*register machine*) *computable* if there is a register machine M with at least $n + 1$ registers $R_0, R_1, \ldots, R_n$ (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$, the computation of M starting with $R_0 = 0, R_1 = x_1, \ldots, R_n = x_n$ and all other registers set to 0, halts with $R_0 = y$ if and only if $f(x_1, \ldots, x_n) = y$.

Note the (somewhat arbitrary) I/O convention: in the initial configuration registers $R_1, \ldots, R_n$ store the function's arguments (with all others zeroed); and in the halting configuration register R_0 stores its value (if any).

Example 4: Registers: R_0, R_1, R_2

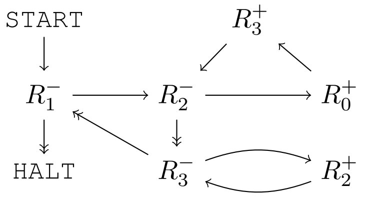
Program:

$$\begin{array}{lcl} L_0 & : & R_1^- \to L_1, L_2 \\ L_1 & : & R_0^+ \to L_0 \\ L_2 & : & R_2^- \to L_3, L_4 \\ L_3 & : & R_0^+ \to L_2 \\ L_4 & : & \texttt{HALT} \end{array}$$



Claim: starting from initial configuration $(0, 0, x, y)$, this machine's computation halts with configuration $(4, x + y, 0, 0)$. So $f(x, y) \triangleq x + y$ is *computable*. ■

Examples of computable functions The multiplication $f(x, y) \triangleq x \cdot y$ is computable. If the following machine is started with $(R_0, R_1, R_2, R_3) = (0, x, y, 0)$, it halts with $(R_0, R_1, R_2, R_3) = (xy, 0, y, 0)$.



Likewise, the following functions are all computable (the proof is left as an exercise):

Projection

$$p(x, y) \triangleq x$$

Constant

$$c(x) \triangleq n$$

Truncated subtraction

$$x \dot{-} y \triangleq \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } y > x \end{cases}$$

Integer division

$$x \text{ div } y \triangleq \begin{cases} \max\{z \in \mathbb{N} | zy \leq x\} & \text{if } y > 0 \\ 0 & \text{if } y = 0 \end{cases}$$

Integer remainder

$$x \text{ mod } y \triangleq x \dot{-} y(x \text{ div } y)$$

Exponentiation with base 2

$$e(x) \triangleq 2^x$$

Logarithm with base 2

$$\log_2(x) \triangleq \begin{cases} \max\{y \in \mathbb{N} | 2^y \leq x\} & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

3 Coding Programs as Numbers

A key insight when discussing computability is the idea that we can express programs (i.e. the machines) as data that can then be manipulated by the very machines themselves: we can encode programs with numbers. This allows a program to analyse itself.

Turing and Church's solutions of the Entscheidungsproblem uses the idea that (formal descriptions of) algorithms can be the data on which algorithms act. To realise this idea with register machines we have to be able to code register machine programs as numbers (in general, such codings are often called Gödel numberings). To do that, first we have to code pairs of numbers and lists of numbers as numbers. There are many ways to do that. We fix upon one.

You are probably already familiar with the idea that a computer program is essentially a sequence of "bytecode instructions", i.e. numbers representing various instructions. The encoding presented here might then strike you as rather inefficient. However, keep in mind that we are exploring possibilities and thus want to work with a minimal set of assumptions about the "computer" (in particular without such notions as what a "byte" or "machine word" is).

3.1 Numerical Coding of Pairs

For $x, y \in \mathbb{N}$, define

$$\begin{aligned} \langle\!\langle x, y \rangle\!\rangle &\triangleq 2^x(2y+1) \\ \langle x, y \rangle &\triangleq 2^x(2y+1) - 1 \end{aligned}$$

So in the binary representation of $\langle\!\langle x, y \rangle\!\rangle$, x is represented by trailing zeroes whereas in $\langle x, y \rangle$ by trailing ones:

$$\mathrm{bin}\left(\langle\!\langle x, y \rangle\!\rangle\right) = \mathrm{bin}(y)1\underbrace{0\cdots 0}_{x\times} \qquad \mathrm{bin}\left(\langle x, y \rangle\right) = \mathrm{bin}(y)0\underbrace{1\cdots 1}_{x\times}$$

E.g. $27 = 0\mathrm{b}11011 = \langle 0, 13 \rangle = \langle 2, 3 \rangle$.

- $\langle \cdot, \cdot \rangle$ gives a bijection (one-one correspondence) between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$.

- $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ gives a bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}_{>0}$.

Hence, we use $\langle \cdot, \cdot \rangle$ when we want to decode every natural numbers as a possible pair and $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ when we need a special marker (that cannot be a pair) for “null” or “end of data”.

3.2 Numerical Coding of Lists

Let $list\mathbb{N}$ be the set of all finite lists of natural numbers. We use ML notation for lists:

- empty list: $[] \in list\mathbb{N}$
- list-cons: $x :: \ell \in list\mathbb{N}$ (given $x \in \mathbb{N}$ and $\ell \in list\mathbb{N}$)
- $[x_1, x_2, \ldots, x_n] \triangleq x_1 :: (x_2 :: (\cdots x_n :: []) \cdots)$

For $\ell \in list\mathbb{N}$, define $\ulcorner \ell \urcorner$ by induction on the length of the list ℓ:

$$\ulcorner [] \urcorner \triangleq 0 \qquad \ulcorner x :: \ell \urcorner \triangleq \langle\!\langle x, \ulcorner \ell \urcorner \rangle\!\rangle = 2^x \, (2 \cdot \ulcorner \ell \urcorner + 1)$$

The binary representation of a list thus looks like:

$$\mathrm{bin}(\ulcorner [x_1, x_2, \ldots, x_n] \urcorner) = 0\mathrm{b}1 | \underbrace{0 \cdots 0}_{x_n} 1 \underbrace{0 \cdots 0}_{x_{n-1}} \cdots 1 \underbrace{0 \cdots 0}_{x_1}$$

Hence $\ell \mapsto \ulcorner \ell \urcorner$ gives a bijection from $list\mathbb{N}$ to $\mathbb{N}$.

Note that we need 0 as a marker for the empty list and therefore use $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ rather than $\langle \cdot, \cdot \rangle$ to cons the list elements.

Example 5:

- $\ulcorner [3] \urcorner = \ulcorner 3 :: [] \urcorner = \langle\!\langle 3, 0 \rangle\!\rangle = 2^3(2 \cdot 0 + 1) = 8 = 0\mathrm{b}1000$
- $\ulcorner [1, 3] \urcorner = \langle\!\langle 1, \ulcorner [3] \urcorner \rangle\!\rangle = \langle\!\langle 1, 8 \rangle\!\rangle = 34 = 0\mathrm{b}100010$
- $\ulcorner [2, 1, 3] \urcorner = \langle\!\langle 2, \ulcorner [1, 3] \urcorner \rangle\!\rangle = \langle\!\langle 2, 34 \rangle\!\rangle = 276 = 0\mathrm{b}100010100$

■

Example 6: The binary encoding $0\mathrm{b}1000010010000000100010000010011001000$ corresponds to the list $[3, 2, 0, 2, 5, 3, 6, 2, 4]$ (observe that the encoding is from right to left). ■

3.3 Numerical Coding of Programs

If P is the register machine program

$$\begin{bmatrix} L_0 & : & \mathit{body}_0 \\ L_1 & : & \mathit{body}_1 \\ & \vdots & \\ L_n & : & \mathit{body}_n \end{bmatrix}$$

then its numerical code is

$$\ulcorner P \urcorner \triangleq \ulcorner [\ulcorner \mathit{body}_0 \urcorner, \ldots, \ulcorner \mathit{body}_n \urcorner] \urcorner$$

where the numerical code $\ulcorner \mathit{body} \urcorner$ of an instruction is defined by:

$$\begin{array}{rcl} \ulcorner R_i^+ \to L_j \urcorner & \triangleq & \langle\langle 2i, j \rangle\rangle \\ \ulcorner R_i^- \to L_j, L_k \urcorner & \triangleq & \langle\langle 2i+1, \langle j, k \rangle \rangle\rangle \\ \ulcorner \texttt{HALT} \urcorner & \triangleq & 0 \end{array}$$

Note again that we use $\langle\langle \cdot, \cdot \rangle\rangle$ because we need 0 as a special marker for `HALT`. To encode the target labels L_j, L_k, on the other hand, we use $\langle j, k \rangle$ so that each possible number can be decoded into two labels. Hence, any $x \in \mathbb{N}$ decodes to a unique instruction $\mathrm{body}(x)$. In ML-like code, we could write a function $\mathrm{body}()$ that decodes a number to an instruction as follows:

```
fun body(0) = HALT
  | body(⟪2i, z⟫) = $R_i^+ \to L_z$
  | body(⟪2i+1, z⟫) =
        let ⟨j, k⟩ = z
         in $R_i^- \to L_j, L_k$
```

So any $e \in \mathbb{N}$ decodes to a unique program $\mathrm{prog}(e)$, called the register machine *program with index* e:

$$\mathrm{prog}(e) \triangleq \begin{bmatrix} L_0 & : & \mathrm{body}(x_0) \\ & \vdots & \\ L_n & : & \mathrm{body}(x_1) \end{bmatrix}$$

where $e = \ulcorner [x_0, \ldots, x_n] \urcorner$.

Example 7:

- $786432 = 2^{19} + 2^{18} = \mathrm{0b11}\underbrace{0\ldots0}_{18\times} = \ulcorner [18, 0] \urcorner$
- $18 = \mathrm{0b10010} = \langle\langle 1, 4 \rangle\rangle = \langle\langle 1, \langle 0, 2 \rangle \rangle\rangle = \ulcorner R_0^- \to L_0, L_2 \urcorner$
- $0 = \ulcorner \texttt{HALT} \urcorner$

So

$$\mathrm{prog}(786432) = \left[\begin{array}{lcl} L_0 & : & R_0^- \to L_0, L_2 \\ L_1 & : & \texttt{HALT} \end{array}\right]$$

■

N.B. In case $e = 0$ we have $0 = \ulcorner [] \urcorner$, so $\mathrm{prog}(0)$ is the program with an empty list of instructions, which by convention we regard as a register machine that does nothing (i.e. that halts immediately).

4 Universal Register Machine

After encoding programs as data, we now show that a computer program can not only "read" such an encoded program in an abstract sense, but that it is possible to build a program than can execute/run any such encoded program. In other words: we can build an interpreter that runs all computer programs, including itself.

After we have shown that every program can be coded as a number n, we need to show that a register machine is capable of decoding such a program and execute the program's instruction. In other words: we can build a complete "interpreter" for register machine programs as a register machine program itself.

In fact, this is quite common practise. We use *virtual machines* to emulate a certain type of computer through a program that runs on a different computer. It is even possible to run a virtual machine that emulates a PC inside your browser, and then, of course, to run yet another browser or virtual machine inside that emulated machine.

A machine that is powerful enough to run programs that emulate any other (computing) machine is called *Turing complete*. By proving that a (universal) register machine can emulate any other register machine, we make a first step towards establishing that it is indeed Turing complete and hence that we are not "missing" something.

4.1 High-level specification

The universal register machine U carries out the following computation, starting with $R_0 = 0$, $R_1 = e$ (i.e. the code of the program), $R_2 = a$ (the encoded list of arguments) and all other registers zeroed:

1. decode e as a register machine program P;
2. decode a as a list of register values $a_1, \ldots, a_n$;
3. carry out the computation of the register machine program P starting with $R_0 = 0, R_1 = a_1, \ldots, R_n = a_n$ (and any other register occurring in P set to 0).

Mnemonics for the registers of U and the role they play in its program:

$R_1 \equiv P$	code of the register machine to be simulated
$R_2 \equiv A$	code of current register contents of a simulated register machine
$R_3 \equiv PC$	program counter – number of the current instruction (counting from 0)
$R_4 \equiv N$	code of the current instruction body
$R_5 \equiv C$	type of the current instruction body ($2i \equiv R_i^+$, $2i+1 \equiv R_i^-$)
$R_6 \equiv R$	current value of the register to be incremented or decremented by current instruction (if not `HALT`)
$R_7 \equiv S$	auxiliary register
$R_8 \equiv T$	auxiliary register
$R_9 \equiv Z$	auxiliary register

4.2 Overall Structure of U's Program

1. `N := P[PC]; GOTO 2`

   Copy the PC^{th} item of the list (program) in P to register N, which thus contains the current instruction body.

2. `if (N = 0) { R0 := A[0]; HALT } else { decode N; GOTO 3 }`

   Either halt and return the result in R_0 or decode the instruction to N: if N is zero, then copy the 0th item of the list in A (i.e. register R_0) to R_0 and halt, else decode N as $\langle\!\langle y, z\rangle\!\rangle$, copy y to C, z to N, and goto 3.

   At this point either $C = 2i$ is even and the current instruction is $R_i^+ \to L_z$, or $C = 2i+1$ is odd and the current instruction is $R_i^- \to L_j, L_k$ where $z = \langle j, k\rangle$.

3. `R := A[i]; GOTO 4`

   Load the contents of the instruction's register to R: copy the i^{th} item of the list in A to R and goto 4.

4. `exec(C); update(PC); A[i] := R; GOTO 1`

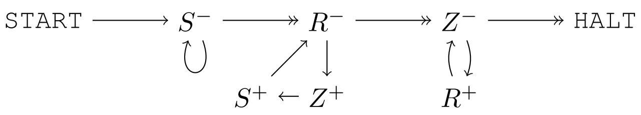
   Execute the current instruction (which is in C) on R, update the PC to the next label (which is in N), and restore the register value R to A. Then goto 1.

To implement this, we need register machines for manipulating (codes of) lists of numbers. . .

Copying registers The program

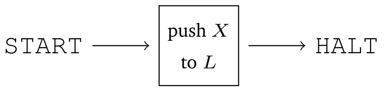
$$\texttt{START} \longrightarrow \boxed{S ::= R} \longrightarrow \texttt{HALT}$$

to copy the contents of R to S can be implemented by:

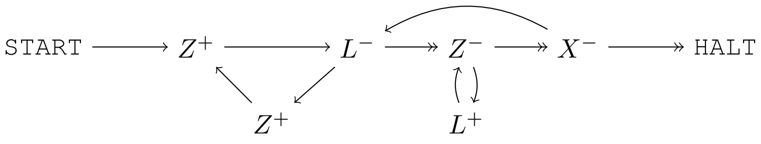


Precondition: $R = x, S = y, Z = 0$
Postcondition: $R = x, S = x, Z = 0$

Pushing an element to a list (cons) The program

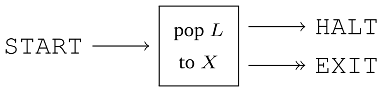

START
push X to L
HALT

to carry out the assignment $(X, L) ::= (0, X :: L)$ can be implemented by:


START
Z^+
L^-
Z^-
X^-
HALT
Z^+
L^+

Precondition: $X = x, L = \ell, Z = 0$
Postcondition: $X = 0, L = \langle\!\langle x, \ell \rangle\!\rangle = 2^x(2\ell + 1), Z = 0$

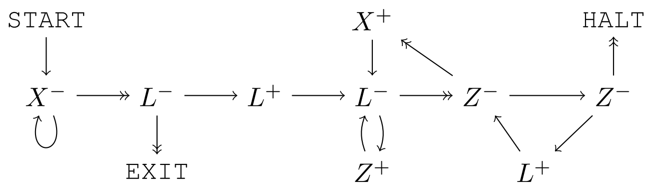
Popping an element from a list The program


START
pop L to X
HALT
EXIT

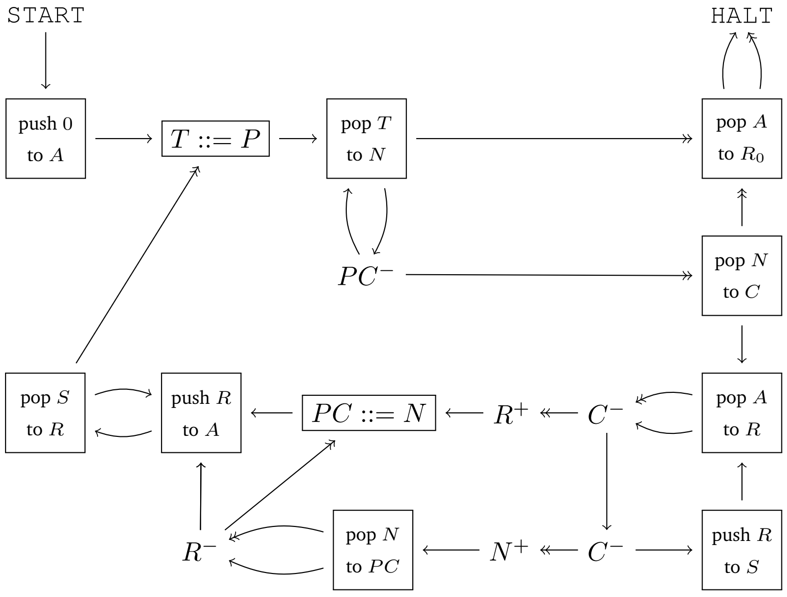
specified by

```
if L = 0
  then (X ::= 0, goto EXIT)
  else let $\langle\!\langle x, \ell \rangle\!\rangle$ = L
        in (X ::= x; L ::= $\ell$; goto HALT);
```

can be implemented by:


START
X^+
HALT
X^-
L^-
L^+
L^-
Z^-
Z^-
EXIT
Z^+
L^+

4.3 The Program for U



Programs, Machines, and Turing Completeness

A *program* describes the behaviour of a *machine*. Hence, a machine is a tuple $(\mathcal{C}, \mathcal{P})$ of components $\mathcal{C}$ and a program $\mathcal{P}$. In the case of a register machine, the components are the registers, i.e. $\mathcal{C} = (R_0, R_1, \ldots, R_n)$.

Processors run a sophisticated program very much like the universal machine desribed above. The processor needs to fetch the next instruction code from memory, decode it, and then take actions according to the meaning (i.e. semantics) of the instruction. The set of instructions is usually described by the *instruction set architecture* (ISA). In case of the register machine, this would encompass the three basic instructions `HALT`, $R_i^+ \to L_j$, and $R_i^- \to L_j, L_k$.

High-level programming languages allow us to use more sophisticated models for the machine. However, the machine is not always as explicit as in Pascal with its separate variable declaration (Program 1). Nonetheless, we consider a "computer program" to be a machine on its own. It is then the job of a PC to emulate/run this machine. To facilitate this emulation, compilers usually translate the original program code to an equivalent code based on the processor's ISA.

Whenever a programming language (or ISA) allows us to write programs that emulate any other machine (at least in principle and not considering resource constraints), the programming language is called *Turing complete*. Saying that the register machines are Turing complete means that you could write an emulator for any computer or processor, say, using nothing but the three basic register machine instructions presented above (the emulator might run incredibly slowly, but this is of no importance here).

Program 1 A “program” in Pascal actually describes the machine: after the `VAR` keyword you find the machine’s components, and following the `BEGIN` the actual program code.

```
PROGRAM GCD;
VAR
  A, B, T: INTEGER;
BEGIN
  READ(A); READ(B);
  WHILE B <> 0 DO
    BEGIN
        T := A mod B;
        A := B;
        B := T;
    END;
  WRITELN(A);
END.
```

As a first step towards Turing completeness, we have proven in this chapter that we can build register machines that can emulate any other register machine – even including themselves (this is important!). Hence, we could have a universal register machine execute another universal register machine that executes yet another universal register machine, and so on.

Finally, compare this to compilers, which are often written in the programming language they compile (i.e. a Pascal compiler would be written in Pascal itself). This shows that the programming language in question is powerful enough to write programs that “understand” and execute or translate any program written in the same language.

5 The Halting Problem

The halting problem plays a key role as it points out some limitations of computability. While the previous chapter(s) showed that we can build a program that runs any program, we now show that we cannot build a program that successfully analyses all programs. Sometimes we cannot even answer such a simple question as whether a specific program ever halts and produces an answer or runs forever.

Definition 6: Halting Problem

A register machine H *decides the Halting Problem* if for all $e, a_1, \ldots, a_n \in \mathbb{N}$, starting H with

$$R_0 = 0 \qquad R_1 = e \qquad R_2 = \ulcorner[a_1, \ldots, a_n]\urcorner$$

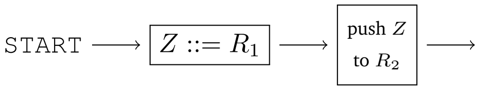
and all other registers zeroed, the computation of H always halts with R_0 containing 0 or 1; moreover when the computation halts, $R_0 = 1$ if and only if the register machine program with index e eventually halts when started with $R_0 = 0$, $R_1 = a_1, \ldots, R_m = a_n$ and all other registers zeroed.

Theorem 7:

No register machine H that decided the Halting Problem can exist.

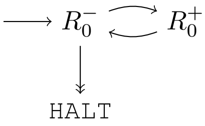
Proof of the theorem Assume we have a register machine H that decides the Halting Problem and derive a contradiction as follows.

- Let H' be obtained from H by replacing `START`$\rightarrow$ by:



(where Z is a register not mentioned in H's program).

- Let C be obtained from H' by replacing each `HALT` (and each erroneous halt) by



- Let $c \in \mathbb{N}$ be the index of C's program.

$$\begin{gathered}
\textit{C started with $R_1 = c$ eventually halts} \\
\text{if and only if} \\
H' \text{ started with } R_1 = c \text{ halts with } R_0 = 0 \\
\text{if and only if} \\
H \text{ started with } R_1 = c, R_2 = \ulcorner [c] \urcorner \text{ halts with } R_0 = 0 \\
\text{if and only if} \\
\mathrm{prog}(c) \text{ started with } R_1 = c \text{ does not halt} \\
\text{if and only if} \\
\textit{C started with $R_1 = c$ does not halt.}
\end{gathered}$$

This is a contradiction: C started with $R_1 = c$ halts if and only if it does not halt. ☐

5.1 Computable Functions

Recall that a (partial) function f is *computable* if there is a register machine that computes f.

Definition 8: Computable

A partial function $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is *(register machine) computable* if there is a register machine M with at least $n+1$ registers $R_0, R_1, \ldots, R_n$ (and maybe more) such that for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and all $y \in \mathbb{N}$, the computation of M starting with $R_0 = 0, R_1 = x_1, \ldots, R_n = x_n$ and all other registers set to 0, halts with $R_0 = y$ if and only if $f(x_1, \ldots, x_n) = y$.

Note that the same register machine M could be used to compute a unary function ($n = 1$), or a binary function ($n = 2$), etc. From now on we will concentrate on the unary case.

5.1.1 Enumerating Computable Functions

For each $e \in \mathbb{N}$, let $\varphi_e \in \mathbb{N} \rightharpoonup \mathbb{N}$ be the unary partial function computed by the register machine with program $\mathrm{prog}(e)$. So for all $x, y \in \mathbb{N}$: $\varphi_e(x) = y$ holds iff the computation of $\mathrm{prog}(e)$ started with $R_0 = 0, R_1 = x$ and all other registers zeroed eventually halts with $R_0 = y$.

Thus $e \mapsto \varphi_e$ defines an *onto* function from $\mathbb{N}$ to the collection of all computable partial function from $\mathbb{N}$ to $\mathbb{N}$.

5.1.2 An Uncomputable Function

Let $f \in \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial function with graph $\{(x, 0) | \varphi_x(x) \uparrow\}$. Thus:

$$f(x) = \begin{cases} 0 & \text{if } \varphi_x(x) \uparrow \\ \textit{undefined} & \text{if } \varphi_x(x) \downarrow \end{cases}$$

f is not computable, because if it were, then $f = \varphi_e$ for some $e \in \mathbb{N}$ and hence:

- if $\varphi_e(e) \uparrow$, then $f(e) = 0$ (by definition of f), so $\varphi_e(e) = 0$ (since $f = \varphi_e$), hence $\varphi_e(e) \downarrow$;
- if $\varphi_e(e) \downarrow$, then $f(e) \downarrow$ (since $f = \varphi_e$), so $\varphi_e(e) \uparrow$ (by definition of f).

This is contradiction. So f cannot be computable.

5.1.3 (Un)decidable Sets of Numbers

Given a subset $S \subseteq \mathbb{N}$, its *characteristic function* $\chi_S \in \mathbb{N} \to \mathbb{N}$ is given by:

$$\chi_S(x) \triangleq \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

Definition 9: Decidable

$S \subseteq \mathbb{N}$ is called (register machine) *decidable* if its characteristic function $\chi_S \in \mathbb{N} \to \mathbb{N}$ is a register machine computable function. Otherwise it is called *undecidable*.

So S is decidable iff there is a register machine M with the property: for all $x \in \mathbb{N}$, M started with $R_0 = 0, R_1 = x$ and all other registers zeroed eventually halts with R_0 containing 1 or 0; and $R_0 = 1$ on halting iff $x \in S$.

Basic strategy: to prove $S \subseteq \mathbb{N}$ is undeciable, try to show that decidability of S would imply decidability of the Halting Problem.

Example 8: **Claim:** $S_0 \triangleq \{e | \varphi_e(0) \downarrow\}$ is undeciable.

Proof (sketch): Suppose M_0 is a register machine computing χ_{S_0}. From M_0's program (using the same techniques as for constructing a universal register machine) we can construct a register machine H to carry out:

```
let $e = R_1$ and $\ulcorner[a_1, \dots, a_n]\urcorner = R_2$ in
      $R_1 ::= \ulcorner(R_1 ::= a_1); \cdots; (R_n ::= a_n); \mathrm{prog}(e)\urcorner$;
$R_2 ::= 0$;
run $M_0$
```

Then by assumption on M_0, H decides the Halting Problem—contradiction. So no such M_0 exists, i.e. χ_{S_0} is uncomputable, i.e. S_0 is undeciable. ∎

Example 9: **Claim:** $S_1 \triangleq \{e | \varphi_e \text{is a total function}\}$ is undecidable.

Proof (sketch): Suppose M_1 is a register machine computing χ_{S_1}. From M_1's program we can construct a register machine M_0 to carry out:

```
let $e = R_1$ in $R_1 ::= \ulcorner R_1 ::= 0; \mathrm{prog}(e) \urcorner$;
run $M_1$
```

Then by assumption on M_1, M_0 decides membership of S_0 from the previous example (i.e. computes χ_{S_0})—contradiction. So no such M_1 exists, i.e. χ_{S_1} is uncomputable, i.e. S_1 is undecidable. ■

6 Turing Machines

Turing machines provide us with an abstract model of what a computer program (or algorithm) is, just like register machines. Turing machines are slightly more abstract than register machines, but not more powerful.

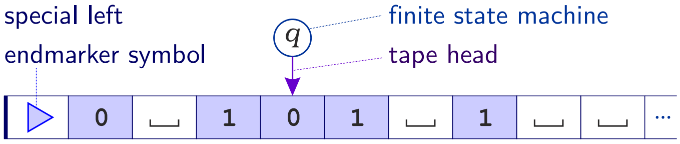
6.1 Turing Machines

At the time Hilbert posed the *Entscheidungsproblem*, there was no precise definition of “algorithm”, just examples. Common features of these examples were:

- a finite description of the procedure in terms of elementary operations (e.g., multiply two decimal digits by looking up their product in a table);
- determinism: the next step is uniquely determined, if there is one;
- the procedure may not terminate on some input data, but we can recognise when it does terminate and what the result is.

Register machine computation abstracts away from any particular, concrete representation of numbers (e.g., as bit strings) and the associated elementary operations of increment, decrement, and zero-test. Turing’s original model of computation (now called a *Turing machine*) is more concrete: even numbers have to be represented in terms of a fixed *finite* alphabet of symbols and increment, decrement, and zero-test have to be programmed in terms of more elementary symbol-manipulating operations.

Informally, a Turing machine is a finite state machine with a tape that is unbound “to the right”. The tape is linear and divided into cells, each of which contains a symbol of a finite alphabet of *tape symbols*, including a special *blank* symbol. Only finitely many cells contain non-blank symbols. A tape head can read or write the symbol in one specific cell, be moved one step to the left (unless the end of the tape is reached), or to the right.



- The Turing machine starts with the tape head pointing to the special left endmarker $\triangleright$.
- The Turing machine computes in discrete steps, each of which depends only on its current state q and the symbol being scanned by the tape head.
- There are five possible actions a Turing machine can take during any one step: overwrite the current tape cell with a symbol, move left or right one cell, stay stationary, and change state.

Definition 10: Turing Machine

A *Turing machine* is specified by a tuple (Q, Σ, s, δ) with:

- Q, a finite set of machine *states*,
- Σ, a finite set of tape *symbols* (disjoint from Q) containing distinguished symbols $\triangleright$ (*left endmaker*) and ␣ (*blank*),
- $s \in Q$, an *initial state*,
- $\delta \in (Q \times \Sigma) \to (Q \cup \{\mathrm{acc}, \mathrm{rej}\}) \times \Sigma \times \{L, R, S\}$, a *transition function* satisfying: for all $q \in Q$, there exists $q' \in Q \cup \{\mathrm{acc}, \mathrm{rej}\}$ with $\delta(q, \triangleright) = (q', \triangleright, R)$ (i.e. the left endmarker is never overwritten and the machine always moves to the right when scanning it).

Example 10: $M = (Q, \Sigma, s, \delta)$ with the states $Q = \{s, q, q'\}$, the symbols $\Sigma = \{\triangleright, \text{␣}, 0, 1\}$ and the transition function:

δ	$\triangleright$	␣	0	1
s	$(s, \triangleright, R)$	$(q, \text{␣}, R)$	$(\mathrm{rej}, 0, S)$	$(\mathrm{rej}, 1, S)$
q	$(\mathrm{rej}, \triangleright, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(\mathrm{rej}, \triangleright, R)$	$(\mathrm{acc}, \text{␣}, S)$	$(\mathrm{rej}, 0, S)$	$(q', 1, L)$

■

Definition 11: Configuration

A *Turing machine configuration* is a triple (q, w, u) where:

- $q \in Q \cup \{\mathrm{acc}, \mathrm{rej}\}$ is the current state,
- w is a non-empty string ($w = va$) of tape symbols under and to the left of the tape head, whose last element a is the contents of the cell under the head,
- u is a (possibly empty) string of tape symbols to the right of the tape head (up to some point beyond which alle symbols are blanks ␣).

A Turing machine starts in the *initial configuration* $(s, \triangleright, u)$.

6.2 Turing Machine Computation

Given a Turing machine $M = (Q, \Sigma, s, \delta)$, we write

$$(q, w, u) \rightarrow_M (q', w', u')$$

to mean that $q \neq \mathrm{acc}, rej$, $w = va$ (for some v, a) and exactly one of the following holds for $\delta(q, a)$:

- $\delta(q, a) = (q', a', L), w' = v, u' = a'u$,
- $\delta(q, a) = (q', a', S), w' = va', u' = u$,
- $\delta(q, a) = (q', a', R), u = a''u'', w' = va'a'', u' = u''$ and u is non-empty,
- $\delta(q, a) = (q', a', R), u = \varepsilon, w' = va'\text{\textvisiblespace}, u' = \varepsilon$.

Definition 12: Computation

A *computation* of a Turing machine M is a (finite or infinite) sequence of configurations $c_0, c_1, c_2, \ldots$ where:

- $c_0 = (s, \triangleright, u)$ is an initial configuration,
- $c_i \rightarrow_M c_{i+1}$ holds for each $i = 0, 1, \ldots$

The computation *does not halt* if the sequence is infinite. It *halts* if the sequence is finite and its last element is of the form (acc, w, u) or (rej, w, u) for some w and u.

Example 11: We consider the Turing machine $M = (Q, \Sigma, s, \delta)$ with the states $Q = \{s, q, q'\}$ (s is the initial state), the symbols $\Sigma = \{\triangleright, \text{\textvisiblespace}, 0, 1\}$, and the transition function:

$$\delta \in (Q \times \Sigma) \rightarrow (Q \cup \{\mathrm{acc}, \mathrm{rej}\}) \times \Sigma \times \{L, R, S\}$$

δ	$\triangleright$	$\text{\textvisiblespace}$	0	1
s	$(s, \triangleright, R)$	$(q, \text{\textvisiblespace}, R)$	$(\mathrm{rej}, 0, S)$	$(\mathrm{rej}, 1, S)$
q	$(\mathrm{rej}, \triangleright, R)$	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'	$(\mathrm{rej}, \triangleright, R)$	$(\mathrm{acc}, \text{\textvisiblespace}, S)$	$(\mathrm{rej}, 0, S)$	$(q', 1, L)$

We claim that the computation of M starting from configuration $(s, \triangleright, \text{\textvisiblespace}1^n0)$ halts in configuration $(\mathrm{acc}, \triangleright\text{\textvisiblespace}, 1^{n+1}0)$.

Indeed:

$$\begin{array}{rcl}
(s, \triangleright, \sqcup 1^n 0) & \to_M & (s, \triangleright \sqcup, 1^n 0) \\
& \to_M & (q, \triangleright \sqcup 1, 1^{n-1} 0) \\
& \vdots & \\
& \to_M & (q, \triangleright \sqcup 1^n, 0) \\
& \to_M & (q, \triangleright \sqcup 1^n 0, \varepsilon) \\
& \to_M & (q, \triangleright \sqcup 1^{n+1} \sqcup, \varepsilon) \\
& \to_M & (q', \triangleright \sqcup 1^{n+1}, 0) \\
& \vdots & \\
& \to_M & (q', \triangleright \sqcup, 1^{n+1} 0) \\
& \to_M & (\mathrm{acc}, \triangleright \sqcup, 1^{n+1} 0)
\end{array}$$

■

Theorem 13:

The computation of a Turing machine M can be implemented by a register machine.

Proof (sketch):

Step 1: fix a numerical encoding of M's states, tape symbols, tape contents and configurations;

Step 2: implement M's transition function (finite table) using Register Machine instructions on codes;

Step 3: implement a Register Machine program to repeatedly carry out the $\to_M$ operation.

Step 1. Identify states and tape symbols with particular numbers:

$$\begin{array}{rcl}
\mathrm{acc} & = & 0 \\
\mathrm{rej} & = & 1 \\
Q & = & \{2, 3, \ldots, n\} \\
\sqcup & = & 0 \\
\triangleright & = & 1 \\
\sigma & = & \{2, 3, \ldots, m\}
\end{array}$$

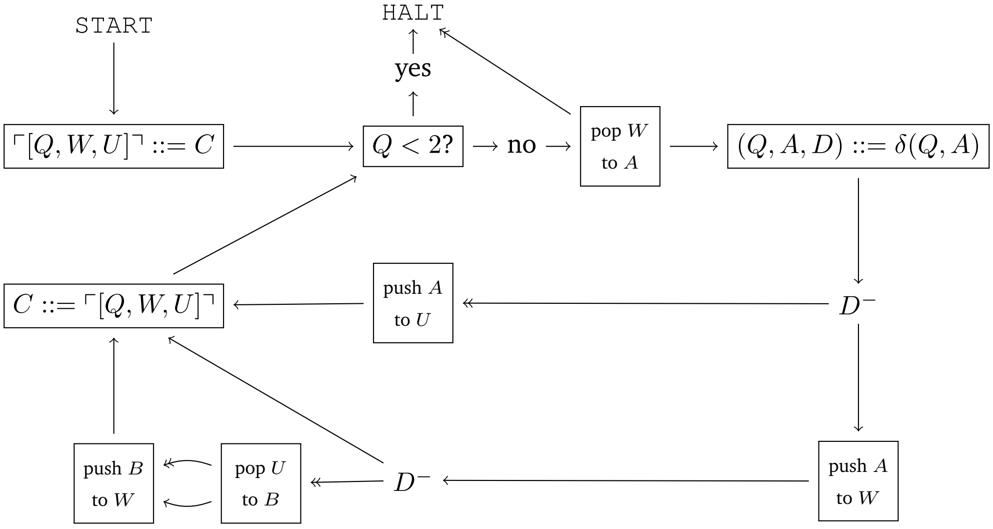
Encode configurations $c = (q, w, u)$ by:

$$\ulcorner c \urcorner = \ulcorner [q, \ulcorner [a_n, \dots, a_1] \urcorner, \ulcorner [b_1, \dots, b_m] \urcorner] \urcorner$$

where $w = a_1 \cdots a_n$ $(n > 0)$ and $u = b_1 \cdots b_m$ $(m \geq 0)$ say.

Step 2. Using the registers Q for the current state, A for the current tape symbol and D for the current direction of the tape head (with $L = 0, R = 1$ and $S = 2$, say) one can turn the finite table of (argument, result)-pairs specifying δ into a register machine program $\to \boxed{(Q, A, D) ::= \delta(Q, A)} \to$ so that starting the program with $Q = q, A = a, D = d$ (and all other registers zeroed), it halts with $Q = q', A = a', D = d'$, where $(q', a', d') = \delta(q, a)$.

Step 3. The register machine shown below will carry out M's computation. It uses the registers C for the code of the current configuration, W for the code of tape symbols at and to the left of the tape head (reading right-to-left) and U for the code of tape symbols right of the tape head (reading left-to-right). Starting with C containing the code of an initial configuration (and all other registers zeroed), the register machine program halts if and only if M halts; and in that case C holds the code of the final configuration.



□

6.3 Computable Functions

We have seen that a Turing machine's computation can be implemented by a register machine. The converse holds, too: the computation of a register machine can be implemented by a Turing machine. To make sense of this, we first have to fix a tape representation of register machine configurations and hence of numbers and lists of numbers.

Definition 14:

A tape over $\Sigma = \{\triangleright, \text{␣}, 0, 1\}$ *codes a list of numbers* if precisely two cells contains 0 and the only cells containing 1 occur between these.

Such tapes look like:

$$\triangleright\text{␣}\cdots\text{␣}0\underbrace{1\cdots1}_{n_1}\text{␣}\underbrace{1\cdots1}_{n_2}\text{␣}\cdots\text{␣}\underbrace{1\cdots1}_{n_k}0\text{␣}\cdots$$

which corresponds to the list $[n_1, n_2, \ldots, n_k]$.

Definition 15: Turing computable function

A function $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is *Turing computable* if and only if there is a Turing machine M with the following property:
Starting M from its initial state with tape head on the left endmarker of a tape coding $[0, x_1, \ldots, x_n]$, M halts if and only if $f(x_1, \ldots, x_n)\downarrow$, and in that case the final tape codes a list (of length ≥ 1) whose first element is y where $f(x_1, \ldots, x_n) = y$.

Theorem 16:

A partial function is Turing computable if and only if it is register machine computable.

Proof (sketch): We have seen how to implement any Turing machine by a register machine. Hence 'f is Turing computable' implies 'f is register machine computable'.

For the converse, one has to implement the computation of a register machine in terms of a Turing machine operating on a tape coding register machine configurations. To do this, one has to show how to carry out the action of each type of register machine instruction on the tape. It should be reasonably clear that this is possible in principle, even if the details (omitted) are tedious.

7 Notions of Computability

One important question that remains open is whether our models actually truly capture the notion of "computability" or "algorithm". There is, however, one important piece of evidence: all the different models that have been proposed so far are equivalent.

Church and Turing both developed formalisms for '*computability*' around the same time (1936): Church presented his λ-*calculus* (chapter 9) and Turing his *Turing machines*. Turing showed that the two very different approaches determine the same class of computable functions, which led to the *Church-Turing Thesis*:

Theorem 17: Church-Turing Thesis

Every algorithm [in an intuitive sense] can be realised as a Turing machine.

Further evidence for the thesis has been given by other formalisms that capture 'computability' since: Gödel and Kleene's *partial recursive functions* (1936); *canonical systems* for generating the theorems of a formal system, by Post (1943) and Markov (1951); the *register machines* of Lambek (1961) and Minsky (1961); as well as variations on all of these (e.g., multiple tapes, non-determinism, parallel execution, etc.). All have turned out to determine the same collection of computable functions.

In the rest of the course we will look at the *partial recursive functions* of Gödel and Kleene (1936), which gave rise to a branch of mathematics called *recursion theory* and Church's λ-*calculus* (1936), which in turn gave rise a branch of computer science called *functional programming*.

7.1 Computable Partial Functions

Our aim is to arrive at a more abstract, machine-independent description of the collection of *computable partial functions* than provided by register and Turing machines. Computable partial functions form the smallest collection of partial functions containing some *basic functions* and that is closed under some fundamental operations for forming new functions from old: *composition*, *primitive recursion* and *minimisation*. This characterisation is due to Kleene (1936), building on work of Gödel and Hebrand.

We build on three basic functions, which are all register machine computable.

Projection $\mathrm{proj}_i^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\mathrm{proj}\,_i^n(x_1, \ldots, x_n) \triangleq x_i$$

The projection is computed by:

$$\texttt{START} \longrightarrow \boxed{R_0 ::= R_i} \longrightarrow \texttt{HALT}$$

Constant (with value 0) $\mathrm{zero}^n \in \mathbb{N}^n \to \mathbb{N}$:

$$\mathrm{zero}\,^n(x_1, \ldots, x_n) \triangleq 0$$

The constant zero is computed by:

$$\texttt{START} \longrightarrow \texttt{HALT}$$

Successor $\mathrm{succ} \in \mathbb{N} \to \mathbb{N}$:

$$\mathrm{succ}(x) \triangleq x + 1$$

The successor is computed by:

$$\texttt{START} \longrightarrow R_1^+ \longrightarrow \boxed{R_0 ::= R_1} \longrightarrow \texttt{HALT}$$

Composition. The *composition* of $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ with $g_1, \ldots, g_n \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ is the partial function $f \circ [g_1, \ldots, g_n] \in \mathbb{N}^m \rightharpoonup \mathbb{N}$ satisfying for all $x_1, \ldots, x_m \in \mathbb{N}$:

$$f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) \equiv f\big(g_1(x_1, \ldots, x_m), \ldots g_n(x_1, \ldots, x_m)\big)$$

where $\equiv$ is the 'Kleene equivalence' of possibly undefined expressions: LHS $\equiv$ RHS means that either both LHS and RHS are undefined, or they are both defined and are equal.

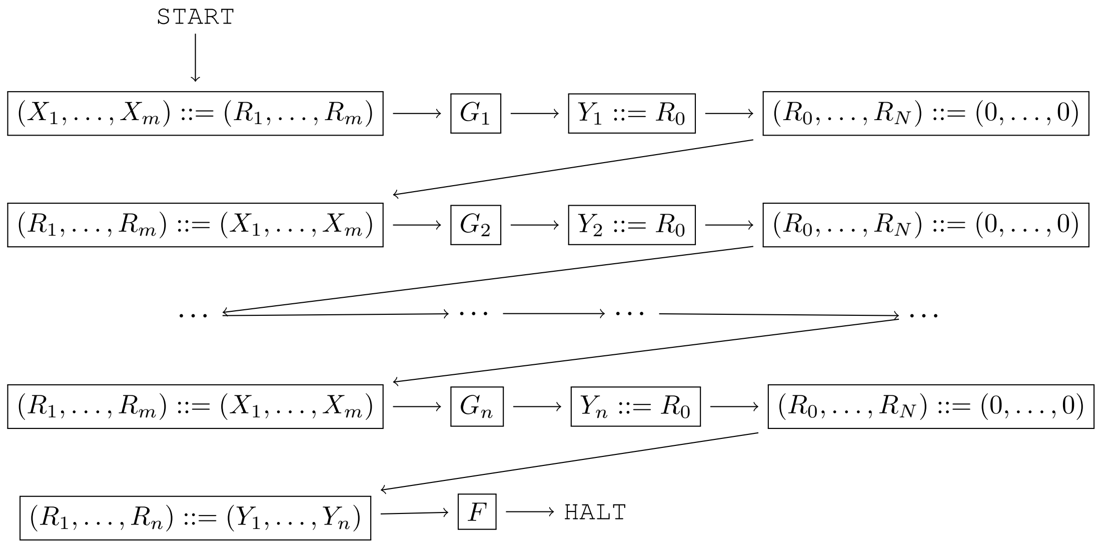
So $f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m) = z$ iff there exists $y_1, \ldots, y_n$ with $g_i(x_1, \ldots, x_m) = y_i$ for all $i = 1 \ldots n$ and $f(y_1, \ldots, y_n) = z$.

In case of $n = 1$, we simply write $f \circ g_1$ for $f \circ [g_1]$.

Theorem 18:

$f \circ [g_1, \ldots, g_n]$ is computable if f and $g_1, \ldots, g_n$ are.

Proof: Given register machine programs F and G_i computing $f(y_1, \ldots, y_n)$ and $g_i(x_1, \ldots, x_m)$, respectively, in R_0 starting with $R_1, \ldots, R_k$ set to $y_1, \ldots, y_n$ or $x_1, \ldots, x_m$, then the register machine program computing $f \circ [g_1, \ldots, g_n](x_1, \ldots, x_m)$ in R_0 starting with $R_1, \ldots, R_m$ set to $x_1, \ldots, x_m$ is specified by:


START
$(X_1, \ldots, X_m) ::= (R_1, \ldots, R_m)$
G_1
$Y_1 ::= R_0$
$(R_0, \ldots, R_N) ::= (0, \ldots, 0)$
$(R_1, \ldots, R_m) ::= (X_1, \ldots, X_m)$
G_2
$Y_2 ::= R_0$
$(R_0, \ldots, R_N) ::= (0, \ldots, 0)$
$\cdots$
$\cdots$
$\cdots$
$\cdots$
$(R_1, \ldots, R_m) ::= (X_1, \ldots, X_m)$
G_n
$Y_n ::= R_0$
$(R_0, \ldots, R_N) ::= (0, \ldots, 0)$
$(R_1, \ldots, R_n) ::= (Y_1, \ldots, Y_n)$
F
HALT

Note: we assume the programs $F, G_1, \ldots, G_n$ only mention registers up to R_N (where $N \geq \max\{n, m\}$) and that $X_1, \ldots, X_m, Y_1, \ldots, Y_n$ are some registers R_i with $i > N$. This assumption is prompted by the lack of *local names* for registers in the register machine model of computation.

□

8 Partial Recursive Functions

Primitive recursive functions are missing one key ingredient to express universal algorithms: the idea of unbound repetition. Many programming languages express this through a "while"-loop, which we formalise here using an idea called minimisation. *By generalising recursive functions in this way, we have to accept that they might not "terminate", i.e. they become* partial *recursive functions.*

Example 12: Examples of recursive definitions:

- $f_1(x)$ is the sum of the integers $0, 1, 2 \ldots, x$:

$$\begin{cases} f_1(0) & \equiv 0 \\ f_1(x+1) & \equiv f_1(x) + (x+1) \end{cases}$$

- $f_2(x)$ is the x^{th} Fibonacci number:

$$\begin{cases} f_2(0) & \equiv 0 \\ f_2(1) & \equiv 1 \\ f_2(x+2) & \equiv f_2(x) + f_2(x+1) \end{cases}$$

- $f_3(x)$ is undefined except when $x = 0$

$$\begin{cases} f_3(0) & \equiv 0 \\ f_3(x+1) & \equiv f_3(x+2) + 1 \end{cases}$$

- $f_4(x)$ is McCarthy's "91 function", which maps x to 91 if $x \leq 100$ and to $x - 10$ otherwise.

$$f_4(x) \equiv \text{if } x > 100 \text{ then } x - 10 \text{ else } f_4(f_4(x+11))$$

■

8.1 Primitive Recursion

Theorem 19:

Given $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \in \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$, there is a unique $h \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ satisfying

$$\begin{cases} h(\vec{x}, 0) & \equiv f(\vec{x}) \\ h(\vec{x}, x+1) & \equiv g(\vec{x}, x, h(\vec{x}, x)) \end{cases} \tag{8.1}$$

for all $\vec{x} \in \mathbb{N}^n$ and $x \in \mathbb{N}$.

We write $\rho^n(f, g)$ for h and call it the partial function *defined by primitive recusrion* from f and g. The recursion here is done on the parameter x. However, we allow the function h to have additional parameters not relevant for the recursion itself, which we collectively denote as $\vec{x}$. If we do not need any additional parameters, f basically turns into a natural number: $f() = n$.

Proof (sketch): *Existence:* the set h defines a partial function satisfying 8.1:

$$h \triangleq \left\{ (\vec{x}, x, y) \in \mathbb{N}^{n+2} \middle| \exists y_0, y_1, \ldots, y_x . f(\vec{x}) = y_0 \wedge \left(\bigwedge_{i=0}^{x-1} g(\vec{x}, i, y_i) = y_{i+1} \right) \wedge y_x = y \right\}$$

Uniqueness: if h and h' both satisfy 8.1, then one can prove by induction on x that $\forall \vec{x} . h(\vec{x}, x) \equiv h'(\vec{x}, x)$. ☐

Example 13: In the examples above, only $f_1(x)$ as the sum of the integers is in a primitive recursive form. In fact, in the notation of the theorem we have:

$$\begin{cases} h(\vec{x}, 0) & \equiv 0 \\ h(\vec{x}, x+1) & \equiv h(\vec{x}, x) + (x+1) \end{cases}$$

In this case, $\vec{x} \in \mathbb{N}^0$, so $\vec{x}$ is a vector of length zero. The basis function $f(\vec{x})$ is then actually just a natural number, i.e. $f() = 0$. The function $g(\vec{x}, x, h')$ is $g(x, h') = h' + (x+1)$, where we insert $h(\vec{x}, x)$ for h'.

Hence, we have $f_1(x) = \rho^0\big(0, ((x, h) \mapsto h + x + 1)\big)$. ■

Example 14: Addition The addition $add \in \mathbb{N}^2 \to \mathbb{N}$ satisfies

$$\begin{cases} add(x_1, 0) & \equiv x_1 \\ add(x_1, x+1) & \equiv add(x_1, x) + 1 \end{cases}$$

So, $add = \rho^1(f, g)$ where $f(x_1) \triangleq x_1$ and $g(x_1, x_2, h) \triangleq h + 1$.

Note that $f = \mathrm{proj}_1^1$ and $g = \mathrm{succ} \circ \mathrm{proj}_3^3$; so add can be built up from basic functions using composition and primitive recursion: $add = \rho^1(\mathrm{proj}_1^1, \mathrm{succ} \circ \mathrm{proj}_3^3)$. ■

Example 15: Predecessor The predecessor $pred \in \mathbb{N} \to \mathbb{N}$ satisfies

$$\begin{cases} pred(0) & \equiv 0 \\ pred(x+1) & \equiv x \end{cases}$$

So, $pred = \rho^0(f, g)$ where $f() \triangleq 0$ and $g(x, h) \triangleq x$.

Thus $pred$ can be built up from basic functions using primitive recursion: $pred = \rho^0(\mathrm{zero}^0, \mathrm{proj}_1^2)$.
■

Example 16: Multiplication The multiplication $mult \in \mathbb{N}^2 \to \mathbb{N}$ satisfies:

$$\begin{cases} mult(x_1, 0) & \equiv 0 \\ mult(x_1, x+1) & \equiv mult(x_1, x) + x_1 \end{cases}$$

and thus $mult = \rho^1(\mathrm{zero}^1, add \circ (\mathrm{proj}_3^3, \mathrm{proj}_1^3))$.

So $mult$ can be built up from basic functions using composition and primitive recursion (since add can be). ■

Note that a function does not necessarily need to use $\rho^n(f, g)$ in order to be primitive recursive. For example, $f(x) = x + 1$ is primitive recursive, as we have $f(x) = \mathrm{succ} \circ \mathrm{proj}_1^1$, without the need for actual recursion.

Definition 20: Primitive Recursive

A (partial) function f is *primitive recursive* ($f \in \mathrm{PRIM}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition and primitive recursion.

In other words, the set PRIM of primitive recursive functions is the *smallest* set (with respect to subset inclusion) of partial functions containing the basic functions and closed under the operations of composition and primitive recursion.

Every primitive recursive function $f \in \mathrm{PRIM}$ is a *total* function, because:

- all the basic functions are total,
- if $f, g_1, \ldots, g_n$ are total, then so is $f \circ (g_1, \ldots, g_n)$,
- if f and g are total, then so is $\rho^n(f, g)$.

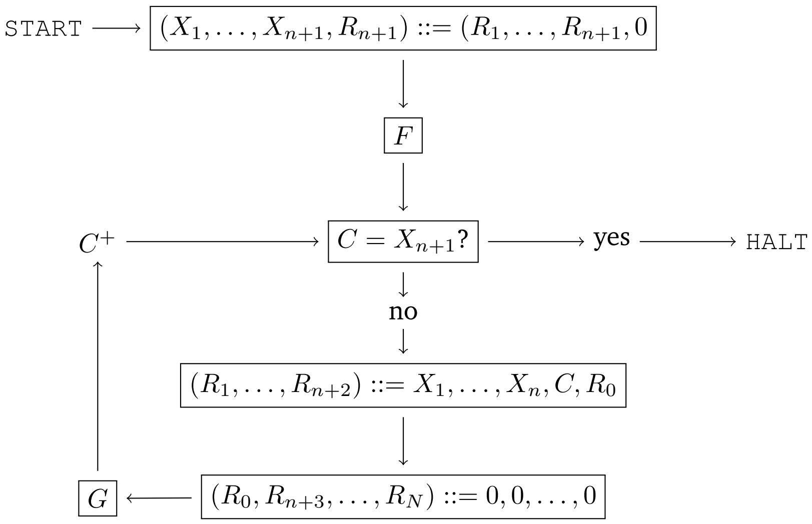
Theorem 21:

Every $f \in \mathrm{PRIM}$ is computable.

Proof. We already proved that basic functions are computable and that composition preserves computability. So we just have to show:

$\rho^n(f, g) \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is computable if $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \in \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$ are.

Suppose f and g are computed by register machine programs F and G (with our usual I/O conventions). Then the register machine specified below computes $\rho^n(f, g)$ (we assume that $X_1, X_2, \ldots, X_{n+1}, C$ are some registers not mentioned in F and G, and that the latter only use registers $R_0, \ldots, R_N$, where $N \geq n + 2$).


START
$(X_1, \ldots, X_{n+1}, R_{n+1}) ::= (R_1, \ldots, R_{n+1}, 0$
F
C^+
$C = X_{n+1}?$
yes
HALT
no
$(R_1, \ldots, R_{n+2}) ::= X_1, \ldots, X_n, C, R_0$
G
$(R_0, R_{n+3}, \ldots, R_N) ::= 0, 0, \ldots, 0$

□

8.2 Minimisation

Our aim is a more abstract, machine-independent description of the collection of computable partial functions than provided by register and Turing machines. The computable partial functions form the smallest collection of partial functions containing some basic functions and are closed under some fundamental operations for forming new functions from old — composition, primitive recursion and *minimisation*.

The primitive recursive functions so far are not yet able to capture the collection of all *computable* partial functions. By complementing composition and primitive recursion with *minimisation*, we finally obtain the full collection of computable partial functions. So, what is minimisation and why do we need it?

In essence, minimisations captures the idea of a "while-loop". At first glance, it looks as if we can already express loops by the means of primitive recursion $\rho^n(f, g)$ above (which, incidentally, allows us to also define an "if"-like conditional). However, this primitive recursion is a *bound* *loop*: one of the arguments specifies the maximum number of iterations it can go through. With minimisation, we remove this bound. Note that this comes at the cost of also introducing the possibility of functions “not terminating”, i.e. while primitive recursive functions are all total, functions using minimisation might be partial and thus undefined for some arguments.

Definition 22:

Minimisation Let $f \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ be a partial function. We then define the *minimisation* $\mu^n f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ as the least x such that $f(\vec{x}, x) = 0$ and for each $i < x$, $f(\vec{x}, i)$ is defined and larger than zero. We write $\mu^n f(\vec{x})$ for this value x.
If no such x exists (either because $f(\vec{x}, x)$ is never zero or because $f(\vec{x}, x')$ is undefined for an x' smaller than x), then $\mu^n f(\vec{x})$ is undefined.

In other words, the graph of $x = \mu^n f(\vec{x})$ presents itself as:

$$\mu^n f = \left\{ (\vec{x}, x) \in \mathbb{N}^{n+1} \,\middle|\, \exists y_0, \ldots, y_x . \left(\bigwedge_{i=0}^{x} f(\vec{x}, i) = y_i \right) \wedge \left(\bigwedge_{i=0}^{x-1} y_i > 0 \right) \wedge (y_x = 0) \right\}$$

We could also express $\mu^n(f)$ as programs in ML and Python, respectively:

```
fun mu(f, args) =
  let fun m(x) =
        if f(args, x) = 0
          then x
          else m(x + 1)
   in m(0);
```

```
def mu(f, *args):
    x = 0
    while f(*args, x):
        x += 1
    return x
```

Example 17: The integer part of the divison x_1/x_2 is the least x_3 such that $x_1 < x_2(x_3 + 1)$ (in case of $x_2 = 0$, it is undefined). Hence $x_3 = \mu^2 f(x_1, x_2)$, where $f \in \mathbb{N}^3 \to \mathbb{N}$ is:

$$f(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_1 \geq x_2(x_3 + 1) \\ 0 & \text{if } x_1 < x_2(x_3 + 1) \end{cases}$$

In fact, if we make the “integer part of x_1/x_2” function total by defining it to be 0 when $x_2 = 0$, it can be shown to be in PRIM. ■

8.3 Partial Recursive Functions

With minimisation we can now define the set PR of all “recursive (partial) functions”. As noted above, a side effect of introducing minimisation is that these functions are no longer guaranteed to be total, but can be partial. The set PR is therefore called the set of *partial recursive* functions.

Definition 23: Partial Recursive

A partial function f is *partial recursive* ($f \in \mathrm{PR}$) if it can be built up in finitely many steps from the basic functions by use of the operations of composition, primitive recursion, and minimisation.

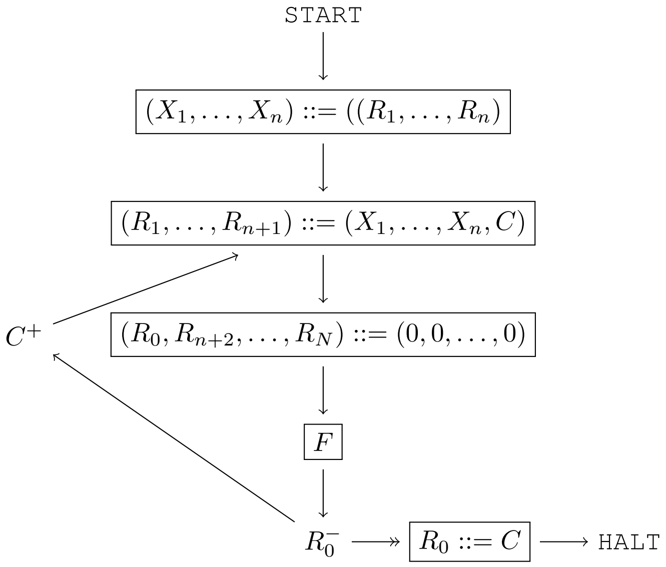
In other words, the set PR of partial recursive functions is the *smallest* set (with respect to subset inclusion) of partial functions containing the basic functions and closed under the operations of composition, primitive recursion, and minimisation.

Theorem 24:

Every partial recursive function $f \in \mathrm{PR}$ is computable.

Proof. We just have to show that $\mu^n f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is computable if $f \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is.

Suppose f is computed by a register machine program F (with our usual I/O conventions). Then the register machine specified below computes $\mu^n f$ (we assume $X_1, \dots, X_n, C$ are some registers not mentioned in F, and that F only uses registers $R_0, \dots, R_N$, where $N \geq n+1$).


START
$(X_1, \dots, X_n) ::= ((R_1, \dots, R_n)$
$(R_1, \dots, R_{n+1}) ::= (X_1, \dots, X_n, C)$
C^+
$(R_0, R_{n+2}, \dots, R_N) ::= (0, 0, \dots, 0)$
F
R_0^-
$R_0 ::= C$
HALT

□

Theorem 25:

Not only is every $f \in \mathrm{PR}$ computable, but conversely, *every computable partial function is partial recursive*.

Proof (sketch). Let $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ be computed by the register machine M with $N \geq n$ registers, say. Recall how we coded instantaneous configurations $c = (\ell, r_0, \ldots, r_N)$ of M as numbers $\ulcorner[\ell, r_0, \ldots, r_N]\urcorner$. It is possible to construct primitive recursive functions $\mathrm{lab}, \mathrm{val}_0, \mathrm{next}_M \in \mathbb{N} \to \mathbb{N}$ satisfying:

$$\begin{array}{rcl} \mathrm{lab}(\ulcorner[\ell, r_0, \ldots, r_N]\urcorner) & = & \ell \\ \mathrm{val}_0(\ulcorner[\ell, r_0, \ldots, r_N]\urcorner) & = & r_0 \\ \mathrm{next}_M(\ulcorner[\ell, r_0, \ldots, r_N]\urcorner) & = & \text{code of } M\text{'s next configuration} \end{array}$$

Showing that $\mathrm{next}_M \in \mathrm{PRIM}$ is tricky; we omit this proof.

Writing $\vec{x}$ for $x_1, \ldots, x_n$, let $\mathrm{config}_M(\vec{x}, t)$ be the code of M's configuration after t steps, starting with initial register values $R_0 = 0, R_1 = x_1, \ldots, R_n = x_n, R_{n+1} = 0, \ldots, R_N = 0$. It's in PRIM because:

$$\begin{cases} \mathrm{config}_M(\vec{x}, 0) & = \ulcorner[0, 0, \vec{x}, \vec{0}]\urcorner \\ \mathrm{config}_M(\vec{x}, t+1) & = \mathrm{next}_M(\mathrm{config}_M(\vec{x}, t)) \end{cases}$$

Without loss of generality, we can assume that M has a single `HALT` as last instruction, the i^{th}, say (and no erroneous halts). Let $\mathrm{halt}_M(\vec{x})$ be the number of steps M takes to halt when started with initial register values $\vec{x}$ (undefined if M does not halt). It satisfies

$$\mathrm{halt}_M(\vec{x}) \equiv \text{least } t \text{ such that } i - \mathrm{lab}(\mathrm{config}_M(\vec{x}, t)) = 0$$

and hence is in PR (because $\mathrm{lab}, \mathrm{config}_M, i - (\cdot) \in \mathrm{PRIM}$). So, $f \in \mathrm{PR}$, because $f(\vec{x}) \equiv \mathrm{val}_0\big(\mathrm{config}_M(\vec{x}, \mathrm{halt}_M(\vec{x}))\big)$. ☐

The members of PR that are total are called *(total) recursive functions*.

8.4 Ackermann's Function

Ackermann's function is a famous example of a (total) recursive function that is *not* primitive recursive. The basic idea behind Ackermann's function is to construct ever more powerful operators, starting from addition, multiplication, exponentiation, and extrapolating this sequence further.

A long time ago, you will probably have learned that multiplication is a convenient way to express repeated addition like so:

$$\underbrace{x + x + x + \cdots + x}_{n \text{ times}} = x \cdot n$$

Likewise, we can define exponentiation as a convenient way to express repeated multiplication:

$$\underbrace{x \cdot x \cdot x \cdots x}_{n \text{ times}} = x^n$$

The next step in our extrapolation becomes slightly more difficult to properly imagine:

$$\underbrace{x^{x^{\cdot^{\cdot^{\cdot x}}}}}_{n \text{ times}}$$

We use this idea to construct a family of functions. In contrast to the addition and multiplication above, however, we stick to a single argument. Addition thus becomes just an increment by 2 (why we use 2 should become clear in just a moment):

$$f_0(x) = x + 2$$

Likewise, we can take the "simplest" case of multiplication and exponentiation, respectively, as:

$$f_1(x) = 2 \cdot x \qquad f_2(x) = 2^x$$

That multiplication is repeated addition and exponentiation is repeated multiplication can easily be expressed through recursion:

$$f_1(x+1) = 2 \cdot (x+1) = 2 \cdot x + 2 = f_0(f_1(x)) \qquad f_2(x+1) = 2^{x+1} = 2 \cdot (2^x) = f_1(f_2(x))$$

It is natural to then also define $f_3(x+1) = f_2(f_3(x))$, etc. In fact, this general pattern of relationships $f_{n+1}(x+1) = f_n(f_{n+1}(x))$ is the core around which Ackermann's function is built. Instead of $f_n(x)$, Ackermann's funcion is usually written in the form $f(n, x)$. And if we use increment by one as our base case, we get the following:

$$f(0, x) = x + 1 \qquad f(n+1, x+1) = f(n, f(n+1, x))$$

The only missing part is a proper definition of $f_n(0) = f(n, 0)$ for all $n > 0$.

Definition 26: Ackermann's function

There is a (unique) function $ack \in \mathbb{N}^2 \to \mathbb{N}$ satisfying:

$$\begin{array}{rcl} ack(0, x_2) & = & x_2 + 1 \\ ack(x_1 + 1, 0) & = & ack(x_1, 1) \\ ack(x_1 + 1, x_2 + 1) & = & ack(x_1, ack(x_1 + 1, x_2)) \end{array}$$

The function ack is computable and hence recursive. However, ack grows faster than any primitive recursive function $f \in \mathbb{N}^2 \to \mathbb{N}$:

$$\exists N_f . \forall x_1, x_2 > N_f . f(x_1, x_2) < ack(x_1, x_2)$$

Hence, *ack is not primitive recursive*, although it is a total recursive function.

And considering that even $ack(3, x)$ is already a form of repeated exponentiation, the fast growth of $ack(n, x)$ should not be that surprising.

9 Lambda-Calculus

Compared to Turing and register machines, λ calculus provides a radically different way to think about computation. A program is here not a sequence of instructions, but rather a (complex) expression to be simplified. Computation is thus the process of simplifying an expression.

9.1 Notions of Computability

Church and Turing concurrently developed rigorous definitions of what an “algorithm” is, and both published their work in 1936: while Turing used *Turing-machines*, Church used λ-*calculus* (*Lambda*-calculus). Turing showed that the two very different approaches determine the same class of computable functions, giving rise to the Church-Turing thesis:

Theorem 27: Church-Turing Thesis

Every algorithm (in an intuitive sense) can be realised as a Turing machine.

Turing’s approach to computation is essentially based on an “abacus” (an ancient tool that encodes numbers through beads on wires, Figure 9.1). The computer (originally a human) follows a set of instructions, manipulating the beads on her abacus until she arrives at the desired result. Since the abacus might not be large enough to capture all computations, we allow the computer to also have an unlimited number of notepads to temporarily store some information (the tape).

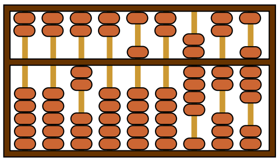


Figure 9.1: An ’abacus’ is an ancient calculating tool.

Church, in contrast, followed more a model of abstract mathematics as manipulating symbols as in simplifying formulas and equations. According to the rules of algebra we know that we can replace $x + y$ by $y + x$, or $a + a$ by $2a$, etc. In this perspective, calculating is just a game of replacing symbols with others, following specific patterns and rules. There is no abacus or machine involved, other than the computer (again, the human) performing the substitutions.

In modern terms, we could say that Turing developed a model of *imperative programming* while Church developed a model of *functional programming*. Both models have, of course, their place and provide important insights into matters of computation. This chapter will look at Church's approach with *Lambda*-calculus.

9.2 Introduction

Lambda-calculus is all about the principle of *substitution*. Given a term or formula, you look for certain patterns and replace a term with a simpler, but equivalent term (strictly speaking, it does not need to be simpler, although that is the overall aim).

Simplifying terms and formulas. The notion of doing mathematics by replacing things is already taught to us at a very early age: do you remember when you had to learn the multiplication table by heart? Instead of actually recalculate 7×8 every time it comes up, you just learned that you can *replace* it by 56. The multiplication of larger numbers is then expressed through algorithms that usually make use of this replacement scheme for small numbers. In other words, all multiplication can be done with a small lookup table, some algorithm, and the idea of replacing things.

The character of replacement might become clearer when applying it to simplify a logical formula, rather than an arithmetic or algebraic expression.

Example 18: Based on a few logical identities such as $A \vee B \Leftrightarrow B \vee A$, we can often simplify a boolean expression by means of simple substitutions. On the right we have written down the substitution patterns we are applying to the terms on the left in order to derive the next line.

$$\begin{array}{rcl}
(\neg q \rightarrow p) \wedge (q \rightarrow p) & \| & (A \rightarrow B) \Leftrightarrow (\neg A \vee B), \neg\neg C \Leftrightarrow C \\
(q \vee p) \wedge (\neg q \vee p) & \| & (A \vee B) \Leftrightarrow (B \vee A) \\
(p \vee q) \wedge (p \vee \neg q) & \| & (A \vee B) \wedge (A \vee C) \Leftrightarrow A \vee (B \wedge C) \\
p \vee (q \wedge \neg q) & \| & (A \wedge \neg A) \Leftrightarrow \bot, A \vee \bot \Leftrightarrow A \\
p & &
\end{array}$$

■

In the context of computation, we consider each application of a rule as *one computational step*. That is, in the example above we arrive at the second line when starting from the first line in three computational steps (two for replacing the implication by disjunction and one for eliminating the double negation).

Thanks to identities like $A \rightarrow B \Leftrightarrow \neg A \vee B$ or $A \wedge B \Leftrightarrow \neg(\neg A \vee \neg B)$, we know that we can express all logical formulas with just a minimum set of symbols: variables (like p, q, etc), parentheses,

$\neg$, and $\vee$, for example. As a follow-up question, we can then ask ourselves, what is the least number of substitution (or transformation) rules that we need?

Church discovered that we can express all computations with an astoundingly small number of symbols and rules, giving rise to his Lambda-calculus.

Two kinds of transformations. There are two different kinds of transformations on terms: those with “computational power” and those without.

In the simplest case we can just change the name of variables. For instance, $f(x) = x^2$ and $f(a) = a^2$ both denote the exact same function. Changing the name of the parameter from x to a has no computational effect whatsoever. Such a renaming or transformation without any “real” effect is an α-transformation.

The more interesting kinds of transformations are those that truly reduce, simplify, or otherwise alter the *structure of a term*. When replacing 7×8 by 56, or $p \vee p$ by p, we actually perform a computational step. These are β-transformations.

In the context of Lambda-calculus we are mostly interested in *α-equivalence* and *β-reduction*. Two terms are α-equivalent if they have the exact same structure and merely differ in the choice of variable names. There is absolutely no difference in stating $(x + y)^2 = x^2 + 2xy + y^2$ vs. $(a + b)^2 = a^2 + 2ab + b^2$. For all intents and purposes, we will consider α-equivalent terms as just fully equivalent. A β-reduction, on the other hand, means a structural change, something like a simplification of a term. Two terms that are β-equivalent can be reduced to the same term. They ‘mean’ the same thing, but in different forms. Hence, 3×7 and $10 + 11$ are β-equivalent, because both terms can be reduced to 21 (the so called β-normal form). Naturally, 3×7 is also β-equivalent to 21, of course. And all terms that are α-equivalent are also β-equivalent.

Functions and algorithms. Church’s terms in λ-calculus are seemingly modelled after functions and function-application. The syntax of λ-terms is indeed often used to actually write a ‘nameless’ function where $f(x) = x^2$ could be written as $f = \lambda x.x^2$ (with the overall syntax $\lambda\text{parameter}.\text{formula}$). This is particularly neat when writing something like $(\lambda x.x^2 - 1)(5)$, which evaluates to 24.

However, Church’s λ-terms are not actually meant to express *functions*, but rather *algorithms*! For instance, recall that $f_1(x) = x^2 - 1$ and $f_2(x) = (x - 1)(x + 1)$ are the same function (as mathematical objects), while they express different algorithms for this function. Similarly, the following two programs implement the same function $p(x, y) = x^y$, although there are clear algorithmic differences between the two implementations.

```
fun power x y =
  if y = 0
    then 1
    else if y mod 2 = 0
      then power (x*x) (y div 2)
      else x * power (x*x) (y div 2)
```

```
fun power x y =
  let
    val i = ref y and z = ref 1
  in
    while !i > 0 do
      (i := !i - 1; z := !z * x);
    !z
  end;
```

In the chapters so far we have discussed the possibilities of implementing (computable) partial functions as register machines. In this chapter, we now consider the implementation of these partial functions as λ-terms.

9.3 λ-Terms

λ-terms are built up from a given, countable collection of *variables* $x, y, z, \ldots$ by two operations for forming λ-terms:

- λ-*abstraction*: $(\lambda x.M)$ (where x is a variable and M is a λ-term),
- *application*: $(M\,M')$ (where M and M' are λ-terms).

Instead of $\lambda x.\lambda y.M$, we often simply write $\lambda x\,y.M$ as a shorthand notation. Another notation that you might encounter is "$\mathrm{let}\ x = N\ \mathrm{in}\ M$", which stands for $(\lambda x.M)\,N$.

Example 19: A random example of a λ-term is $\big(\lambda y.(x\,y)\big)x'$. The entire term is an application, where we have the λ-abstraction $\lambda y.(x\,y)$ and apply it to the variable x'. If we perform a *reduction*, we replace all occurences of the parameter y in the body $(x\,y)$ of the λ-abstraction by the term x'. Hence:

$$\underbrace{\big(\lambda y.\,\underbrace{(x\,y)}_{\text{body}}\big)}_{\text{abstraction}}x' \quad\longrightarrow\quad x\,x'$$

■

Example 20: The λ-term $\lambda x.x$ stands for the *identity* and is usually abbreviated as I. For any λ-term t, the term $\mathrm{I}\,t = (\lambda x.x)t$ reduces to t.

The λ-term $\lambda x\,y.x$ returns only the first of two given terms, while $\lambda x\,y.y$ returns the second of two given terms. It might look like $\lambda xy.y\,x$ swaps two terms, but keep in mind that $y\,x$ actually stands for the *application* of y to x. ■

Example 21: In order to *use* λ-calculus, we need to find ways to *encode* data as λ-terms. As a first example, we encode pairs (x, y) as λ-terms.

It might seem straight-forward to just write the pair (x, y) as λ-term $(x\,y)$. However, this actually denotes an application and the computer performing the reduction is free to reduce this pair (by applying x to y). To avoid this, we therefore rather encode the pair as $\lambda f.f\,x\,y$.

With the pair encoded as $\lambda f.f\,x\,y$, we can now plug in a term like $\lambda x\,y.x$ or $\lambda x\,y.y$ to retrieve the first or second term in our pair, i.e. $(\lambda f.f\,A\,B)(\lambda x\,y.x)$ reduces to A. The following is a more complicated version of the same and also reduces to A. Can you do the necessary steps of reduction?

$$\big(\lambda p.(\mathrm{I}\,p)\,(\lambda s.\lambda t.s)\big)(\lambda f.f\,A\,B)$$

■

Bound and free variables. In $\lambda x.M$ we call x the *bound variable* and M the *body* of the λ-abstraction. An occurence of x in a λ-term M is called:

– **binding** if it is in between λ and the dot, e.g., $(\lambda\mathbf{x}.yx)x$;

– **bound** if it is in the body of a binding occurence of x, e.g., $(\lambda x.y\mathbf{x})x$;

– **free** if it is neither binding nor bound, e.g., $(\lambda x.yx)\mathbf{x}$.

For a λ-term M we can define the sets of *free* and *bound* variables $FV(M)$ and $BV(M)$, respectively:

$$\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(\lambda x.M) & = & FV(M) \setminus \{x\} \\
FV(M\,N) & = & FV(M) \cup FV(N) \\
BV(x) & = & \emptyset \\
BV(\lambda x.M) & = & BV(M) \cup \{x\} \\
BV(M\,N) & = & BV(M) \cup BV(N)
\end{array}$$

We write $x\#M$ to mean that x does not occur in the term M.

If $FV(M) = \emptyset$, M is called a *closed term* or *combinator*. For instance, the identity $\mathrm{I} = \lambda x.x$ is a combinator, as is $\mathrm{K} = \lambda x\,y.x$ (why is K a combinator? Isn't the y a free variable here?).

Terms and functions*. The identity term $I = \lambda x.x$ can be applied to any λ-term, including itself, i.e. $\mathrm{I}\,\mathrm{I}$ is a perfectly legal term that obviously reduces to I. This example demonstrates once again that λ-terms are not functions (at least not in the mathematical sense).

Modern mathematics understands functions as mapping from a set A to a set B. A function f is thus a subset of $A \times B$. Applying a function f to itself (i.e. $f(f)$) would require that f be also an element of A. This quickly leads to very strange behaviour.

However, you can think of a term such as I as *representing* a function (together with reduction) from λ-terms to λ-terms (to be more precise, it is actually operating on equivalence classes of λ-terms). But note that you need *reduction* for this to make sense.

Consider, for instance, 3×3 matrices, which we can understand as transformations of 3-dimensional space (you will probably have encountered them in computer graphics). Obviously you can multiply such matrices and any matrix *together with the multiplication operation* gives rise to a function that transforms all matrices (including itself). However, the underlying matrix itself is neither an operation or function, only the combination with an operator makes it an actual function. For λ-terms, it is quite analogous in that a λ-term itself is never a function, but together with reduction, it may be thought of as a function.

9.4 α-Equivalence

The variable x in the abstraction $\lambda x.M$ acts as a placeholder to be replaced by a λ-term. The name of such a bound variable is immaterial: if $M' = M[x'/x]$ is the result of taking M and changing all occurrences of x to some variable $x'\#M$, then $\lambda x.M$ and $\lambda x'.M'$ both represent the same algorithm (just like $f(x) = x^2$ and $f(z) = z^2$ represent the exact same function).

For example, $\lambda x.x$ and $\lambda y.y$ both represent the identity combinator.

Such 'structurally' equivalent λ-terms that only differ in the names of bound variables are called α-equivalent. Changing the name of a bound variable is also known as α-conversion and is the simplest form of a substitution. Note that any α-conversion must make sure that distinct variable names remain distinct: $\lambda x.x\,y$ and $\lambda y.y\,y$. are *not* α-equivalent!

α-Equivalence $M =_\alpha M'$ is the binary relation inductively generated by the rules:

$$\frac{}{x =_\alpha x} \qquad \frac{z\#(M\,N) \quad M[z/x] =_\alpha N[z/y]}{\lambda x.M =_\alpha \lambda y.N} \qquad \frac{M =_\alpha M' \quad N =_\alpha N'}{M\,N =_\alpha M'\,N'}$$

where $M[z/x]$ is M with all occurences of x replaced by z. Some authors write this also as $M[x \mapsto z]$ or $[z/x]M$.

Example 22: Note that x is bound twice in this example, thus introducing completely different variables with the same name. When doing α-conversion, take care not to replace variables that only look alike!

$$\lambda x\,y.x\,(\lambda x.x\,y) =_\alpha \lambda z\,y.z\,(\lambda x.x\,y) =_\alpha \lambda z\,t.z\,(\lambda x.x\,t)$$

■

Example 23:

$$\begin{array}{crcl}
 & \lambda x.(\lambda x x'.x)x' & =_\alpha & \lambda y.(\lambda x x'.x)x' \\
\text{because} & (\lambda z x'.z)x' & =_\alpha & (\lambda x x'.x)x' \\
\text{because} & (\lambda z x'.z =_\alpha \lambda x x'.x & \text{and} & x' =_\alpha x' \\
\text{because} & \lambda x'.u =_\alpha \lambda x'.u & \text{and} & x' =_\alpha x' \\
\text{because} & u =_\alpha u & \text{and} & x' =_\alpha x'
\end{array}$$

■

Fact: $=_\alpha$ is an equivalence relation (reflexive, symmetric, and transitive).

We do not care about the particular names of bound variables, just about the distinctions between them. So α-equivalence classes of λ-terms are more important than λ-terms themselves. Textbooks (and these notes) suppress any notation for α-equivalence classes and refer to an equivalence class via a representative λ-term (look for phrases like 'we identify terms up to α-equivalence' or 'we work up to α-equivalence'). For implementations and computer-assisted reasoning, there are various devices for picking canonical representatives of α-equivalence classes (e.g. de Bruijn indexes, graphical representations, etc.).

9.5 β-Reduction

9.5.1 Substitution

If we replace variables not only by other variables, but by any λ-term M, we arrive at the general concept of substitution, as determined by the followig rules:

$$\begin{array}{rcll} x[M/x] & = & M & \\ y[M/x] & = & y & \text{if } y \neq x \\ (\lambda y.N)[M/x] & = & \lambda y.(N[M/x]) & \\ (N_1\, N_2)[M/x] & = & N_1[M/x]\, N_2[M/x] & \end{array}$$

The side-condition $y\#(M\,x)$ (y does not occur in M and $y \neq x$) makes substitution 'capture-avoiding', i.e. making sure that two distinct variables do not suddenly coincide and become one (see above).

Example 24: In this example, we replace the variable y by the term $M = \lambda z.z\,z$:

$$(\lambda x.y\,x)[(\lambda z.z\,z)/y] = \big(\lambda x.(\lambda z.z\,z)\,x\big)$$

■

Example 25: If $x \neq y \neq z \neq x$, then:

$$(\lambda y.x)[y/x] =_\alpha (\lambda z.x)[y/x] = \lambda z.y$$

Note the $=_\alpha$: the α is a hint that we might have had to rename some of the variables so as to avoid capture. Indeed, we cannot replace x by y in $\lambda y.x$ because the 'new' y would be captured by the binding y. In order to avoid that, we first perform an α-conversion. ■

The substitution $N \mapsto N[M/x]$ induces in fact a totally defined function from the set of α-equivalence classes of λ-terms to itself.

9.5.2 Reduction

Just as you can apply a function to an argument, you can apply a λ-abstraction to another λ-term. The notation $(\lambda x.M)\,a$ is thus intended to mean that all variables x in the term M shall be replaced by a, i.e. $(\lambda x.M)\,a = M[a/x]$. This substitution is at the heart of β-reduction.

The natural notion of computation for λ-terms is thus given by stepping from a *β-redex* $(\lambda x.M)\,N$ to the corresponding *β-reduct* $M[N/x]$ ('*redex*' is short for 'reducible expression'). This is sometimes also called *contraction* of $(\lambda x.M)\,N$ to $M[N/x]$.

One-step β-reduction, $M \to M'$. The one-step β-reduction performs a single contraction, i.e. it replaces one λ-subterm with the pattern $(\lambda x.M)\,N$ by $M[N/x]$, leading to the following rules:

$$\frac{}{(\lambda x.M)N \to M[N/x]} \qquad \frac{M \to M'}{\lambda x.M \to \lambda x.M'} \qquad \frac{M \to M'}{M\,N \to M'\,N}$$

$$\frac{M \to M'}{N\,M \to N\,M'} \qquad \frac{N =_\alpha M \quad M \to M' \quad M' =_\alpha N'}{N \to N'}$$

Some authors write $\to_\beta$, but we omit the β as long as it is clear from context.

Example 26: A typical λ-term contains more than one β-redex that can be reduced. However, β-reduction is *confluent*, which is to say that if $A \to B$ and $A \to C$, then there is another λ-term D such that $B \to D$ and $C \to D$:

$$\begin{tikzcd}
& ((\lambda y.\lambda z.z)u)y \arrow[dr] & & \\
(\lambda x.xy)((\lambda y.\lambda z.z)u) \arrow[ur] \arrow[dr] & & (\lambda z.z)y \arrow[r] & y \\
& (\lambda x.xy)(\lambda z.z) \arrow[ur] & &
\end{tikzcd}$$

■

Example 27: Example of the 'up to α-equivalence' aspect of reduction:

$$(\lambda x.\lambda y.x)y =_\alpha (\lambda x.\lambda z.x)y \to \lambda z.y$$

The α-conversion is important here, as a reduction to $\lambda y.y.$ would clearly be wrong (why?). ■

Many-step β-reduction, $M \twoheadrightarrow M'$. It is convenient to combine several single steps of β-reduction and perform them as 'one'. This leads to the many-step or 'big step' reduction:

$$\frac{M =_\alpha M'}{M \twoheadrightarrow M'} \qquad \frac{M \to M'}{M \twoheadrightarrow M'} \qquad \frac{M \twoheadrightarrow M' \quad M' \to M''}{M \twoheadrightarrow M''}$$

As before, some authors write $\twoheadrightarrow_\beta$. Another notation you might encounter is $\rhd_\beta$.

Example 28:

$$(\lambda x.xy)\big((\lambda yz.z)u\big) \twoheadrightarrow y \qquad (\lambda x.\lambda y.x)y \twoheadrightarrow \lambda z.y$$

■

9.5.3 β-Conversion $M =_\beta N$

Informally $M =_\beta N$ holds if N can be obtained from M by performing zero or more steps of α-equivalence, β-reduction, or β-expansion (the inverse of a β-reduction).

Example 29: $u\big((\lambda xy.vx)y\big) =_\beta (\lambda x.ux)(\lambda x.vy)$ because $(\lambda x.ux)(\lambda x.vy) \to u(\lambda x.vy)$ and so we have:

$$\begin{aligned} u\big((\lambda xy.vx)y\big) &=_\alpha u\big((\lambda xy'.vx)y\big) \\ &\to u(\lambda y'.vy) \\ &=_\alpha u(\lambda x.vy) \\ &\leftarrow (\lambda x.ux)(\lambda x.vy) \end{aligned}$$

■

β-conversion $M =_\beta N$ is the binary relation inductively generated by the rules:

$$\frac{M =_\alpha M'}{M =_\beta M'} \qquad \frac{M \to M'}{M =_\beta M'} \qquad \frac{M =_\beta M'}{M' =_\beta M} \qquad \frac{M =_\beta M' \quad M' =_\beta M''}{M =_\beta M''}$$

$$\frac{M =_\beta M'}{\lambda x.M =_\beta \lambda x.M'} \qquad \frac{M =_\beta M' \quad N =_\beta N'}{M\,N =_\beta M'\,N'}$$

Theorem 28: Church-Rosser Theorem

$\twoheadrightarrow$ is *confluent*, that is, if $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$, then there exists M' such that $M_1 \twoheadrightarrow M'$ and $M_2 \twoheadrightarrow M'$.

$$\begin{tikzcd} & M_1 \arrow[dr, two heads] & \\ M \arrow[ur, two heads] \arrow[dr, two heads] & & M' \\ & M_2 \arrow[ur, two heads] & \end{tikzcd}$$

Theorem 29: Corollary

$M_1 =_\beta M_2$ iff there is an M such that $M_1 \twoheadrightarrow M \twoheadleftarrow M_2$. In other words: M_1 and M_2 are β-equivalent if both can be reduces to a common term M'.

Proof. $=_\beta$ satisfies the rules generating $\twoheadrightarrow$; so $M \twoheadrightarrow M'$ implies $M =_\beta M'$. Thus if $M_1 \twoheadrightarrow M$ and $M_2 \twoheadrightarrow M$, then $M_1 =_\beta M =_\beta M_2$ and so $M_1 =_\beta M_2$.

Conversely, the relation $\{(M_1, M_2) \mid \exists.M_1 \twoheadrightarrow M \wedge M_2 \twoheadrightarrow M\}$ satisfies the rules generating $=_\beta$: the only difficult case is closure of the relation under transitivity and for this we use the Church-Rosser theorem. Hence $M_1 =_\beta M_2$ implies that there is an M such that $M_1 \twoheadrightarrow M$ and $M_2 \twoheadrightarrow M$.

□

9.6 β-Normal Forms

Definition 30: β-Normal Form

A λ-term is in *β-normal form (β-nf)* if it contains no β-redexes (no sub-terms of the form $(\lambda x.M)M')$. M *has β-normal form* N if $M =_\beta N$ with N a β-normal form.

Note that if N is a β-normal form and $N \twoheadrightarrow N'$, then it must be that $N =_\alpha N'$ Hence, if $N_1 =_\beta N_2$ with N_1 and N_2 both β-normal forms, then $N_1 =_\alpha N_2$. Because if $N_1 =_\beta N_2$, then by Church-Rosser $N_1 \twoheadrightarrow M' \twoheadleftarrow N_2$ for some M', so $N_1 =_\alpha M' =_\alpha N_2$.

So, the β-normal form of M is unique up to α-equivalence if it exists.

Non-termination. Some λ-terms have no β-normal form. Every attempt to reduce such a λ-term introduces another β-redex, leading to an infinite sequence. With λ-terms as representations of algorithms, this means that the algorithm never terminates.

Example 30: $\Omega = (\lambda x.xx)(\lambda x.xx)$ has no β-normal form, as it satisfies:

- $\Omega \to (xx)[(\lambda x.xx)/x] = \Omega$, i.e. $\Omega \to \Omega$;
- $\Omega \twoheadrightarrow M$ thus implies $\Omega =_\alpha M$.

So there is no β-normal form N such that $\Omega =_\beta N$. ■

The picture is not always that clear: in fact, a term can possess both a β-normal form and infinite chains of reductions from it. In other words: depending on which β-redex you reduce first, you either arrive at a β-normal form, or you are caught in an 'infinite loop'.

Example 31: $(\lambda \text{x}.y)\boldsymbol{\Omega}$ reduces to the β-normal form y. But if we reduce Ω first (rather than the λ abstraction on the left), we get:

$$(\lambda x.y)\boldsymbol{\Omega} \to (\lambda x.y)\boldsymbol{\Omega} \to (\lambda x.y)\boldsymbol{\Omega} \to \dots$$

■

In order to address this ambiguity, we introduce *normal-order reduction*. Normal-order reduction is a deterministic strategy for reducing λ-terms: reduce the 'left-most, outer-most' redex first.

A redex is in *head position* in a λ-term M if M takes the form

$$\lambda x_1 \ldots \lambda x_n . \underline{(\lambda x . M') \, M_1} \, M_2 \ldots M_m \qquad (n \geq 0, m \geq 1)$$

where the redex is the underlined subterm. A λ-term is said to be in *head normal form* if it contains no redex in head position, in other words takes the form:

$$\lambda x_1 \ldots \lambda x_n . x \, M_1 \, M_2 \ldots M_m \qquad (mn \geq 0)$$

Normal order reduction first continually reduces redexes in head position; if that process terminates then one has reached a head normal form and one continues applying head reduction in the subterms $M_1, M_2, \ldots$ from left to right.

Fact: normal-order reduction of M always reaches the β-normal form of M if it possesses one.

9.7 Applications*

Lambda calculus rarely appears in its pure form. However, various elements and ideas can be found in a wealth of other areas, fields, and applications.

For instance, in constrast to functions in mathematics, λ-abstractions do not require a 'name' (in $f(x) = x + 1$, the f would be the name of the function), which gave rise to *lambda functions* in various programming languages as unnamed (anonymous) functions. In Python, this is written, e.g., as `lambda x: x+1` for $x \mapsto x + 1$. But keep in mind that such lambda functions are not the same thing as the λ-abstractions presented in this chapter.

In formal settings, lambda calculus is often enhanced or mixed with other calculi or notations. You might therefore well encounter something like $\lambda x . x^2 - 1$, although the expression $x^2 - 1$ is clearly not a λ-term in the strict sense. In this regard, $\lambda x . x^2 - 1$ serves about the same notational purpose as $x \mapsto x^2 - 1$.

9.7.1 Compiler Transformations

In practice, most computing machines operate as finite state machines (or simplified Turing or register machines if you will). Nonetheless, the ideas of α-conversion and β-reduction play essential roles in compilers and their various steps of optimisations.

Constant folding is an optimisation technique, where symbols with a fixed and known value are replaced by their respective value. This may eliminate the need for the constant symbols, and open up further optimisations. Consider the following code excerpt:

```
int N = 3;
...
for (int i = 0; i < N; i++) { foo(); }
```

If the compiler knows that N always has a value of 3, it may replace it in the for loop below. This, in turn reveals that the loop's body will be iterated exactly three times, without making use of the loop variable i. The compiler could thus completely unroll the loop, getting rid of conditional branching:

```
...
foo(); foo(); foo();
```

Note that these transformations are done statically as transformations of the program code, i.e. without executing any part of the program. And you could imagine that an outstanding compiler might even be able to reduce a simple program to a point where the output can be derived without ever 'running' the program.

α-conversion, on the other hand, is often found in obfuscation, but also in compilers that minimise JavaScript files by shortening all names to a bare minimum.

9.7.2 Macros

In many systems, *macros* capture the essence of β-reduction quite neatly. In contrast to functions/subroutines/procedures in a programming language, a macro is not executed at run time, but rather expanded (i.e. substituted) during compile time to create the code for later execution.

Knuth's TeX. The typesetting system TeX by Donald Knuth (and its better known variant LaTeX) looks superficially similar to markup languages like HTML or Markdown. The text is interspersed with instructions on how to format the text, e.g.:

`Text in \textit{italics} and \textbf{boldface}.`

However, TeX offers a full macro system that is rather close to λ-calculus as discussed here.

You might define a macro 'double' as follows, where `#1` plays the role of the binding variable:

`\def\double#1{#1#1}`

As a λ-term, we might write this as $\mathrm{double} \triangleq \lambda x.x\,x$. If you then use it inside your text as in, e.g.: '`\double{Wonderful! }`', you get '*Wonderful! Wonderful!* '.

The purely substitutional aspect can lead to surprises for people expecting a behaviour more in line with functions. Consider the following macro that takes two arguments and swaps their order:

`\def\flip#1#2{#2#1}`

What output do you then expect from the following text?

`\flip{Hey!}\double{Ho!}`

As the system operates purely on substitution, it does *not* 'evaluate' the `\double` macro to get the second argument for `\flip`. Rather, the two arguments for `\flip` are `{Hey!}` and `\double`. The eventual output thus is: '*Hey!Hey!Ho!*'.

In constrast to compilers as discussed above, the objective of TeX is not to create a runnable program, but almost exclusively to 'expand' the macros in the text. In other words, TeX is a system that relies heavily on β-reduction.

10 Lambda-Definable Functions

Comprising only abstractions and applications, λ-terms seem deceptively simple. In this chapter we will demonstrate that λ-terms suffice to express all natural numbers as well as every computable function.

10.1 Encoding Data in λ-Calculus

Computation in λ-calculus is given by β-reduction. To relate this to register/Turing machine computation, or to partial recursive functios, we first have to see how to encode numbers, pairs, lists, etc. as λ-terms. We will use the original encoding of numbers due to Church (although others exist).

Definition 31: Church's Numerals

$$\begin{aligned}
\underline{0} &\triangleq \lambda fx.x \\
\underline{1} &\triangleq \lambda fx.fx \\
\underline{2} &\triangleq \lambda fx.f(fx) \\
&\cdots \\
\underline{n} &\triangleq \lambda fx.\, \underbrace{f(\cdots(f}_{n\times}\,x)\cdots)
\end{aligned}$$

Notation:

$$M^0\,N \triangleq N \qquad M^1\,N \triangleq M\,N \qquad M^{n+1}\,N \triangleq M\,(M^n\,N)$$

so we can write $\underline{n}$ as $\lambda fx.f^n x$ and we have $\underline{n}\,M\,N =_\beta M^n\,N$. Note that M^n alone has no meaning in this context. In particular $M\,M\,M = M^2\,M \neq M^3$.

In addition to numbers, we also want to encode boolean values, test-for-zero and pairs (tuples).

Representing booleans. Like Church numerals, the boolean values for *true* and *false* are λ-abstractions that may be applied to a λ-term. Note, in particular, that **False** is actually α-equivalent to $\underline{0}$.

$$\begin{aligned}\mathbf{True} &\triangleq \lambda x\,y.x\\ \mathbf{False} &\triangleq \lambda x\,y.y\\ \mathbf{If} &\triangleq \lambda f\,x\,y.f\,x\,y\end{aligned}$$

These definitions behave as expected under β-reduction and satisfy:

$$\mathbf{If}\,\mathbf{True}\,M\,N =_\beta \mathbf{True}\,M\,N =_\beta M \qquad \mathbf{If}\,\mathbf{False}\,M\,N =_\beta \mathbf{False}\,M\,N =_\beta N$$

Representing test-for-zero. Testing whether a value is zero (i.e. $\underline{0} = \lambda f\,x.x$) is based on the idea that $\underline{0}xy = y$, while for $n \neq 0$ we have: $\underline{n}xy = x^n y$.

$$\mathbf{Eq_0} \triangleq \lambda x.x\ (\lambda y.\mathbf{False})\ \mathbf{True}$$

For $\underline{0}$ we simply get:

$$\mathbf{Eq_0}\,\underline{0} =_\beta \underline{0}(\lambda y.\mathbf{False})\,\mathbf{True} =_\beta \mathbf{True}$$

For any number $n \in \mathbb{N}$, on the other hand, $\mathbf{Eq_0}$ satisfies:

$$\begin{aligned}\mathbf{Eq_0}\,\underline{n+1} &=_\beta \underline{n+1}(\lambda y.\mathbf{False})\,\mathbf{True}\\ &=_\beta (\lambda y.\mathbf{False})^{n+1}\mathbf{True}\\ &=_\beta (\lambda y.\mathbf{False})\big((\lambda y.\mathbf{False})^{n}\ \mathbf{True}\big)\\ &=_\beta \mathbf{False}\end{aligned}$$

Representing ordered pairs. As already mentioned earlier, we cannot write a pair simply as $(A\,B)$ because this would mean A applied to B. By writing the pair as an abstraction $\mathbf{Pair}\,A\,B = \lambda f.f\,A\,B$, we prevent any such interpretation as this is equivalent to $\lambda f.(f\,A)\,B$, i.e. we would have to reduce $f\,A$ first (rather than $A\,B$), which we can only do once we have a specific λ-term for f.

$$\begin{aligned}\mathbf{Pair} &\triangleq \lambda x\,y\,f.f\,x\,y\\ \mathbf{Fst} &\triangleq \lambda f.f\,\mathbf{True}\\ \mathbf{Snd} &\triangleq \lambda f.f\,\mathbf{False}\end{aligned}$$

For the definition of **Fst** and **Snd**, we make use of the booleans as λ-terms that choose the first or the second of two terms, respectively. This then satisfies:

$$\begin{aligned}\mathbf{Fst}\ (\mathbf{Pair}\,M\,N) &=_\beta \mathbf{Fst}\ (\lambda f.f\,M\,N)\\ &=_\beta (\lambda f.f\,M\,N)\ \mathbf{True}\\ &=_\beta \mathbf{True}\,M\,N\\ &=_\beta M\end{aligned}$$

The case for **Snd** is analogous, of course:

$$\mathbf{Snd}\ (\mathbf{Pair}\,M\,N) =_\beta \cdots =_\beta \mathbf{False}\,M\,N =_\beta N$$

10.2 λ-Definable Functions

Definition 32: λ-Definable Functions

$f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is λ-*definable* if there is a closed λ-term F that *represents* it: for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$:

- if $f(x_1, \ldots, x_n) = y$, then $F\,\underline{x_1} \cdots \underline{x_n} =_\beta \underline{y}$;
- if $f(x_1, \ldots, x_n)\uparrow$, then $F\,\underline{x_1} \cdots \underline{x_n}$ has no β-normal form.

The second condition (requiring that $F\,\underline{x_1} \cdots \underline{x_n}$ has no β-normal form if $f(x_1, \ldots, x_n$ is not defined) can make it quite tricky to find a λ-term representing a non-total function. See the paragraph on composition below for an example of how to deal with this.

Example 32: The addition is λ-definable because it is represented by $P \triangleq \lambda x_1 x_2.\lambda x.x_1 f(x_2 f x)$:

$$\begin{aligned} P\,\underline{m}\,\underline{n} &=_\beta \lambda f\,x.\underline{m}\,f\,(\underline{n}\,f\,x) \\ &=_\beta \lambda f\,x.\underline{m}\,f\,(f^n\,x) \\ &=_\beta \lambda f\,x.f^m\,(f^n\,x) \\ &=_\beta \lambda f\,x.f^{m+n}\,x \\ &=_\beta \underline{m+n} \end{aligned}$$

■

Theorem 33:

A partial function is computable if and only if it is λ-definable.

We already know that *register machine computable*, *Turing computable* and *partial recursive* are equivalent. Using this, we can break the proof of the theorem into two parts:

- every partial recursive function is λ-definable;
- λ-definable functions are register machine computable.

However, we will focus on the first part only here, as the second part is rather boring and tedious.

We start by concentrating on total functions. First, let us see why the elements of PRIM (primitive recursive functions) are λ-definable.

10.2.1 Representing Basic Functions

The three basic function types $\mathrm{proj}^n_i, \mathrm{zero}^n, \mathrm{succ}$ can be represented quite naturally in λ-calculus:

- **Projection** functions, $\mathrm{proj}^n_i \in \mathbb{N}^n \to \mathbb{N}$ with $\mathrm{proj}^n_i(x_1, \ldots, x_n) \triangleq x_i$ are represented by:

$$\lambda x_1 \ldots x_n . x_i$$

- **Constant** functions with value 0, $\mathrm{zero}^n \in \mathbb{N}^n \to \mathbb{N}$ with $\mathrm{zero}^n(x_1, \ldots, x_n) \triangleq 0$ are represented by:

$$\lambda x_1 \ldots x_n . \underline{0}$$

- **Successor** function, $\mathrm{succ} \in \mathbb{N} \to \mathbb{N}$ with $\mathrm{succ}(x) \triangleq x + 1$ is represented by:

$$\mathbf{Succ} \triangleq \lambda x_1\, f\, x . f (x_1\, f\, x)$$

  since

$$\begin{aligned} \mathbf{Succ}\,\underline{n} &=_\beta \lambda f\, x . f\,(\underline{n}\, f\, x) \\ &=_\beta \lambda f\, x . f\,(f^n x) \\ &= \lambda f\, x . f^{n+1} x \\ &= \underline{n+1} \end{aligned}$$

10.2.2 Representing Composition

If the total functions $f \in \mathbb{N}^n \to \mathbb{N}$ and $g_1, \ldots, g_n \in \mathbb{N}^m \to \mathbb{N}$ are represented by F and $G_1, \ldots, G_n$, respectively, then their composition $f \circ (g_1, \ldots, g_n) \in \mathbb{N}^m \to \mathbb{N}$ is represented simply by:

$$\lambda x_1 \ldots x_m . F \big(G_1\, x_1 \ldots x_m\big) \ldots \big(G_n\, x_1 \ldots x_m\big)$$

because

$$\begin{aligned} F\,\big(G_1\,\underline{a_1} \ldots \underline{a_m}\big) \ldots \big(G_n\,\underline{a_1} \ldots \underline{a_m}\big) &=_\beta F\,\underline{g_1(a_1, \ldots, a_m)} \ldots \underline{g_n(a_1, \ldots, a_m)} \\ &=_\beta \underline{f\big(g_1(a_1, \ldots, a_m), \ldots, g_n(a_1, \ldots, a_m)\big)} \\ &= \underline{f \circ (g_1, \ldots, g_n)(a_1, \ldots, a_m)} \end{aligned}$$

This does not necessarily work for *partial* functions. For instance, the totally undefined function $u \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is represented by $\mathbf{U} \triangleq \lambda x_1 . \Omega$ and $\mathrm{zero}^1 : \mathbb{N} \to \mathbb{N}$ is represented by $\mathbf{Z} \triangleq \lambda x_1 . \underline{0}$. But $\mathrm{zero}^1 \circ u$ is not represented by $\lambda x_1 . \mathbf{Z}\,(\mathbf{U}\, x_1)$, because $\mathrm{zero}^1 \circ u(n)$ is undefined whereas:

$$(\lambda x_1 . \mathbf{Z}\,(\mathbf{U}\, x_1))\ \underline{n} =_\beta \mathbf{Z}\,\Omega =_\beta \underline{0}.$$

The composition of partial functions must make sure that each G_i is fully reduced even if it does not contribute to the overall result otherwise. One way of doing this is, e.g.:

$$(G\,a_1\,\ldots\,a_m)\,I\,I$$

If $(G\,a_1\,\ldots\,a_m)$ evaluates to a Church numeral $\underline{n_G}$, we have:

$$\underline{n_G}\,I\,I =_\beta I^{n_G}\,I = I$$

But if $(G\,a_1\,\ldots\,a_m)$ has no β-normal form, then neither does $(G\,a_1\,\ldots\,a_m)\,I\,I$.

10.2.3 Representing Predecessor

We want a λ-term $\mathbf{Pred}$ that satisfies:

$$\mathbf{Pred}\,\underline{n+1} =_\beta \underline{n} \qquad \mathbf{Pred}\,\underline{0} =_\beta \underline{0}$$

Our strategy is to take a pair $(0,0)$ together with mapping $f : (a,b) \mapsto (a+1,a)$. Repeatedly applying f to the pair yields pairs of the form $(n, n-1)$, where n in the first field is the number of iterations and the second field naturally contains the predecessor of n. The λ-term that represents this is as follows:

$$\mathbf{Pred} \triangleq \lambda y\,f\,x.\mathbf{Snd}\big(y\,(G\,f)\,(\mathbf{Pair}\,x\,x)\big)$$

where

$$G \triangleq \lambda f\,p.\mathbf{Pair}\,\big(f\,(\mathbf{Fst}\,p)\big)\,\left(\mathbf{Fst}\,p\right).$$

Note the $y\,(G\,f)\,P$ in the $\mathbf{Pred}$ term above: whenever y reduces to a Church numeral $\underline{n_y}$, this becomes $(G\,f)^{n_y}\,P$, i.e.:

$$\underbrace{(G\,f)\,\Big(\cdots\,(G\,f)\,\big((G\,f)}_{n\times}\,(\mathbf{Pair}\,x\,x)\big)\cdots\Big)$$

10.3 Primitive Recursion

Recall primitive recursion:

Theorem 34:

Given $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \in \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$, there is a unique $h \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ satisfying

$$\begin{cases} h(\vec{x}, 0) & \equiv \quad f(\vec{x}) \\ h(\vec{x}, y+1) & \equiv \quad g\big(\vec{x}, y, h(\vec{x}, y)\big) \end{cases}$$

for all $\vec{x} \in \mathbb{N}^n$ and $y \in \mathbb{N}$.

We write $\rho^n(f,g)$ for h and call it the partial function *defined by primitive recursion* from f and g.

If $f \in \mathbb{N}^n \to \mathbb{N}$ is represented by a λ-term F and $g \in \mathbb{N}^{n+2} \to \mathbb{N}$ is represented by a λ-term G, we want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying:

$$\left\{ \begin{array}{lcl} h(\vec{a}, 0) & \equiv & f(\vec{a}) \\ h(\vec{a}, b+1) & \equiv & g\big(\vec{a}, b, h(\vec{a}, b)\big) \end{array} \right.$$

or equivalently:

$$h(\vec{a}, b) = \left\{ \begin{array}{ll} f(\vec{a}) & \text{if } b = 0 \\ g\left(\vec{a}, b-1, h(\vec{a}, b-1)\right) & \text{else} \end{array} \right.$$

That is, we want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying

$$h = \Phi_{f,g}(h)$$

where $\Phi_{f,g} \in \big(\mathbb{N}^{n+1} \to \mathbb{N}\big) \to \big(\mathbb{N}^{n+1} \to \mathbb{N}\big)$ is given by:

$$\Phi_{f,g}(h)(\vec{a}, b) = \left\{ \begin{array}{ll} f(\vec{a}) & \text{if } b = 0 \\ g\left(\vec{a}, b-1, h(\vec{a}, b-1)\right) & \text{else} \end{array} \right.$$

Our strategy is to first show that $\Phi_{f,g}$ is λ-definable and then to show that we can solve *fixed point equations* $X = M\,X$ up to β-conversion in the λ-calculus.

10.3.1 Curry's Fixed Point Combinator $\mathbf{Y}$

You are, of course, already familiar with functional equations. They are particularly useful for recursive functions such as, e.g.: $f(n) = nf(n-1)$ or $f(n+2) = f(n) + f(n+1)$. Similarly, you might also have seen differential equations. What all these have in common is that you do not have an *explicit form* for the function f, but you know something about its *behaviour*. The question then is, whether the given behaviour lets you determine the entire function in question, or even find a closed explicit form for it.

During the last few centuries, mathematics has addressed this general challenge by changing the concept of a function so that any mapping from one set to another can be a function—irrespective of whether there is an explicit closed form or not. However, in λ-calculus we are not operating with functions, but with λ-terms. So, we face the question: *given an equation of λ-terms with a variable f, can we find a λ-term for f such that the equation holds?*

Another way to approach the problem here is as: *how do we define a recursive λ-term, i.e. a term whose definition depends on itself?*

The answer to these questions is given by the concept of a *fixed point*. It turns out that for any λ-term M there is a λ-term y such that $M\,y =_\beta y$. In other words: for each λ-term there is a λ-term that remains entirely unaffected by applying M. Such a y is called a *fixed point*.

So, how do we find the fixed point of any given λ-term M? We use a $\mathbf{Y}$ combinator: a λ-term that, when applied to a λ-term M yields a representation of the fixed point. Hence, with $y = \mathbf{Y}\,M$ we get $M\,(\mathbf{Y}\,M) =_\beta \mathbf{Y}\,M$.

There are several such $\mathbf{Y}$ combinators. We present here the $\mathbf{Y}$ combinator according to Curry:

$$\mathbf{Y} \triangleq \lambda f.\big(\lambda x.f(x\,x)\big)\big(\lambda x.f(x\,x)\big)$$

This satisfies:

$$\mathbf{Y}\,M \to \big(\lambda x.M(x\,x)\big)\big(\lambda x.M(x\,x)\big) \to M\,\big((\lambda x.M(x\,x))\big(\lambda x.M(x\,x)\big)$$

hence

$$\mathbf{Y}\,M \twoheadrightarrow M\Big(\big(\lambda x.M(x\,x)\big)\big(\lambda x.M(x\,x)\big)\Big) \twoheadleftarrow M(\mathbf{Y}\,M)$$

So for all λ-terms M we have

$$\mathbf{Y}\,M =_\beta M(\mathbf{Y}\,M)$$

Note that this $\mathbf{Y}$ combinator does not really give rise to a 'usable' explicit form of the fixed point. It is not a means to actually 'solve' an equation, but merely tells us that there *is a solution*.

Representing primitive recursion. If $f \in \mathbb{N}^n \to \mathbb{N}$ is represented by a λ-term F and $g \in \mathbb{N}^{n+2} \to \mathbb{N}$ is represented by a λ-term G, we want to show λ-definability of the unique $h \in \mathbb{N}^{n+1} \to \mathbb{N}$ satisfying:

$$h = \mathbf{\Phi}_{f,g}(h)$$

where $\mathbf{\Phi}_{f,g} \in (\mathbb{N}^{n+1} \to \mathbb{N}) \to (\mathbb{N}^{n+1} \to \mathbb{N})$ is given by:

$$\mathbf{\Phi}_{f,g}(h)(\vec{a}, a) \triangleq \begin{cases} f(\vec{a}) & \text{if } a = 0 \\ g\,(\vec{a}, a-1, h(\vec{a}, a-1)) & \text{else} \end{cases}$$

We now know that h can be represented by:

$$\mathbf{Y}\Big(\lambda z\,\vec{x}\,x.\mathbf{If}(\mathbf{Eq_0}x)(F\,\vec{x})\big(G\,\vec{x}\,(\mathbf{Pred}\,x)(z\,\vec{x}\,(\mathbf{Pred}\,x))\big)\Big)$$

Recall that the class PRIM of primitive recursive functions is the smallest collection of (total) functions containing the basic functions and closed under the operations of composition and primitive recursion.

Combining the results about λ-definability so far, we have: *every $f \in \mathrm{PRIM}$ is λ-definable*.

Examples of fixed point equations. The idea of a fixed point in λ-calculus is very powerful and lets us define both data and functions. Let us extend the usual notation of λ-calculus for a moment and concentrate on the concept of solving recursive equations.

Suppose we have an inc operator that takes a list of integers and replaces each number by its successor (i.e. adds one to each number), for instance:

$$\mathrm{inc}([2,3,5,7]) = [3,4,6,8]$$

How do you properly define such an operator? In a functional language you would probably do something like the following (where we assume that :: stands for the *cons*-operator):

```
inc nil    = nil
  | h :: t = succ(h) :: inc t
```

From the perspective of a programmer this (recursive) definition is simple enough and obviously works. From a more mathematical point of view, however, this is an equation and we are rather interested in the question: is there a solution, i.e. an actual function inc that fulfills this equation?

For simplicity's sake we will omit the base case and concentrate on the recursive part, which we can then simply write as:

$$\mathrm{inc}(h :: t) = \mathrm{succ}(h) :: \mathrm{inc}(t),$$

or alternatively (where hd and tl stand for "head" and "tail" of the list, respectively):

$$\mathrm{inc} = \lambda\ell.\ \mathrm{succ}(\mathrm{hd}\,\ell) :: \mathrm{inc}(\mathrm{tl}\,\ell)$$

We now have an equation of the form $f = \mathcal{F}(f)$. In other words the function inc that we seek is a fixed point of the functional $\mathcal{F}$ with:

$$\mathcal{F} = \lambda f.\,\lambda\ell.\ \mathrm{succ}(\mathrm{hd}\,\ell) :: f(\mathrm{tl}\,\ell)$$

If we can express this functional $\mathcal{F}$ in λ-calculus then the fixed point operator $\mathbf{Y}$ gives us a solution to this equation with $f = \mathbf{Y}\mathcal{F}$ as an expression for our function inc.

Once we have the function inc defined and at our disposal, we can then also define a list of natural numbers like so:

$$\tilde{\mathbb{N}} = 0 :: \mathrm{inc}(\tilde{\mathbb{N}})$$

In other words, if you increase each number in $\tilde{\mathbb{N}}$ by one and "cons" the number zero in front of it, you get back the exact same list (this is, of course, owed to the fact that $\tilde{\mathbb{N}}$ is an infinite list).

As before, we have a recursive equation of the form $n = \mathcal{F}(n)$ with:

$$\mathcal{F} = \lambda n.\,0 :: \mathrm{inc}(n)$$

The list of natural numbers is then obviously the fixed point of this functional, i.e. $\tilde{\mathbb{N}} = \mathcal{F}(\tilde{\mathbb{N}})$. Using the $\mathbf{Y}$ combinator we write this as $\tilde{\mathbb{N}} = \mathbf{Y}\mathcal{F}$.

The idea of a recursive equation and its fixed point is a concept that lets you express both (infinite) data structures as well as functions. While recursive definitions are common enough in programming, we need to ensure that λ calculus is actually powerful enough to also express such entities. The $\mathbf{Y}$ combinator thus basically tells us that there is always a λ-term to express your recursive function definition, even though those actual λ-terms tend to be incredibly complex.

10.3.2 Representing Minimization

So for λ-definability of all recursive functions, we just have to consider how to represent minimisation. Recall:

Definition 35:

Given a parial function $f \in \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$, define $\mu^n f \in \mathbb{N} \rightharpoonup \mathbb{N}$ by: $\mu^n f(\vec{x})$ is the least x such that $f(\vec{x}, x) = 0$ and for each $i = 0, \ldots, x-1$, $f(\vec{x}, i)$ is defined and larger than zero. $\mu^n f(\vec{x})$ is undefined if there is no such x.

We can express $\mu^n f$ in terms of a fixed point equation:

$$\mu^n f(\vec{x}) \equiv g(\vec{x}, 0)$$

where g satisfies $g = \boldsymbol{\Psi}_f(g)$ with $\boldsymbol{\Psi}_f \in (\mathbb{N}^{n+1} \rightharpoonup \mathbb{N}) \to (\mathbb{N}^{n+1} \rightharpoonup \mathbb{N})$ defined by

$$\Psi_f(g)(\vec{x}, x) \equiv \begin{cases} x & \text{if } f(\vec{x}, x) = 0 \\ g(\vec{x}, x+1) & \text{else} \end{cases}$$

Supopse $f \in \mathbb{N}^{n+1} \to \mathbb{N}$ (totally defined function) satisfies $\forall \vec{a}\, \exists a. \big(f(\vec{a}, a) = 0\big)$, so that $\mu^n f \in \mathbb{N}^n \to \mathbb{N}$ is totally defined.

Thus for all $\vec{a} \in \mathbb{N}^n$, $\mu^n f(\vec{a}) = g(\vec{a}, 0)$ with $g = \boldsymbol{\Psi}_f(g)$ and $\boldsymbol{\Psi}_f(g)(\vec{a}, a)$ is given by:

$$\Psi_f(g)(\vec{a}, a) \equiv \begin{cases} a & \text{if } f(\vec{a}, a) = 0 \\ g(\vec{a}, a+1) & \text{else} \end{cases}$$

So if f is represented by a λ-term F, then $\mu^n f$ is represented by:

$$\lambda \vec{x}.\mathbf{Y}\Big(\lambda z\,\vec{x}\,x.\mathbf{If}\big(\mathbf{Eq_0}(F\,\vec{x}\,x)\big)\,x\,\big(z\,\vec{x}\,(\mathbf{Succ}\,x)\big)\Big)\,\vec{x}\,\underline{0}$$

10.4 λ-Definability

Recursive implies λ-definable Fact: every partial recursive $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ can be expressed in a standard form as $f = g \circ (\mu^n h)$ for some $g, h \in \mathrm{PRIM}$ (this follows from the proof that computable is equivalent to partial-recursive). Hence every (total) recursive function is λ-definable.

More generally, every partial recursive function is λ-definable, but matching up $\uparrow$ with that there exists no β-normal form makes the representations more complicated than for total functions (see: Hindley & Seldin, Chapter 4).

Computable = λ-definable

Theorem 36:

A partial function is computable if and only if it is λ-definable.

We already know that computable means partial recursive and hence is λ-definable. So it just remains to see that *λ-definable functions are register machine computable*. To show this one can

- Code λ-terms as numbers (ensuring that operations for constructing and deconstructing terms are given by register machine computable functions on codes);
- Write a register machine interpreter for (normal oder) β-reduction.

The details are straightforward, although tedious.

Bibliography

[1] N. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[2] M. Davis, R. Sigal, and E. J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Elsevier, 1994.

[3] J. R. Hindley and J. P. Seldin. *Lambda-calculus and Combinators, an Introduction*. Cambridge University Press, 2008.

[4] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[5] T. A. Sudkamp. *Languages and Machines–An Introduction to the Theory of Computer Science*. Addison-Wesley, 1997.

Index

α-conversion, 60
α-equivalence, 60
α-transformation, 57
β-expansion, 62
β-nf, 64
β-normal form, 64
β-redex, 61
β-reduct, 61
β-reduction, 61
β-transformation, 57
$\equiv$, 44
proj, 44
succ, 44
zero, 44
LaTeX, 66
TeX, 66

abacus, 55
abstraction, 58
Ackermann
    function, 53
addition, 48
algorithm, 37
alpha-transformation, 57
application, 58
axiom, 9

basic functions, 70
beta-transformation, 57
binding, 59
body, 59
boolean, 68
bound, 59
bound variable, 59

calculus
    Lambda, 55
canonical systems, 43
characteristic function, 35
Church-Rosser theorem, 63
Church-Turing Thesis, 43
Church-Turing thesis, 55
closed term, 59
coding, 23
Combinator
    Fixed point, 72
    Y, 72
combinator, 59
compiler, 31, 65
complete
    Turing, 27, 30
completeness
    Turing, 30
composition, 44, 70
computability, 43
computable, 69
    Turing, 42
computable function, 21
computable partial function, 43
computation, 39
    register machine, 19
computer, 55
configuration, 38
    register machine, 18
confluence, 63
confluent, 62, 63
constant, 22, 44
constant function, 70
contraction, 61
conversion
    α, 60
Curry
    Fixed point combinator, 72

deciding halting problem, 33
decision problem, 10
definable
  λ, 69
  lambda, 69
deterministic, 20
division
  integer, 22

Entscheidungsproblem, 10, 37
Eq_0, 68
equivalence
  α, 60
  Kleene, 44
Euclid, 9, 15
expansion, 62
exponentiation, 22

False, 68
fixed point, 72
Fixed point combinator, 72
four colour theorem, 10
free, 59
free variable, 59
Fst, 68
function, 59
  Ackermann, 53
  characteristic, 35
  computable, 21
  constant, 70
  partial, 20, 47
  recursive, 47, 53
  successor, 70
  total, 21
functional programming, 43, 56

Gödel, 15, 23, 43
graph
  register machine, 19

halting, 19
Halting problem, 33
head position, 65
Hilbert, 9, 37

I, 58
identity, 58, 59
If, 68
imperative programming, 56
inc, 73
incompleteness theorem, 15
instruction set arcitecture, 30
integer division, 22
integer remainder, 22
ISA, 30

Kleene, 43
Kleene equivalence, 44
Knuth, 66

Lambda calculus, 55
lambda-definable, 69
Lambek, 43
language
  programming, 30
logarithm, 22

machine, 30
  register, 18
  Turing, 37, 55
  virtual, 27
macro, 66
Markov, 43
minimisation, 50, 51
minimization, 74
Minsky, 43
multiplication, 49

nf, 64
non-termination, 64
normal form, 64
normal-order reduction, 64

optimisation, 65
ordered pair, 68

Pair, 68
pair, 58, 68
parallel postulate, 15
partial function, 20, 47
  computable, 43
partial recursive, 52
partial recursive function, 43
Pascal, 31
Post, 43
postulate
  fifth, 15
  parallel, 15
Pred, 71
predecessor, 49, 71

primitive recursion, 71
primitive recursive, 49
program, 30
programming
    functional, 43, 56
    imperative, 56
programming language, 30
projection, 22, 44, 70

recursion
    primitive, 71
recursion theory, 43
recursive
    primitive, 49
    total, 53
recursive function, 47, 53
recusrive
    partial, 52
redex, 61, 65
reduct, 61
reduction
    β, 61
    normal-order, 64
register machine, 18
    universal, 27
remainder
    integer, 22
replace, 56
representation, 59
    graphical, 19
RM, 18
Rosser
    Church-R. theorem, 63

Snd, 68
substitution, 56, 61
subtraction
    truncated, 22
succ, 70, 73
succesor, 44
successor, 70

term, 59
    closed, 59
    lambda, 59
termination, 64
test
    for zero, 68
theorem
    Church-Rosser, 63
    four colour, 10
    incompleteness, 15
thesis
    Church-Turing, 55
total function, 21
total recursive, 53
transformation
    α, 57
    β, 57
True, 68
truncated subtraction, 22
tuple, 68
Turing, 43, 55
    computable, 42
Turing complete, 27, 30
Turing completeness, 30
Turing machine, 37

variable, 58
    bound, 59
    free, 59
virtual machine, 27

Y, 72

zero, 44, 70