University of The Gambia

Computer Sciences Department

CPS321: Theory of Computing II

J. Boillat & P. Fierz

Keywords: Computability, Complexity, Turing machines, Recursive Functions, Lambda Calculus, Numbers

[File cc.tex, Date 09.09.09]

Abstract

An introduction to the modern theory of computing. Topics include computability and complexity theory.

Chapter 1

Turing Machines

1.1 Problems that Computers Cannot Solve

It is important to know whether a program is correct, namely that it does what we expect. It is easy to see that the following JAVA program

```
public class Main {
  void main(String [] args)  {
    System.out.println("hello, world");
  }
}
```

prints `hello, world` and terminates.

But about following program:

```
public class Main {
  int exp(int i, int n) {
    int r;
    for (int j = 1; j <= n; j++) {
      r *= i;
    }
    return r;
  }

  void main(String [] args)  {
    int n = 123456;
    int t = 254;
    while (true) {
      for int x = 1; x <= t - 2; x++) {
        for (int y = 1; y <= t - x -1; y++) {
          for (z = 1; z <= t - x - y; z++) {
            if (exp(x,n) + exp(y,n) == exp(z,n) {
              System.out.println("hello, world");
            }
          }
        }
```

```
        }
      }
    }
  }
```

Given an input n, it prints hello, world only if the equation

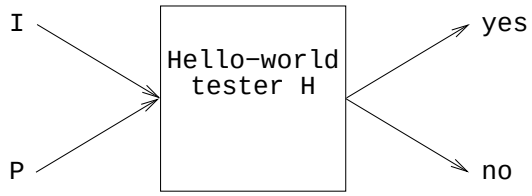
$$x^n + y^n = z^n$$

has a solution where x, y, and z are integers. We know nowadays that it will print hello, world for $n = 2$, and loop forever for $n > 2$.

It took mathematicians 300 years to prove this so-called *Fermat's last theorem*[1]. Can we expect to write a program `H` that solves the general problem of telling whether any given program `P`, on any given input `I`, eventually prints `hello, world` or not?

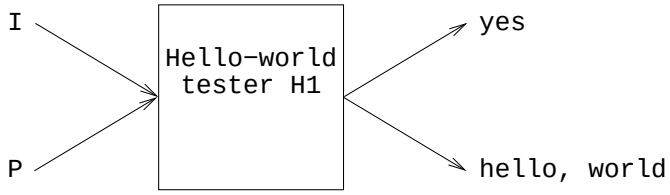
1.1.1 The hypothetical `hello, world` tester

Proof by contradiction that `H` is impossible to write. Suppose that `H` exists:

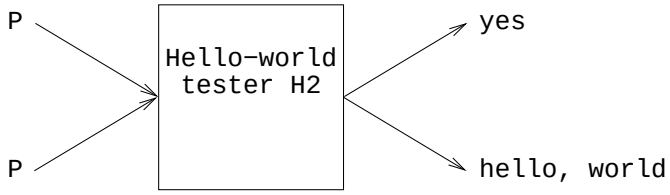


`H` takes as input a program `P` and an input `I`, and tells whether `P` with input `I` prints `hello, world`. In particular, the only output `H` makes is either to print `yes` or to print `no`.

We modify the response no of `H` to `hello, world`, getting a program `H1`:

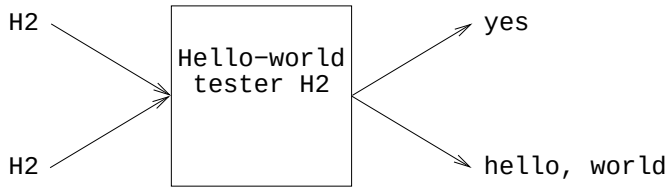


We modify `H1` to take `P` and `I` as a single input, getting a program `H2`:



We provide `H2` as input to `H2`:

[1]The conjecture has been proved by Andrew Wiles in year 1994



If `H2` prints `yes`, then it should have printed `hello, world`.

If `H2` prints `hello, world`, then it should have printed `yes`.

So `H2` and hence `H` cannot exist. □

Hence we have an *undecidable problem.* It is similar to the language L_d we will see later.

1.1.2 Undecidable Problems

Definition 1.1 [Undecidable] *A problem is* undecidable *if no program can solve it.*

- *Problem* means deciding on the membership of a string in a language.
- Languages over an alphabet are not enumerable.
- Programs (finite strings over an alphabet) are enumerable: order them by length, and then lexicographically.
- Hence there are infinitely more languages than programs.
- Hence there must be undecidable problems (Gödel[2], 1931).

Gödel Incompleteness Theorem

In 1931, Gödel published his famous incompleteness theorems in *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme* [Gö31]. In that article, he proved for any computable axiomatic system that is powerful enough to describe the arithmetic of the natural numbers, that:

1. If the system is consistent, it cannot be complete.
2. The consistency of the axioms cannot be proved within the system.

In hindsight, the basic idea at the heart of the incompleteness theorem is rather simple. Gödel essentially constructed a formula that claims that it is unprovable in a given formal system. If it were provable, it would be false, which contradicts the fact that in a consistent system, provable statements are always true. Thus there will always be at least one true but unprovable statement. That is, for any computably enumerable set of axioms for arithmetic (that is, a set that can in principle be printed out by an idealized computer with unlimited resources), there is a formula that obtains in arithmetic, but which is not provable in that system. To make this precise, however, Gödel needed to solve several technical issues, such as encoding statements, proofs, and the very concept of provability into the natural numbers. He did this using a process known as Gödel numbering.

[2]Kurt Gödel (1906 - 1978) was an Austrian-American logician, mathematician and philosopher. One of the most significant logicians of all time, Gödel made an immense impact upon scientific and philosophical thinking in the 20th century, a time when many, such as Bertrand Russell, A. N. Whitehead and David Hilbert, were pioneering the use of logic and set theory to understand the foundations of mathematics.

1.1.3 Problem Reduction

A *reduction* is a transformation of one problem into another problem. Depending on the transformation used this can be used to define complexity classes on a set of problems.

Intuitively, problem A is reducible to problem B if solutions to B exist and give solutions to A whenever A has solutions. Thus, solving A cannot be harder than solving B. We write $A \leq B$, usually with a subscript on the $\leq$ to indicate the type of reduction being used.

Often we find ourselves trying to solve a problem that is similar to a problem we've already solved. In these cases, often a quick way of solving the new problem is to transform each instance of the new problem into instances of the old problem, solve these using our existing solution, and then use these to obtain our final solution. This is perhaps the most obvious use of reductions.

Another, more subtle use is this: suppose we have a problem that we've proved is hard to solve, and we have a similar new problem. We might suspect that it, too, is hard to solve. We argue by contradiction: suppose the new problem is easy to solve. Then, if we can show that every instance of the old problem can be solved easily by transforming it into instances of the new problem and solving those, we have a contradiction. This establishes that the new problem is also hard.

Example 1.1 [Multiplication/Squaring]

A very simple example of a reduction is from multiplication to squaring. Suppose all we know how to do is to add, subtract, take squares, and divide by two. We can use this knowledge, combined with the following formula, to obtain the product of any two numbers:

$$a \times b = \frac{\left((a+b)^2 - a^2 - b^2\right)}{2}$$

We also have a reduction in the other direction; obviously, if we can multiply two numbers, we can square a number. This seems to imply that these two problems are equally hard.

Example 1.2 [Function call] Does a program Q, given input q, ever call function `foo()`?

This problem is undecidable.

We construct a program R and an input z such that R with input z, calls `foo()` if and only if Q with input y prints `hello, world`:

1. If Q has a function called `foo()`, rename it and all calls to that function. Clearly the new program Q_1 does exactly what Q does.
2. Add to Q_1 a function `foo()`. This function does nothing, and is not called. The resulting program is Q_2.
3. Modify Q_2 to remember the first 12 characters that it prints, storing them in a global array A. The resulting program is Q_3.
4. Modify Q_3 so that whenever it executes any output statement, it then checks in the array A to see if it has written 12 characters or more, and so, whether `hello, world` are the first 12 characters. In that case call the new function `foo()`. The resulting program is R, and input z is the same as y.

Now suppose that Q with input y prints `hello, world` as its first output. Then R as constructed will call `foo()`. However, if Q with input y does not print `hello, world` as its first output, then R will never call `foo()`. If we can decide whether R with input z calls `foo()`, then we also know whether Q with input y (remember $y = z$) prints `hello, world`. Since we know that no algorithm to decide the hello-world problem exists, so the assumption that there is a calls-foo tester is wrong. Thus the calls-foo problem is undecidable.

Exercise 1.1 [Halt-Tester]

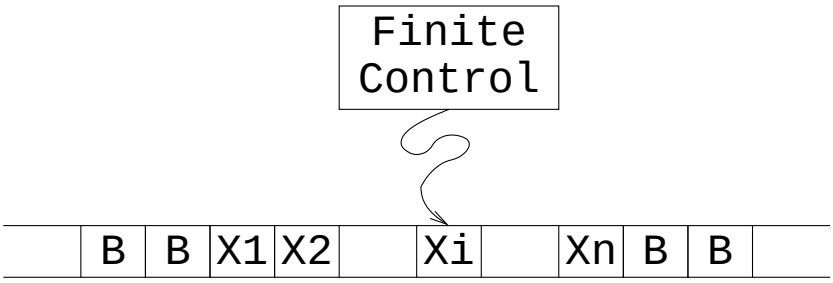
Give a reduction from the hello-world problem to the following problem:

Given a program and an input, does the program eventually halt, i.e., does the program not loop forever on the input?

Use informal style for describing plausible program transformations. Don't forger to consider the exceptions that may be thrown by the original program.

1.2 Turing Machines

A Turing machine is a theoretical device that manipulates symbols contained on a strip of tape. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside of a computer. The Turing machine was described in 1936 by Alan Turing[3]. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine (thus they have never actually been constructed).



A move of a Turing machine (TM) is a function of the state of the finite control and the tape symbol just scanned.

In one move, the Turing machine will:

1. Change state.
2. Write a tape symbol in the cell scanned.
3. Move the tape head left or right.

[3]Alan Turing, 23 June 1912 - 7 June 1954), was an English mathematician, logician, cryptanalyst, and computer scientist. He was influential in the development of computer science and provided an influential formalisation of the concept of the algorithm and computation with the Turing machine. During the Second World War, Turing worked for the Government Code and Cypher School at Bletchley Park, Britain's codebreaking centre. For a time he was head of Hut 8, the section responsible for German naval cryptanalysis. He devised a number of techniques for breaking German ciphers, including the method of the bombe, an electromechanical machine that could find settings for the Enigma machine.

1.2.1 Alan Turing and the *Entscheidungsproblem*

In his momentous paper *On Computable Numbers, with an Application to the Entscheidungsproblem* [Tur37], Turing reformulated Kurt Gödel's 1931 results on the limits of proof and computation, replacing Gödel's universal arithmetic-based formal language with what are now called Turing machines, formal and simple devices. He proved that some such machine would be capable of performing any conceivable mathematical computation if it were representable as an algorithm.

Turing machines are to this day the central object of study in theory of computation. He went on to prove that there was no solution to the Entscheidungsproblem by first showing that the halting problem for Turing machines is undecidable: it is not possible to decide, in general, algorithmically whether a given Turing machine will ever halt. While his proof was published subsequent to Alonzo Church's equivalent proof in respect to his lambda calculus, Turing's work is considerably more accessible and intuitive. It was also novel in its notion of a *Universal Machine*, the idea that such a machine could perform the tasks of any other machine. *Universal* in this context means what is now called programmable. The paper also introduces the notion of definable numbers.

1.3 Turing Machines: Formal Definition

Definition 1.2 [Turing Machine] *Formally, a* Turing machine *is a 7-tuple*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where:

1. Q *is the finite set of* states *of the finite control.*
2. Σ *is the finite set of* input symbols.
3. Γ *is the finite set of* tape symbols*;* $\Sigma \subset \Gamma$.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the* transition function*, which is a partial function.*
   The arguments of $\delta(q, X)$ *are a state* q *and a tape symbol* X*. The value of* $\delta(q, X)$*, if it is defined is a triple* (p, Y, D)*, where:*
   (a) p *is the next state, in* Q.
   (b) Y *is the symbol, in* Γ*, written in the cell being scanned, replacing whatever symbol was there.*
   (c) D *is a* direction*, either* L *or* R*, standing for* left *or* right*, respectively, and telling us the direction in which the head moves.*
5. $q_0 \in Q$ *is the* start state.
6. $B \in \Gamma$ *is the* blank symbol. $B \notin \Sigma$.
7. $F \subset Q$ *is the set of* final *or* accepting states.

Notation 1.1 [Blank] We denote the blank symbol B by $\Box$.

Definition 1.3 [Output] *The output of a TM consists in the longest string* $w \in (\Gamma - \{\Box\})^*$ *immediately following on the right of the actual tape position.*

1.3.1 Instantaneous Description and Moves

A Turing machine changes its configuration upon each move.

We use *instantaneous descriptions* (*ID*s) for describing such configurations.

Definition 1.4 [Instantaneous Description]

An instantaneous description *of a TM is a string of the form*

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n$$

where

1. *q is the state of the Turing machine.*
2. *The tape head is scanning the ith symbol from the left.*
3. *$X_1X_2\cdots X_n$ is the portion of the tape between the leftmost and rightmost nonblanks.*

We use $\underset{M}{\vdash}$ to designate a move of a Turing machine M from one ID to another.

Definition 1.5 [Move] *If $\delta(q, X_i) = (p, Y, L)$, then:*

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$

If $\delta(q, X_i) = (p, Y, R)$, then:

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-1}YpX_{i-+}X_{i+2}\cdots X_n$$

The reflexive-transitive closure of $\underset{M}{\vdash}$ is denoted $\overset{*}{\underset{M}{\vdash}}$.

1.3.2 Language of a TM

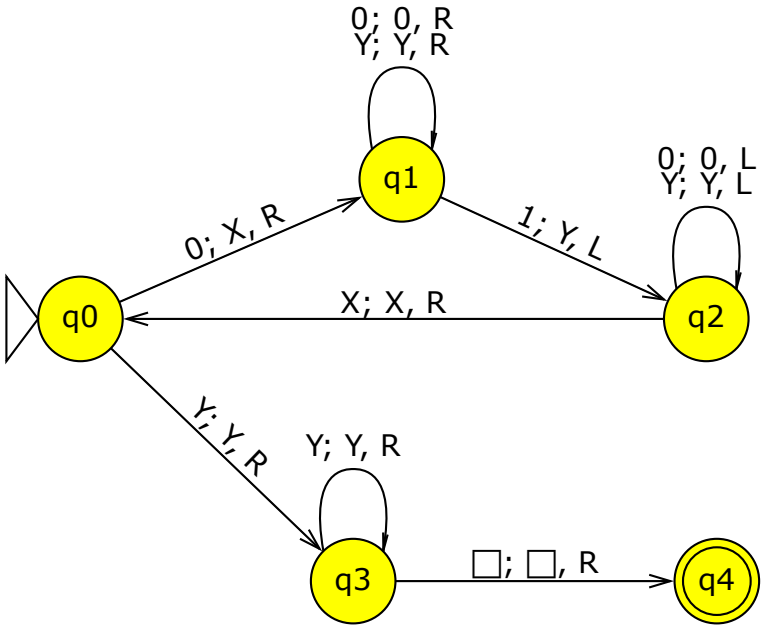
Definition 1.6 [Language] *A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ accepts the* language

$$L(M) = \{w \in \Sigma^* : q_0w \overset{*}{\underset{M}{\vdash}} \alpha p\beta, p \in F, \alpha, \beta \in \Gamma^*$$

Example 1.3 [TM for L_{01}] $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, \square\}, \delta, q_0, \square, \{q_4\})$ where δ is given by the following table:

	0	1	X	Y	$\square$
$\rightarrow q_0$	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	$(q_4, \square, R)$
$*q_4$					

We can also represent M by the following transition diagram:



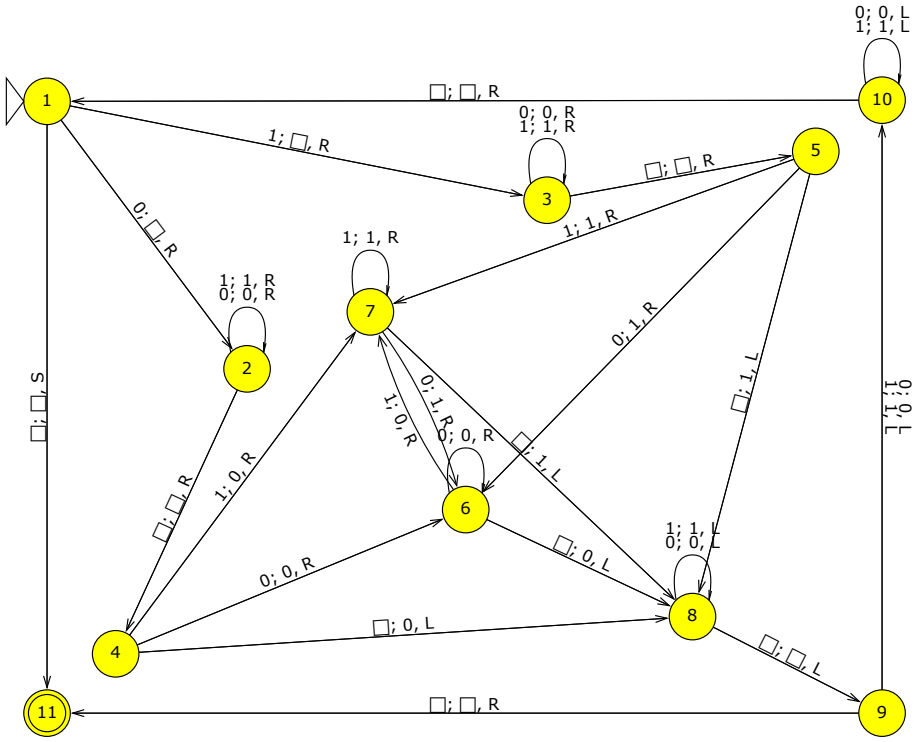
As M performs its computation, the portion of the tape, where M's tape head has visited, will always be a sequence of symbols described by the regular expression $X^*0^*Y^*1^*$.

State q_0 is the initial state, and M also enters state q_0 every time it returns to the leftmost remaining 0. If M is in state q_0 and scanning a 0, it will go to state q_1 and change the 0 to an X. Once in state q_1, M keeps moving right over all 0's and Y's remaining in that state. If M sees an X or a $\square$, it dies. If M sees a 1 when in state q_1 it changes that 1 to a Y, enters state q_2, and starts moving left.

In state q_2, M moves left over 0's and Y's, remaining in state q_2. When M reaches the rightmost X, which marks the right end of the block of 0's that have already been changed to X, M returns to state q_0 and moves right. There are two cases:

1. If M sees a 0, the it repeats the matching cycle we have just described.
2. If M sees a Y, then it has changed all 0's to X's. If all 1's have been changed to Y's, then the input was of the form 0^n1^n, and M should accept. Thus M enters state q_3, and starts moving right, over Y's. If the first symbol other than Y that M sees is a blank, then indeed there were an equal number of 0's and 1's, so M enters state q_4 and accepts. On the other hand, if M encounters another 1, then there are too many 1's ant M dies without accepting. If it encounters a0, then the input was of the wrong form, and M also dies.

Example 1.4 [Reverse] Following TM reserves and input string of 0's and 1's.

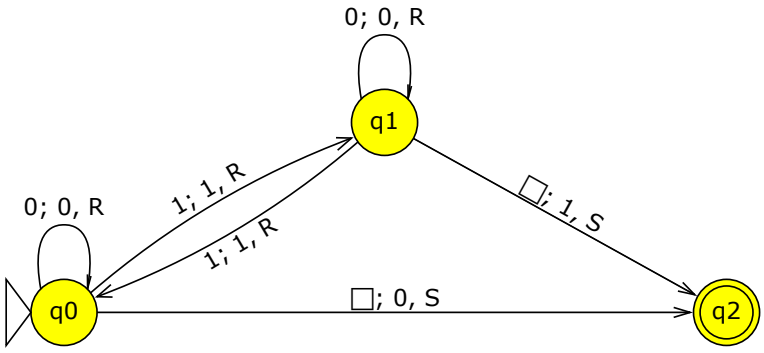


In state 1, M reads the leftmost symbol and replaces it by a blank. If the symbol was a 0, M enters state 2 and moves to the right until it reaches the first blank symbol, otherwise M enters state 3 and moves to the right until it reaches the first blank symbol. States 2 and 3 are used to remember the value of the leftmost symbol.

The leftmost symbol is written in state 4 or 5 after the first blank on the right of the input. This may overwrite an already copied leftmost symbol. If it is the case, that symbol will be copied to the right (states 6 and 7). This process continues until the rightmost blank it reached.

The TM now returns to the next leftmost non blank symbol (states 8, 9 and 10). If there is no more such symbol, the TM enters accepting state 11 and halts.

Example 1.5 [Parity] The input of following TM is a string of 0's and 1's. If the input string has an even number of 1's, the TM will write a 0 at the end of the input and halt, otherwise it will write a 1 at the end of the input and halt.



The TM always moves to the right. If the TM is in state q_0, then it has read an even number of 1' so far, otherwise the machine will be in state q_1. At the end of the input, the Tm will

replace the first blank by a 0 if it was in state q_0 and enter q_2, otherwise (the TM was in state q_1) the Tm will replace the first blank by a 1 and enter q_2. The TM halts in accepting state q_2.

1.3.3 Acceptance by Halting

A Turing machine *halts* if it enters a state q, scanning a tape symbol X, and there is no move in this situation, i.e., $\delta(q, X)$ is undefined.

We can always assume that a Turing machine halts if it accepts, as we can make $\delta(q, X)$ undefined whenever q is an accepting state. We will always assume that a TM halts when it is in an accepting state.

Unfortunately, it is not always possible to require that a Turing machine halts even if it does not accept.

Definition 1.7 [Recursive Language] *A* Recursive language *is a language corresponding to a TM that eventually, regardless of whether or not they accept.*

Remark 1.1 [Recursive Language] If a TM M always halts, regardless of whether or not it accepts, we can assume that M has two accepting states q_{accept} and q_{reject}.

1.3.4 Turing Computable Function

Definition 1.8 [Turing Computable Function] *A function $f : \Sigma^* \to \Sigma^*$ is a* computable function *if some TM M, on every input w, halts with output $f(w)$.*

Exercise 1.2 [TM for Palindromes] *Construct a Turing Machine for L_{pal}, the language of palindromes.*

Exercise 1.3 [TM for Addition] *Construct a Turing Machine for the addition of binary natural numbers. The initial tape contents is a sum of binary numbers, e.g.* `101+11`

Hint: You may replace `0` *by* `a` *and* `1` *by* `b` *to remember the position of the last bit addition.*

Chapter 2

Recursive Functions

One central question in computer science is the basic question:

What functions are computable by a computer?

Before we try to formalise the concept of a computable function, let us be precise about what we mean by a function. We will be considering functions from natural numbers ($\mathbb{N} = \{0, 1, 2, \dots\}$) to natural numbers. This might seem restrictive, but in fact it is not since we can code almost any type of object as a natural number.

Example 2.6 [Words as Numbers] We can associate a number to each character of the English alphabet, i.e. $a = 01, b = 02, \dots z = 26$. Since a word is a sequence of characters, it is easy to associate a natural number to each English word, i.e. the number associated with the word *hello* is 0804121215.

Example 2.7 [Sets as Numbers] Let $A\{a, b, c, x, z\}$ be a subset of $E = \{a, b, c, \dots y, t\}$, then A can be represented by the 26 digits binary number 11100000000000000000000101 (characteristic function).

Example 2.8 [Graphs as Numbers] Any undirected graph can be represented as a number. Let $G = \{V, E)$ given by $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, b\}, \{b, c\}\}$. We use the following characteristic function:

$\{a,a\}$	$\{a,b\}$	$\{a,c\}$	$\{b,b\}$	$\{b,c\}$	$\{c,c\}$
0	1	0	1	1	0

Adding a leading 1 we get the number 1010110.

2.1 Primitive Recursive Functions

The primitive recursive functions are defined using primitive recursion and composition as central operations and are a strict subset of the recursive functions.

In computability theory, primitive recursive functions are a class of functions which form an important building block on the way to a full formalisation of computability. These functions are also important in proof theory.

Most of the functions normally studied in number theory are primitive recursive. For example: addition, division, factorial, exponential and the nth prime are all primitive recursive. In fact,

it is difficult to devise a function that is not primitive recursive, although some are known. The set of primitive recursive functions is known as PR in complexity theory.

Definition 2.1 [Primitive Recursive Function]

The primitive recursive functions are among the number-theoretic functions which are functions from $\mathbb{N}^n$ to $\mathbb{N}$. These functions take n arguments for some natural number n and are called n-ari.

The basic primitive recursive functions are given by these axioms:

1. **Constant:** *The n-ari constant functions C_i^n which returns the value i is primitive recursive. Note that the value $n = 0$ is allowed.*

2. **Successor:** *The 1-ari successor function S, which returns the successor of its argument, is primitive recursive. That is, $S(k) = k + 1$ for $k \in \mathbb{N}$.*

3. **Projection:** *For every $n \geq 1$ and each i with $1 \leq i \leq n$, the n-ari projection function P_i^n, which returns its i-th argument, is primitive recursive.*

More complex primitive recursive functions can be obtained by applying the operators given by these axioms:

4. **Composition:** *Given g, a m-ari primitive recursive function, and m n-ari primitive recursive functions $h_1, \ldots, h_m$, the composition of $SUB(g, h_1, \ldots, h_m)$, i.e. the n-ari function $SUB(g, h_1, \ldots, h_m)(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$, is primitive recursive.*

5. **Primitive Recursion:** *Given g, a n-ari primitive recursive function, and h, a $(n+2)$-ari primitive recursive function, the $(n+1)$-ari function defined as the primitive recursion $PR(g, h)$ of g and h, i.e. $PR(g, h)(\vec{x}, 0) = g(\vec{x})$ and $PR(g, h)(\vec{x}, S(y)) = h(\vec{x}, y, PR(g, h)(\vec{x}, y))$, is primitive recursive.*

We use here the notation $\vec{x} = (x_1, \ldots, x_n)$.

The primitive recursive *functions are the basic functions and those which can be obtained from the basic functions by applying the operators a finite number of times.*

Notation 2.1 [Primitive Recursive Function] We use the notation $F(PRIM)$ for the set of the primitive recursive functions.

Remark 2.1 [Projection Functions] The projection functions can be used to avoid the apparent rigidity in terms of the arity of the functions above; by using compositions with various projection functions, it is possible to pass a subset of the arguments of one function to another function. For example, if g and h are 2-ari primitive recursive functions then

$$f(a, b, c) = g(h(c, a), h(a, b))$$

is also primitive recursive. One formal definition using projections functions is

$$f(a, b, c) = g(h(P_3^3(a, b, c), P_1^3(a, b, c)), h(P_1^3(a, b, c), P_2^3(a, b, c)))$$

2.1.1 Some Primitive Recursive Functions

Arithmetic Primitive Recursive Functions

Definition 2.2 [Addition]

Intuitively, addition can be recursively defined with the rules:

$$add(x, 0) = x,$$
$$add(x, n + 1) = add(x, n) + 1.$$

In order to fit this into a strict primitive recursive definition, define:

$$add(x, 0) = P_1^1(x),$$
$$add(x, S(n)) = S(P_3^3(x, n, add(x, n))).$$

or

$$add = PR(P_1^1, S \circ P_3^3)$$

Here P_3^3 is the projection function that takes 3 arguments and returns the third one.

P_1^1 is simply the identity function; its inclusion is required by the definition of the primitive recursion operator above; it plays the role of f. The composition of S and P_3^3, which is primitive recursive, plays the role of g.

Definition 2.3 [Subtraction] *Because primitive recursive functions use natural numbers rather than integers, and the natural numbers are not closed under subtraction, a limited subtraction function is studied in this context. This limited subtraction function $sub(a, b)$ returns $b - a$ if this is nonnegative and returns 0 otherwise.*

The predecessor function acts as the opposite of the successor function S and is recursively defined by the rules:

$$pred(0) = 0,$$
$$pred(n + 1) = n.$$

The limited subtraction function is definable from the predecessor function in a manner analogous to the way addition is defined from successor:

$sub(x, 0) = P_1^1(x)$,
$sub(x, S(n)) = pred(P_3^3(x, n, sub(x, n)))$.

Notation 2.2 [Subtraction] We write also $n \perp m$ for $sub(n, m)$

Definition 2.4 [Multiplication]

Multiplication can be defined as:

$$mul(x, 0) = C_0^1(x),$$
$$mul(x, S(n)) = add(x, mul(x, n))$$

Example 2.9 [Finite Sum]
The finite sum $f(x,y) = \sum_{i=0}^{y-1} g(x,i)$

$$f(x,0) = C_0^1(x),$$
$$f(x,S(n)) = add(f(x,n), g(x,n))$$

Example 2.10 [Finite Product]
The finite product $f(x,y) = \prod_{i=0}^{y-1} g(x,i)$

$$f(x,0) = C_1^1(x),$$
$$f(x,S(n)) = mul(f(x,n), g(x,n))$$

Primitive Recursive Predicates

It is convenient to identify *predicates* with functions that take the values 0 and 1, letting the value of the function be 1 exactly when the predicate is *true*. With this convention we define a predicate to be primitive recursive exactly when the corresponding function is primitive recursive.

Definition 2.5 [Signum] *We can define a miniature version sg of the signum function by*

$$sg(0) = 0,$$
$$sg(n+1) = 1.$$

Using sg we can define the equality.

Definition 2.6 [Equality]

$$eq(m,n) = sub(1, add(sg(sub(n,m)), sg(sub(m,n))))$$

Remark 2.2 [If Then Else] Let g and h be primitive recursive functions and let p be a primitive recursive predicate. Then the function $f(x)$ defined by $g(x)$ if $p(x)$ and $h(x)$ otherwise will be primitive recursive since it can be written as

$$add(mul(g(x), p(x)), mul(h(x), sub(1, p(x))))$$

corresponding to $p \cdot g + (1 \perp p) \cdot h$.

Definition 2.7 [Logical Predicates] *Let p and q be primitive recursive predicates. We define*

1. *not:* $\neg(p) \underset{def}{=} 1 \perp p$
2. *or:* $p \vee q \underset{def}{=} sg(p+q)$
3. *and:* $p \wedge q \underset{def}{=} p \cdot q$

Theorem 2.1 [Existence Quantifier] *Let f be a primitive recursive function and p be a primitive recursive predicate, then the predicate q given by*

$$q(n) \underset{def}{=} \exists m < f(n) : p(m)$$

is a primitive recursive predicate.

Proof: We define a primitive recursive function r as follows:

$r(0) = 0$
$r(S(n)) = S(n) \cdot a(n) + r(n) \cdot (1 \perp a(n))$
where $a(n) = (1 \perp p(0)) \cdot p(S(n)) \cdot (1 \perp r(n))$

Therefore, $r(n+1)$ is either $n+1$ (if $a(n) = 1$) or $= r(n)$ (if $a(n) = 0$). Obviously, the first case will occur if and only if all factors of a are 1, i.e. if we have

$$\neg p(0) \wedge p(n+1) \wedge (r(n) = 0)$$

This implies that the function $r(n)$ remains 0 up to the smallest value of n for which $p(n)$ holds, and has that value from then on (if $p(0)$ already holds then $r(n)$ is correspondingly constant and $= 0$). Therefore, we have

$$q(n) = p(r(f(n))$$

(see also [Gö31]) □

Remark 2.3 [Arithmetic and Logic] The function a, which makes use of the fact that a product is 0 if one of its factors is 0, can be described by the following pseudo-code:

```
if (p(0) = 1) {
    return 0
} else if (p(n + 1) = 0) {
    return 0
} else if (r(n) = 0) {
    return 0
} else {
    return 1
}
```

It is a nice example for how arithmetic can be used to emulate logic (see also [Gö31]).

Theorem 2.2 [Argmin] *Let f be a primitive recursive function and p be a primitive recursive predicate, then the function g given by*

$$g(n) \underset{def}{=} argmin\, m \leq f(n) : p(m)$$

where $argmin\, m \leq f(n) : p(m)$ stands for the smallest m for which $(m \leq f(n)) \wedge p(m)$ holds, and for 0 if there is no such number.

Then g is a primitive recursive function.

Proof: $g(n) = r(f(n)$. See proof of theorem 2.1 $\square$

The construct above corresponds to a loop that tries every value from 1 to $f(n)$ to determine the result.

Other Common Primitive Recursive Functions

It is not difficult to prove that most functions are in $F(PRIM)$.

The functions 16-20 are of particular interest with respect to converting primitive recursive predicates to, and extracting them from, their arithmetical form expressed as Gödel numbers [Gö31].

1. Addition: $n + m \underset{def}{=} add(n, m)$
2. Multiplication: $n \cdot m \underset{def}{=} mul(n, m)$
3. Exponentiation: $n^m \underset{def}{=}$ if $m = 0$ then 1 else $mul(n, pred(m))$
4. Factorial: $n! \underset{def}{=}$ if $n = 0$ then 1 else $mul(n, pred(n)!)$
5. Predecessor: $pred(n)$
6. Subtraction: $sub(n, m)$
7. Minimum: $min(n_1, \ldots n_k)$
8. Maximum: $max(n_1, \ldots n_k)$
9. Negation: $\neg p(n) \underset{def}{=} 1 \perp p(n)$
10. Signum: $sg(n) \underset{def}{=}$ if $n = 0$ then 0 else 1
11. Remainder: $rem(n, m) \underset{def}{=}$ the leftover of the division of n by m
12. Divides: $n|m \underset{def}{=}$ if $rem(n, m) = 0$ then 1 else 0"
13. Equality $eq(n, m) \underset{def}{=} sub(1, add(sg(sub(n, m)), sg(sub(m, n))))$
14. Less than: $n < m \underset{def}{=} sg(sub(S(m, n))$
15. Is prime: $prime(n) \underset{def}{=} n > 1$ and $\neg(\exists m)1 < m < n : m|n$
16. p_i the $i + 1$-st prime number
17. Exponent: $(n)_i \underset{def}{=}$ exponent n_i of $p_i \underset{def}{=} argmin\, m < n : p_i^m | n \wedge \neg(p_i^{S(m)} | n)$
18. Length: $lh(n) \underset{def}{=}$ the length or number of non-vanishing exponents in n
19. Prime factors: $n \times m \underset{def}{=}$ given the expression of n and m as prime factors then $n \times m$ is the product's expression as prime factors

20. Logarithm: $lo(n,m) \underset{def}{=}$ logarithm of n to the base m

Exercise 2.1 [Multiplication] *Definition 2.4 is informal. The right side of* $mul(x, S(n)) = add(x, mul(n, x))$ *should be a function with the three arguments* $mul(x, n)$, n *and* x. *Give a formal definition of the multiplication. You will need two projections.*

Exercise 2.2 [Logical Predicates] *Let* p *and* q *be recursive primitive predicates. Show that* $p \rightarrow q$, $p \leftrightarrow q$ *and* $p \oplus q$ *are also primitive recursive predicates.*

Exercise 2.3 [If] *Let* a_i *and* n_i $(i = 1, \ldots, k)$ *be natural numbers such that* $a_i \neq a_j$ *if* $i \neq j$. *Prove that there is* $f \in F(PRIM)$, *such that:*

```
int f(n) {
    if (n = a_1) {
        return n_1
    } else if (n = a_2) {
        return n_2
        ...
    } else if (n = a_k) {
        return n_k
    } else {
        return 0
    }
}
```

2.2 Recursive Functions

2.2.1 Partial Recursive Functions

We add a new building rule to those of definition 2.1.

Definition 2.8 [Partial Recursive Functions] *The* partial recursive functions *contains the basic functions (1-3) defined in 2.1 and are closed under the operations 4 and 5. There is an extra way of forming new functions:*

6. **Unbounded Search:** *Assume* $g(\vec{x}, y)$ *is a partial recursive function and let* $f(\vec{x}) = \mu_y(g)(\vec{x})$ *be the least* m *such that* $g(\vec{x}, m) = 0$ *and such that* $g(\vec{x}, i) \neq 0$ *is defined for all* $i < m$. *If no such* m *exists then* $f(\vec{x})$ *is undefined. Then* f *is partial recursive.*

$$\begin{aligned} \mu_y(g)(\vec{x}) &= \mu y : g(\vec{x}, y) = 0 \\ &= min\{y : g(\vec{x}, y) = 0\} \end{aligned}$$

Notation 2.3 [Partial Recursive Function] We use the notation $F(PAR)$ for the set of the partial recursive functions.

The μ-operator is also called *minimisation-operator.*

2.2.2 Recursive functions

Definition 2.9 [Recursive Functions] *A function is* recursive *(or* total recursive*) if it is a partial recursive function which is total, i.e. which is defined for all inputs.*

Notation 2.4 [Recursive Function] We use the notation $F(REC)$ for the set of the recursive functions.

The Ackermann Function

Definition 2.10 [Ackermann Function] *The* Ackermann function *is defined as follows:*

$$\begin{aligned} a(0,m) &= 1 \,\forall m \geq 0 \\ a(1,0) &= 2 \\ a(n,0) &= n+2 \,\forall n \geq 2 \\ a(n+1,m+1) &= a(a(n,m+1),m) \,\forall n,m \geq 0 \end{aligned}$$

The Ackermann[1] function was one of the first functions that is recursive but not primitive recursive.

Theorem 2.3 [Ackermann Function] *The Ackermann function is not primitive recursive. Furthermore, let a_m be defined as $a_m(n) = a(n,m)$ and let f be a primitive recursive function, then there is a number $k \in \mathbb{N}$ such that*

$$f(x_1,\ldots,x_n) < a_k(x_1+\cdots+x_n) \,\forall (x_1,\ldots,x_n) \in \mathbb{N}^n$$

□

Theorem 2.4 [$F(PRIM)$ and $F(REC)$] *The set of recursive primitive functions is a proper subset of the set of recursive functions.*

$$F(PRIM) \subset F(REC)$$

> **Proof:** $F(PRIM)$ is defined using rules 1-5 of definition 2.1, $F(REC)$ is defined using rules 1-5 of definition 2.1 and rule 6 of definition 2.8. So any primitive recursive function is recursive. We can use theorem 2.3 to show that $F(PRIM)$ is a proper subset of $F(REC)$. □

Exercise 2.4 [Ackermann Function]

1. *Compute $a(2,1)$.*
2. *Prove that $a(n,1) = 2n \;\forall n \geq 1$.*
3. *Prove that $a(n,2) = 2^n \,\forall n \geq 0$.*
4. *Evaluate $a(4,3)$.*
5. *What is the value of $a(n,3)$?*

Exercise 2.5 [Ackermann Function] *Prove that the function a_n defined in theorem 2.3 is primitive recursive for all $n \in \mathbb{N}$.*

[1] Wilhelm Friedrich Ackermann (1896 - 1962) was a German mathematician best known for the Ackermann function, an important example in the theory of computation.

2.3 Gödel Numbers

In mathematical logic, a Gödel numbering is a function that assigns to each symbol and well-formed formula of some formal language a unique natural number, called its Gödel number. The concept was first used by Kurt Gödel for the proof of his incompleteness theorem.

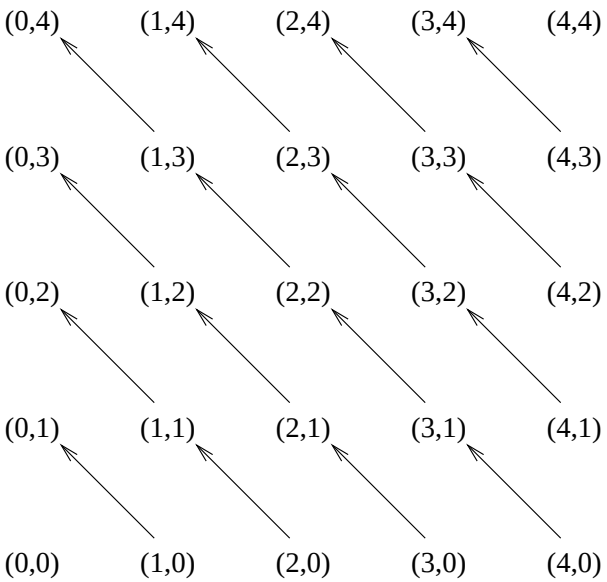
A Gödel numbering can be interpreted as an encoding in which a number is assigned to each symbol of a mathematical notation, after which a sequence of natural numbers can then represent a sequence of strings. These sequences of natural numbers can again be represented by single natural numbers, facilitating their manipulation in formal theories of arithmetic.

2.3.1 Gödel Numbers for Sequences

We want a totally recursive function f that satisfies: For all n and for any n-length sequence of natural numbers $(a_0, \ldots a_{n-1})$, there exists an appropriate natural number a, called the Gödel number of the sequence such that for all i in the range of $0 \ldots n-1$

$$f(a, i) = a_i$$

We start with a simple numbering of $\mathbb{N}^2$. We can use the following construction to show that $\mathbb{N}^2$ is a countable set. We write the elements of $\mathbb{N}^2$ in the right upper quadrant and follow the arrows from down right $(i, 0)$ to left up $(0, i)$ for $i = 0, \ldots, n$:



Theorem 2.5 [Numbering of $\mathbb{N}^2$]

Consider $\tau : \mathbb{N}^2 \to \mathbb{N}$ given by

$$\tau(i, j) = \frac{(i+j)(i+j+1)}{2} + j$$

then

1. *τ is bijective.*

2. *τ is primitive recursive.*

3. *The associated projection functions π_1 and π_2, such that $\pi_i(\tau(x_1, x_2)) = x_i$ $(i = 1, 2)$ are primitive recursive.*

Proof:
1) See remark 2.4.
2) By definition, τ is a composition of primitive recursive functions, it follows that τ is primitive recursive.
3) It is easy to see that

$$\pi_1(x) = \mu x_1 \leq x(\exists x_2 \leq x : \tau(x_1, x_2) = x)$$

□

Remark 2.4 [Cantor Function] The function τ is called *Cantor Function*[2] the inverse of τ as follows:

Suppose we are given z with

$$z = \tau(x, y) = \frac{(x + y)(x + y + 1)}{2} + y$$

and we want to find x and y. It is helpful to define some intermediate values in the calculation:

$$\begin{array}{rcl} w & = & x + y \\ t & = & \frac{w(w+1)}{2} \\ z & = & t + y \end{array}$$

where t is the triangle number of w. If we solve the quadratic equation

$$w^2 + w - 2t = 0$$

for w as a function of t, we get

$$w = \frac{\sqrt{8t + 1} - 1}{2}$$

which is a strictly increasing and continuous function when t is non-negative real. Since

$$t \leq z = t + y < t + (w + 1) = \frac{(w + 1)^2 + (w + 1)}{2}$$

we get that

$$w \leq \frac{\sqrt{8z + 1} - 1}{2} < w + 1$$

and thus

[2]Georg Cantor (1845-1918) was a German mathematician, born in Russia. He is best known as the creator of set theory, which has become a fundamental theory in mathematics. Cantor established the importance of one-to-one correspondence between sets, defined infinite and well-ordered sets, and proved that the real numbers are not countable.

$$w = \lfloor \frac{\sqrt{8z+1}-1}{2} \rfloor$$

So to calculate x and y from z and w, we do:

$$\begin{array}{rcl} t & = & \frac{w^2+w}{2} \\ y & = & z - t \\ x & = & w - y \end{array}$$

We can iterate the construction of theorem 2.5 to get a numbering for arbitrary sequences of natural numbers:

Theorem 2.6 [Numbering of $\mathbb{N}^n$]

We define the functions $\tau_n : \mathbb{N}^n \to \mathbb{N}$ inductively:

Basis: $\tau_1(x) = x$.

Induction: $\tau_{n+1}(x_1, x_2, \ldots, x_{n+1}) = \tau(x_1, \tau_n(x_2, \ldots, x_{n+1})$

1. *τ_n is bijective.*
2. *τ_n is primitive recursive.*
3. *The associated projection functions π_i^n, such that $\pi_i^n(\tau_n(x_1, \ldots, x_n)) = x_i$ $(i = 1, \ldots, n)$ are primitive recursive.*

□

We write also $(z)_i$ for $\pi_i^n(z)$.

Theorem 2.7 [Numbering of $\mathbb{N}^*$]

Let $\mathbb{N}^*$ be the set of all finite sequences of natural numbers. We define the functions $\tau_* : \mathbb{N}^* \to \mathbb{N}$ inductively:

Basis: $\tau_*() = 0$.

Induction: $\tau_*(x_1, x_2, \ldots, x_n) = \tau(n-1, \tau_n(x_1, x_2, \ldots, x_n)) + 1$

1. *τ_* is bijective.*

□

Definition 2.11 [Gödel Number for Sequences] *Let $(x_1, x_2, \ldots, x_n) \in \mathbb{N}^*$ be a sequence. The* Gödel number *$\langle x_1, x_2, \ldots, x_n \rangle$ of that sequence is the natural number $\tau_*(x_1, x_2, \ldots, x_n)$.*

We can use Gödel numbering to extend the notion of primitive recursive functions to functions of $\mathbb{N}^n$ to $\mathbb{N}^m$.

Definition 2.12 [Simultaneous Primitive Recursion] *Let $f_1^{(n+1)}, \ldots, f_k^{(n+1)} (k \geq 1)$ be $n+1$ ari functions, $g_1^{(n)}, \ldots, g_k^{(n)}$ be n ari functions and $h_1^{(n+1+k)}, \ldots, h_k^{(n+1+k)}$ be $n+1+k$ ari functions. We call* simultaneous primitive recursion *following recursion scheme ($1 \leq i \leq k$):*

$$\begin{array}{rcl} f_i(\vec{x}, 0) & = & g_i(\vec{x}) \\ f_i(\vec{x}, m+1) & = & h_i(\vec{x}, m, f_1(\vec{x}, m), \ldots, f_k(\vec{x}, m)) \end{array}$$

We use the notation $(f_1, \ldots, f_k) = SPR(g_1, \ldots, g_k, h_1, \ldots, h_k)$

Theorem 2.8 [Simultaneous primitive recursion] *If the functions* g_i *and* h_i $(1 \leq i \leq k)$ *are primitive recursive, then the functions* f_i $(1 \leq i \leq k)$ *such that* $(f_1, \ldots, f_k) = SPR(g_1, \ldots, g_k, h_1, \ldots, h_k)$ *are primitive recursive.*

Proof: Define $f : \mathbb{N}^n \to \mathbb{N}$ as

$$\begin{array}{rcll} f(\vec{x}, 0) & = & \tau_k & (\\ & & & \quad g_1(\vec{x}), \cdots, g_k(\vec{x}) \\ & & &) \\ f(\vec{x}, m+1) & = & \tau_k & (\\ & & & \quad h_1(\vec{x}, m, \pi_1^k(f(\vec{x}, m)), \ldots \pi_k^k(f(\vec{x}, m)), \\ & & & \quad \cdots \\ & & & \quad h_k(\vec{x}, m, \pi_1^k(f(\vec{x}, m)), \ldots \pi_k^k(f(\vec{x}, m)), \\ & & &) \end{array}$$

Then f is primitive recursive, and $f_i = SUB(\pi_i^k, f)$ is also primitive recursive (we have been using the notation $\vec{x}$ for $(x_1, \ldots, x_n)$). $\square$

2.3.2 Gödel Numbers for Recursive Functions

To each recursive function f, we can now associate a Gödel number $\ulcorner f \urcorner$. The Gödel number consists in a code for the type of the function, the arity and optional supplementary informations.

1. **Constant:** $\ulcorner C_i^n \urcorner = \ulcorner 3, n, i \urcorner$
2. **Successor:** $\ulcorner S \urcorner = \ulcorner 1, 1 \urcorner$
3. **Projection:** $\ulcorner P_i^n \urcorner = \ulcorner 2, n, i \urcorner$
4. **Composition:** Let $g^{(m)}$ $h_1^{(n)}, \ldots, h_m^{(n)}$ be functions, then $\ulcorner SUB(g, h_1, \ldots, h_m) \urcorner = \langle 4, n, \ulcorner g \urcorner, \langle \ulcorner h_1 \urcorner, \ldots, \ulcorner h_m \urcorner \rangle \rangle$
5. **Primitive Recursion:** Let $g^{(n)}$ and $h^{(n+2)}$ be functions then $\ulcorner PR(g, h) \urcorner = \langle 5, n+1, \ulcorner g \urcorner, \ulcorner h \urcorner \rangle$
6. **Unbounded Search:** Let $g^{(n+1)}$ be function then $\ulcorner \mu(g) \urcorner = \langle 6, n, \ulcorner g \urcorner \rangle$

Notation 2.5 [Gödel Numbering] We use the notation φ for that Gödel numbering scheme and we write φ_n for the function corresponding to the Gödel number n.

Example 2.11 [Gödel Numbering] Let compute the Gödel number $\ulcorner P_1^5 \urcorner$.

By definition $\ulcorner P_1^5 \urcorner = \langle 2, 1, 5 \rangle$.

Now

$$
\begin{aligned}
\ulcorner P_1^5 \urcorner &= \langle 2,5,1\rangle \\
&= \tau^*(2,5,1) \\
&= \tau(2,\tau_3(2,5,1))+1 \\
&= \tau(2,\tau(2,\tau_2(5,1)))+1 \\
&= \tau(2,\tau(2,\tau(5,\tau_1(1))))+1 \\
&= \tau(2,\tau(2,\tau(5,1)))+1 \\
&= \tau(2,\tau(2,22))+1 \\
&= \tau(2,322)+1 \\
&= 52973
\end{aligned}
$$

Definition 2.13 [Derivation] *Each $f \in F(PRIM)$ is defined as a sequence of statements starting with the basic functions constant, successor and projection, and then using the composition and the primitive recursion rules.*

We call this a derivation *of the function.*

Each step of a derivation of the function f corresponds to a step in the computation of the Gödel number $\ulcorner f \urcorner$.

Example 2.12 [Derivation] Derivation for $add(2,3)$ (informal):

$$
\begin{aligned}
add(2,3) &= S(P_1^3(add(1,3),1,3)) \\
&= S(P_1^3(S(P_1^3(add(0,3),0,3)),1,3)) \\
&= S(P_1^3(S(P_1^3(P_1^1(3),0,3)),1,3))
\end{aligned}
$$

Exercise 2.6 [Gödel Number] *Compute the Gödel Number of the function $SUB(S,S)$.*

2.4 Computability

2.4.1 Primitive Recursive Functions

We show now that primitive recursive functions are mechanically computable. Let f be an n-ari primitive recursive function and suppose that we want to compute $z = f(x_1,\ldots,x_n)$. The computation will consist in some steps that will be coded as 5-tuples (e,x,r,c,s) where e is the Gödel number of f, x is the code for the arguments of f, r is the result of the computation, c the code for the computation of the arguments of f and s are the number of computation steps.

Definition 2.14 [Computation of Primitive Recursive Functions]

1. **Constant:** *The code for $C_i^n(\vec{x})$ is*

$$\langle \ulcorner C_i^n \urcorner, \langle x_1,\ldots,x_n\rangle, i, \langle\rangle, 1\rangle$$

2. **Successor:** *The code for $S(x)$ is*

$$\langle \ulcorner S \urcorner, \langle x\rangle, S(x), \langle\rangle, 1\rangle$$

3. **Projection:** *The code for $P_i^n(\vec{x})$ is*

$$\langle \ulcorner P_i^n \urcorner, \langle x_1,\ldots,x_n\rangle, x_i, \langle\rangle, 1\rangle$$

4. Composition: *The code for* $SUB(g, h_1, \ldots, h_m)(\vec{x})$ *is*

$$\langle \ulcorner SUB(g, h_1, \ldots, h_m) \urcorner, \langle x_1, \ldots, x_n \rangle, y, \langle b_g, b_{h_1}, \ldots, b_{h_m} \rangle, s \rangle$$

Where b_{h_i} $i = 1, \ldots, m$ *are the codes for* $h_i(x_1, \ldots, x_n)$, b_g *is the code for* $g(u_1, \ldots, u_m)$ *such that* $h_i(x_1, \ldots, x_n) = u_i$ *and* $g(u_1, \ldots, u_m) = y$ *and*

$$s = (b_h)_5 + \sum_{i=1}^{m} (b_{h_i})_5$$

5. Primitive Recursion: *The code for* $PR(g, h)(\vec{x})$ *is*

$$\langle \ulcorner PR(g, h) \urcorner, \langle z, x_1, \ldots, x_n \rangle, y, \langle b_0, \ldots, b_z \rangle, s \rangle$$

Where b_0 *is the codes for for* $g(x_1, \ldots, x_n)$ *and* b_i $i = 1, \ldots, z$ *are the codes for* $h(x_1, \ldots, x_n, i, (b_i)_3)$, $(b_z)_3 = y$ *and*

$$s = \sum_{i=0}^{z} (b_i)_5$$

Remark 2.5 [Computation] This defines a primitive recursive predicate $COMPUTATION(\langle e, x_1, \ldots, x_n, r, c, s \rangle)$ stating that the function $f(x_1, \ldots, x_n) = c$ corresponding to the Gödel number e can be computed in s steps with result r.

Theorem 2.9 [Computability] *Recursive functions are mechanically computable.* □

Remark 2.6 [Computation] Each function defined with primitive recursion can be computed using a `for` loop. Let f be defined by primitive recursion, i.e. $f(\vec{x}, 0) = h(\vec{x})$ and $f(\vec{x}, S(n)) = g(\vec{x}, n, f(\vec{x}))$. We shall use the notation $f = PR(g, h)$. The following algorithm can be used to compute $z = f(\vec{x}, y)$:

$z = g(\vec{x})$
for $(i = 1, \ldots, y)$ {
    $z = h(\vec{x}, i, z)$
}

Theorem 2.10 [Non Primitive Recursive Function] *There are mechanically computable functions that are not primitive recursive.*

Proof: Let $f_1 \in F(PRIM)$ in one variable corresponding to the the smallest number giving a legal computation and then let $f_2 \in F(PRIM)$ be in one variable corresponding to the second smallest number, etc. Now let

$$v(n) = f_n(n) + 1$$

v is mechanically computable, since once we have found a a computation for f_n we can compute it on any input. We claim that $v \notin F(PRIM)$. If $v \in F(PRIM)$, then $v = f_m$ for some number m. But $v(m) = f_m(m) + 1 \neq f_m(m)$ contradiction! □

Remark 2.7 [Non Primitive Recursive] The method used in the proof of theorem 2.10 is called *diagonalisation*. The values $f_i(j)$ of the functions f_i can be described in an infinite two-dimensional array. The values of v are constructed in such a way that they differ from those of f_i on the main diagonal of the array, i.e. $v(i) \neq f_i(i)$.

2.4.2 Recursive Functions

Recursive functions are mechanically computable but it is not obvious to determine whether a computation defines a total function f and thus defines a total recursive function. The problem is being that it is difficult to decide whether the defined function is total, i.e. if $\mu(g)(x_1, \ldots, x_k)$ exists for all $(x_1, \ldots, x_k) \in \mathbb{N}^k$.

Definition 2.15 [Computation of Recursive Functions]

6. Unbounded Search: *The code for* $\mu(g)(\vec{x})$ *is*

$$\langle \ulcorner \mu(g) \urcorner, \langle x_1, \ldots, x_n \rangle, y, \langle b_0, \ldots, b_y \rangle, s \rangle$$

Where b_i $i = 1, \ldots, y$ *are the code for for* $g(\vec{x}, i)$, $(b_i)_3 > 0$, $(b_y)_3 = 0$ *and*

$$s = \sum_{i=0}^{y} (b_i)_5$$

Remark 2.8 [Computation] Each function defined with unbounded search can be computed using a `while` loop. The following algorithm can be used to compute $j = f(\vec{x}) = \mu(g)(\vec{x})$:

$y = 0$
while $(g(\vec{x}, y) \neq 0)$ {
    $y = y + 1$
}
$\mu(g)(\vec{x}) = y$

Note that if the function g does not have the value 0, the algorithm will enter an endless loop, thus it will not halt (see also 2.6.2).

Definition 2.16 [Computable Function] *A function is* computable *if and only if it is recursive.*

2.5 Functions, Sets and Languages

Definition 2.17 [Characteristic Function] *Let* S *be a set, then we can identify* S *with a predicate* χ_S*, such that*

$$x \in S \Leftrightarrow \chi_S(x) = 1$$

The function χ_S *is called the* characteristic function *of the set* S*.*

Definition 2.18 [Recursive Set] *A set* S *is* (primitive) recursive *if and only if its characteristic function* χ_S *is (primitive) recursive.*

2.5.1 Recursively Enumerable Sets

A set S is recursive if given x it is computable to test whether $x \in S$, i.e. if its characteristic function χ_S is recursive. Another interesting class of sets is the class of sets which can be listed mechanically.

Definition 2.19 [Recursively Enumerable Set] *The set S is* recursively enumerable *if it is the domain of a partial recursive function f. That is*

$$n \in S : f(n) \text{ is defined}$$

Theorem 2.11 [Recursive/Recursively Enumerable] *A set A is recursively enumerable if and only if there is a recursive set B such that $x \in A \Leftrightarrow \exists y : (x, y) \in B$.* □

2.5.2 Languages

A language is a set L of strings over an alphabet Σ. Using the characteristic function χ_L, we can say that L is recursive, if χ_L is recursive, L is primitive recursive, if χ_L is primitive recursive.

Example 2.13 [Prime Numbers] The set of the prime numbers is primitive recursive since it has following primitive recursive function as characteristic function:

$$prime(n) \underset{def}{=} n > 1 \wedge \neg(\exists m) 1 < m < n : m|n$$

Exercise 2.7 [Even Numbers] *Prove that $2\mathbb{N} = \{2n : n \in \mathbb{N}$ is a recursive set.*

2.6 Universal Function

2.6.1 Normal Form of Kleene

Theorem 2.12 [Normal Form of Kleene] *For each $n \in \mathbb{N}$ there is a function $U^{(1)}$ and a primitive recursive predicate $T^{(n+2)}$, such that for all recursive functions $\psi^{(n)}$ there is a number e such that*

$$\forall (x_1, \ldots, x_n) \in \mathbb{N}^n : \psi(\vec{x}) = U(\mu y T(e, \vec{x}, y))$$

Proof: Using the predicate $COMPUTATION(\langle e, \vec{x}, r, c, s\rangle)$ of remark 2.5 we define

$$T(e, \vec{x}, y) = 1 \perp (COMPUTATION(b) \wedge (b)_1 = e \wedge (b)_2 = \langle x_1, \ldots, x_n\rangle)$$

and

$$U(y) = (y)_3$$

□

Note that the functions $\phi(e, \vec{x}, y) = U(\mu y : T(e, \vec{x}, y))$ define a Gödel numbering of the (partial) recursive functions.

Remark 2.9 [Invalid Gödel Numbers] A lot of natural numbers are not Gödel numbers of a recursive function. But there is a numbering $\hat{\phi}$ and an injective function h such that

$$\hat{\phi}(e, \vec{x}) = \phi(h(e), \vec{x})$$

Note that there is a primitive recursive predicate gn such that $gn(e) = 1$ if and only if e is a valid Gödel number. We just can choose h as

$$\begin{aligned} h(0) &= \mu y(1 \perp gn(y)) \\ h(n+1) &= \mu y(1 \perp (gn(y) \wedge y > h(n))) \end{aligned}$$

The function h is called *translationfunction*.

Definition 2.20 [Universal Function] *Let φ be a Gödel numbering of the set of computable functions, the* Universal Function u_φ *is the partial function*

$$u_\varphi : \mathbb{N}^2 \to \mathbb{N}$$

defined as

$$u_\varphi(i, x) := \varphi_i(x) \qquad i, x \in \mathbb{N}$$

Here a nice consequence of the theorem of Kleene.

Theorem 2.13 [Projection] *Let $A \subset \mathbb{N}^n$ be a set. Following statements are equivalent:*

1. *A is recursively enumerable.*
2. *A is the projection of a primitive recursive set $B \subset \mathbb{N}^{n+1}$, i.e.*

$$(x_1, \ldots, x_n) \in A \Leftrightarrow \exists y : (x_1, \ldots, x_n, y) \in B$$

Proof: ($\Rightarrow$) By definition, there is a partial recursive function f such that $(x_1, \ldots, x_n) \in A$ if and only if $f(x_1, \ldots, x_n)$ is defined. That is using the theorem of Kleene $f(x_1, \ldots, x_n)$ is defined if and only if $\exists y : T(e, x_1, \ldots, x_n, y)$. Let

$$B = \{(x_1, \ldots, x_n, y) : T(e, x_1, \ldots, x_n, y)\}$$

Then A is the projection of B.

($\Leftarrow$) If A is the projection of the recursive set B then the function

$$f(x_1, \ldots, x_n) = \mu y(B(x_1, \ldots, x_n, y))$$

is a partial recursive function and $(x_1, \ldots, x_n) \in A$ if and only if $f(x_1, \ldots, x_n)$ is defined. It follows that A is recursively enumerable. $\square$

The last theorem says that recursively enumerable sets are just recursive sets plus an existential quantifier. We will later see that there is a similar relationship between the complexity classes P and NP.

Theorem 2.14 [Recursive Sets] *A set A is recursive if and only if A and $\overline{A}$ are recursively enumerable.*

Proof: ($\Rightarrow$) If A is recursive then

$$(x_1, \ldots, x_n) \in A \Leftrightarrow \exists y((x_1, \ldots, x_n) \in A$$

it follows that A is the projection of the recursive set

$$B = \{(x_1, \ldots, x_n, y) : (x_1, \ldots, x_n) \in A\}$$

The same for $\overline{A}$.

($\Leftarrow$) If A and $\overline{A}$ are recursively enumerable, there are recursive sets B_1 and B_2 whose projections are A and $\overline{A}$ respectively. Now

$$(x_1, \ldots, x_n) \in A \Leftrightarrow (x_1, \ldots, x_n, y_x) \in B_1 \text{ for } y_x = \mu y((x_1, \ldots, x_n, y) \in B_1 \cup B_2)$$

□

2.6.2 Halting Problem

Definition 2.21 [Halting Set] *The* halting problem *is the set H defined by*

$$H = \{(i, j) : \varphi_i(j) \textit{ is defined}\}$$

The diagonal halting problem *is the set H_D is defined by*

$$H_D = \{i : \varphi_i(i) \textit{ is defined}\}$$

Note that the sets H and H_D are recursively enumerable. This follows directly from the definition 2.19.

According to remark 2.8, the value of $\mu(g)$ can be computed using a, `while`-loop. The execution of the loop will stop if $g(x_1, \ldots, x_k, y) = 0$ for some y. If the value of g is never 0, the loop will never halt. This is the reason why the problem above is called halting problem.

Theorem 2.15 [Non Computable] *There is a function $f : \mathbb{N} \to \{0, 1\}$ that is not recursive.*

Proof: Consider the (partial) recursive function h

$$h(i) = \begin{cases} \varphi_i(i) & \text{if } \varphi_i(i) \textit{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Now define the function f by

$$f(n) = \begin{cases} 1 & \text{if } h(n) \, is \; undefined \\ undefined & \text{otherwise} \end{cases}$$

We claim that the function f is not recursive. If f would be recursive, then $f = \varphi_i$ for some i. If $f(i)$ is defined, then $f(i) = 1$ but by definition of f, $f(i) = \varphi_i(i) = h(i)$ is undefined. If $f(i)$ is undefined, but by definition of f, $f(i) = \varphi_i(i) = h(i)$ is defined. We get a contradiction in both cases. □

It follows from the last theorem, that it is not computable, whether a given function is computable or not. In fact, we have been proving following theorem.

Theorem 2.16 [Halting Problem] *The halting set H is not recursive.*

Proof: Assume that H is recursive. By definition, its characteristic function χ_H is recursive.

$$\chi_H(e, x) = \begin{cases} 1 & \text{if } \varphi_e(x) \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

If χ_H is recursive, so $1 \perp \chi_H$, the characteristic function of $\overline{H}$. But we know from theorem 2.15 that it is not possible. □

Theorem 2.17 [Recursive/Recursively enumerable] *Let L be a set. The following statements are equivalent:*

1. *L is recursive.*
2. *L and $\overline{L}$ are recursively enumerable.*

□

Exercise 2.8 [Invalid Gödel Number] *Find out a number n that is not a Gödel number of a (partial) recursive function.*

2.7 The Recursion Theorem

Can programs reproduce oneself? While most assembler programs can do this since their code is contained in the memory cells that they can manipulate, it is not clear whether JAVA programs or Turing machines can do this. We will answer this question affirmatively. Even ordinary sentences can reproduce themselves, as the following example shows:

Write down the following sentence inside the quotes two times, the second time in quotes!
'Write down the following sentence inside the quotes two times, the second time in quotes!'

While it is certainly an interesting philosophical question whether programs can reproduce themselves , the recursion theorem, from which the result above follows, has a lot of other applications.

Theorem 2.18 [S_n^m Theorem] *For all $m \geq 1$ and $n \geq 1$ there is a total recursive $(m+1)$-ari function S_n^m such that*

$$\forall e \in \mathbb{N}\, \forall (x_1, \ldots, x_n) \in \mathbb{N}^n\, \forall (y_1, \ldots, y_m) \in \mathbb{N}^m : \varphi_{S_n^m(e,y_1,\ldots,y_m)}(x_1, \ldots, x_n) = \varphi_e(y_1, \ldots, y_m, x_1, \ldots, x_n)$$

Proof: We use the translation function h of remark 2.9 and define

$$S_n^m(e, y_1, \ldots, y_m) = h(\langle e, y_1, \ldots, y_m \rangle)$$

It follows that

$$\begin{aligned} S_n^m(e, y_1, \ldots, y_m)(x_1, \ldots, x_n) &= h(\langle e, y_1, \ldots, y_m \rangle)(x_1, \ldots, x_n) \\ &= \varphi(h(\langle e, y_1, \ldots, y_m \rangle), x_1, \ldots, x_n) \\ &= \varphi_e(y_1, \ldots, y_m, x_1, \ldots, x_n) \end{aligned}$$

□

Theorem 2.19 [Recursion Theorem] *For all $n \geq 1$ and for all $(n+1)$-ari partial recursive function f there is a number g such that*

$$f(g, x_1, \ldots, x_n) = \varphi_g(x_1, \ldots, x_n)$$

Proof: The function h given by

$$h(y, z_1, \ldots, z_n) = f(S_n^1(y, y), z_1, \ldots, z_n)$$

is partial recursive. Let e be the Gödel number of h, i.e. $\varphi_e = h$. From the S_n^m Theorem, it follows that

$$\varphi_e(y, z_1, \ldots, z_n) = \varphi_{S_n^1(e,y)}(z_1, \ldots, z_n)$$

If we now set $y = e$ and $g = S_n^1(e, e)$, we get

$$\begin{aligned} f(g, z_1, \ldots, z_n) &= f(S_n^1(e, e), z_1, \ldots, z_n) \\ &= h(e, z_1, \ldots, z_n) \\ &= \varphi_e(e, z_1, \ldots, z_n) \\ &= \varphi_{S_n^1(e,e)}(z_1, \ldots, z_n) \\ &= \varphi_g(z_1, \ldots, z_n) \end{aligned}$$

□

2.7.1 Applications

Halting Problem

Here is an alternative proof that the halting problem H is not decidable: Assume that H is decidable. Then following function

$$f(e,x) = \begin{cases} 0 & \text{if } \varphi_e(x) \text{ is undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

is computable since we can check whether $\varphi_e(x)$ is defined. By the Recursion Theorem, there is an index e_0 such that

$$\varphi_{e_0}(x) = f(e_0, x)$$

Assume that $\varphi_{e_0}(e_0)$ is defined. Then $f(e_0, e_0) = \varphi_{e_0}(e_0)$ is undefined by construction, a contradiction. But if $\varphi_{e_0}(e_0)$ were undefined, then $f(e_0, e_0) = \varphi_{e_0}(e_0) = 0$, a contradiction again. Thus H cannot be decidable.

Self Reference

Consider the function $s : \mathbb{N}^2 \to \mathbb{N}$ given by $s(y,z) = y = P_1^2(y,z)$. By the recursion theorem there is a number g such that

$$\varphi_g(z) = s(g,z) = g$$

This means that the function given by the Gödel number g computes the constant function with value g and in this sense outputs its own source code.

Exercise 2.9 [JAVA Self Reference] *Write a* JAVA *that outputs its own source code.*

2.8 Problem Reduction

Definition 2.22 [Problem Reduction] *For sets A and B let the notation $A <_m B$ mean that there is a recursive function f such that $x \in A \Leftrightarrow f(x) \in B$.*

The reason for the letter m on the less than sign is that one usually defines several different reductions. This particular reduction is usually referred to as a *many-one reduction.*

Theorem 2.20 [Problem Reduction] *If $A <_m B$ and B is recursive then A is recursive.*

> **Proof:** To decide whether $A <_m B$,first compute $f(x)$ and then check whether $f(x) \in B$. Since both f and B are recursive this is a recursive procedure and it gives the correct answer by the definition of $A <_m B$. □

Next let us define the hardest problem within a given complexity class.

Definition 2.23 [R.E. Complete] *A set A is* r.e.-complete *if and only if*

1. *A is recursively enumerable.*

2. If B is recursively enumerable then $B <_m A$.

Theorem 2.21 [Halting R.E. Complete] *The halting set is r.e.-complete.*

Proof: We already know that H is recursively enumerable.

To see that it is complete we have to prove that any other recursively enumerable set, A can be reduced to H. There is a recursive function ψ such that $x \in A \Leftrightarrow \psi(x)\ is\ defined$. Now there is a number e such that $\psi = \varphi_e$. It follows that

$$x \in A \Leftrightarrow \psi(x) \text{ is defined } \Leftrightarrow \varphi_e(x) \text{ is defined } \Leftrightarrow (e, x) \in H$$

Using $f(x) = (e, x)$ it follows that $A <_m H$. □

Theorem 2.22 [R.E. Complete] *If A is r.e.-complete then A is not recursive.*

Proof: Let B be a set that is recursively enumerable but not recursive (e.g.the halting problem) then by the second property of being r.e.-complete $B <_m A$. Now if A was recursive then by Theorem 2.20 we could conclude that B is recursive, contradicting the initial assumption that B is not recursive. □

Exercise 2.10 [R.E. Complete] *Prove that if A is r.e. complete, then A is not recursive. Hint: use theorem 2.20 and try to find a contradiction.*

Chapter 3

Lambda Calculus

Remark 3.1 [Papers] This chapter is essentially based on the following papers [Nip98], [Smi04] and [Sel07].

3.1 Introduction

The lambda calculus or λ-calculus shortly, is a formal system designed to investigate function, function application and recursion. It was introduced by Alonzo Church and Stephen Cole Kleene in the late 1930s and early 1940s. The calculus can be used to cleanly define what a computable function is. The question of whether two lambda calculus expressions are equivalent cannot be solved by a general algorithm, and this was the first question, even before the halting problem, for which undecidability could be proved. Lambda calculus has greatly influenced logical programming languages as Lisp and Haskell.

3.1.1 Extensional vs. intensional view of functions

What is a function? In modern mathematics, the prevalent notion is that of "functions as graphs": each function f has a fixed domain X and codomain Y , and a function $f : X \to Y$ is a set of pairs $f \subset X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$. Two functions $f, g : X \to Y$ are considered equal if they yield the same output on each input, i.e., $\forall x \in X : f(x) = g(x)$ This is called the *extensional* view of functions, because it specifies that the only thing observable about a function is how it maps inputs to outputs.

However, before the 20th century, functions were rarely looked at in this way. An older notion of functions as that of "functions as rules". In this view, to give a function means to give a rule for how the function is to be calculated. Often, such a rule can be given by a formula, for instance, the familiar $f(x) = x^2$ or $g(x) = \sin(e^x)$ from calculus. As before, two functions are extensionally equal if they have the same input-output behavior; but now we can also speak of another notion of equality: two functions are *intensionally* equal if they are given by (essentially) the same formula.

When we think of functions as given by formulas, it is not always necessary to know the domain and codomain of a function. Consider for instance the function $f(x) = x$. This is, of course, the identity function. We may regard it as a function $f : X \to X$ for any set X.

In most of mathematics, the "functions as graphs" paradigm is the most elegant and appropriate way of dealing with functions. Graphs define a more general class of functions, because it

includes functions that are not necessarily given by a rule. Thus, when we prove a mathematical statement such as "any differentiable function is continuous", we really mean this is true for all functions (in the mathematical sense), not just those functions for which a rule can be given.

On the other hand, in computer science, the "functions as rules" paradigm is often more appropriate. Think of a computer program as defining a function that maps input to output. Most computer programmers (and users) do not only care about the extensional behavior of a program (which inputs are mapped to which outputs), but also about how the output is calculated: How much time does it take? How much memory and disk space is used in the process? How much communication bandwidth is used? These are intensional questions having to do with the particular way in which a function was defined.

3.1.2 The lambda calculus

The lambda calculus is a theory of functions as formulas. It is a system for manipulating functions as expressions. Another well-known language of expressions is arithmetic.

The lambda calculus extends the idea of an expression language to include functions. Where we normally write

$$\text{Let } f \text{ be the function } x \mapsto x^2. \text{ Then consider } A = f(5)$$

in the lambda calculus we just write

$$A = (\lambda x.x^2)(5)$$

The expression $\lambda x.x^2$ stands for the function that maps x to x^2 (as opposed to the statement that x is being mapped to x^2). As in arithmetic, we use parentheses to group terms.

One advantage of the lambda notation is that it allows us to easily talk about higher-order functions, i.e., functions whose inputs and/or outputs are themselves functions. An example is the operation $f \to f \circ f$ in mathematics, which takes a function f and maps it to $f \circ f$, the composition of f with itself. In the lambda calculus, $f \circ f$ is written as

$$\lambda x.f(f(x))$$

and the operation that maps f to $f \circ f$ is written as

$$\lambda f.\lambda x.f(f(x))$$

The evaluation of higher-order functions can get somewhat complex; as an example, consider the following expression:

$$((\lambda f.\lambda x.f(f(x)))(\lambda y.y^2))(5)$$

Now let us evaluate this expression

$$\begin{aligned} ((\lambda f.\lambda x.f(f(x)))(\lambda y.y^2))(5) &= (\lambda x.(\lambda y.y^2)((\lambda y.y^2)x))(5) \\ &= ((\lambda y.y^2)((\lambda y.y^2)5)) \\ &= ((\lambda y.y^2)(25)) \\ &= 625 \end{aligned}$$

Exercise 3.1 [Evaluating λ-expressions]

Evaluate the following λ-expression

a) $(((\lambda f.\lambda x.f(f(f(x))))(\lambda g.\lambda y.g(g(y))))(\lambda z.z+1))(0)$

b) $(\lambda x.xx)(\lambda y.yy)$

3.1.3 Lambda calculus and computability

In the 1930's, several people were interested in the question: what does it mean for a function $f : N \to N$ to be computable? An informal definition of computability is that there should be a pencil-and-paper method allowing a trained person to calculate $f(n)$, for any given n. The concept of a pencil-and-paper method is not so easy to formalize. Three different researchers attempted to do so, resulting in the following definitions of computability:

1. **Turing** defined an idealized computer we now call a Turing machine , and postulated that a function is computable (in the intuitive sense) if and only if it can be computed by such a machine.
2. **Gödel** defined the class of general recursive functions as the smallest set offunctions containing all the constant functions, the successor function, and closed under certain operations (such as compositions and recursion). He postulated that a function is computable (in the intuitive sense) if and onlyif it is general recursive.
3. **Church** defined an idealized programming language called the lambda calculus , and postulated that a function is computable (in the intuitive sense) if and only if it can be written as a lambda term.

It was proved by Church, Kleene, Rosser, and Turing that all three computational models were equivalent to each other, i.e., each model defines the same class of computable functions. Whether or not they are equivalent to the "intuitive" notion of computability is a question that cannot be answered, because there is no formal definition of "intuitive computability". The assertion that they are in fact equivalent to intuitive computability is known as the *Church-Turing thesis* .

3.1.4 Connections to computer science

The lambda calculus is a very idealized programming language; arguably, it is the simplest possible programming language that is Turing complete . Because of its simplicity, it is a useful tool for defining and proving properties of programs. Many real-world programming languages can be regarded as extensions of the lambda calculus. This is true for all functional programming languages, a class that includes Lisp, Scheme, Haskell, and ML. Such languages combine the lambda calculus with additional features, such as data types, input/output, side effects, udpateable memory, object orientated features, etc. The lambda calculus provides a vehicle for studying such extensions, in isolation and jointly, to see how they will affect each other, and to prove properties of programming language (such as: a well-formed program will not crash).

3.2 The pure lambda calculus

The lambda calculus, invented by Alonzo Church in the 1930s, is a mathematical model of computation based around the operations of "abstraction" (defining functions) and "application" (evaluating them at a point). Nothing else is permitted in the pure lambda calculus. In particular, there are no numbers, variables, or data structures of any kind.

3.2.1 The Syntax

The Syntax of the lambda calculus is very simple and is given in the following definition

Definition 3.1 [Syntax] *Assume given an infinite set of variables $\mathcal{V}$. We build an alphabet A containing $\mathcal{V}$ and the special symbols "(", ")", "λ" and ".". Let $A*$ be the set of strings over the alphabet A. The set of lamda terms is the smallest subset $\Lambda \subset A*$ such that*

- *whenever $x \in \mathcal{V}$ then $x \in \Lambda$*
- *whenever $M, N \in \Lambda$ then $(MN) \in \Lambda$*
- *whenever $x \in \mathcal{V}$ and $M \in \Lambda$ then $(\lambda x.M) \in \Lambda$.*

Alternately we can also define the Lambda expressions in BNF

Definition 3.2 [Syntax in BNF] *Assume given an infinite set V of variables, denoted by x; y; z etc. The set of lambda terms is given by the following Backus-Naur Form :*

$$\begin{array}{rcll} L & \rightarrow & x & \\ & | & (LL) & \rightarrow \\ & | & (\lambda x.L) & \end{array}$$

Note that in the definition of lambda terms, we have built in enough mandatory parentheses to ensure that every term $M \in \Lambda$ can be uniquely decomposed into subterms. This means, each term $M \in \Lambda$ is of precisely one of the forms x, (MN) or $(\lambda x.M)$. Terms of these three forms are called *variables*, *applications*, and *lambda abstractions*, respectively.

To awoid an excessive use of parenthesis we will use the following conventions:

- We omit outermost parentheses. For instance, we write MN instead of (MN).
- Application is left assoziative. So MNP means $(MN)P$ This is convenient if we apply a function to more than one argument. For instance $fxyz$ means $((fx)y)z$.
- The body of a lambda abstraction extends as far as possible to the right. For instance $\lambda x.MN$ means $\lambda x.(MN)$ and not $(\lambda x.M)N$.
- Multiple lambda abstractions can be contracted; thus $\lambda xyz.M$ means $\lambda x.\lambda y.\lambda z.M$.

Exercise 3.2 [Parenthesis]

(a) Write the following terms with as few parenthesis as possible, without changing the meaning or structure of the terms:

(i) $(\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$

(ii) $(((ab)(cd))((ef)(gh)))$

(iii) $(\lambda x.((\lambda y.(yx))(\lambda v.v)z)u)(\lambda w.w)$

(b) Restore all the dropped parentheses in the following terms, without changing the meaning or structure of the terms:

(i) $xxxx$

(ii) $\lambda x.x\lambda y.y$

(iii) $\lambda x.(x\lambda y.yxx)x$

3.2.2 Free and bound variables, α-equivalence

In a lambda term of the form $\lambda x.M$ the variable x is a *local* variable. Thus, it does not make any difference if we write $\lambda y.M$ instead. A local variable is also called a bound variable. So the terms $\lambda x.M$ and $\lambda y.M$ which differ only in the name of their bound variable, are essentially the same. We will call such terms α–equivalent and will write $M =_\alpha N$.

An occurrence of a variable x inside a term of the form $\lambda x.N$ is said to be bound. The corresponding λx is called a binder, and we say that the subterm N is the scope of the binder. A variable occurrence that is not bound is free. Thus, for example, in the term

$$M \equiv (\lambda x.xy)(\lambda y.yz)$$

x is bound, but z is free. The variable y has both a free and a bound occurrence. The set of free variables of M is y, z

Next we define the set of free variables of a lambda term.

Definition 3.3 [The set of free variables] *The set of free variables of a lambda term M is denoted by $FV(M)$, and it is defined formaly as follows:*

$$\begin{array}{lcl} FV(x) & = & x \\ FV(MN) & = & FV(M) \cup FV(N) \\ FV(\lambda x.M) & = & FV(M) \setminus x \end{array}$$

Before we define α–equivalence we need to define what it means to *rename* a variable in a term.

Definition 3.4 [Renaming of a variable] *If x, y are variables, and M is a term, we write $M\{y/x\}$ for the result of renaming x as y in the term M. Formaly renaming is defined as follows:*

$$\begin{array}{lcll} x\{y/x\} & \equiv & y & \\ z\{y/x\} & \equiv & z & \text{if } x \neq z \\ (MN)\{y/x\} & \equiv & (M\{y/x\}N\{y/x\}) & \\ (\lambda x.M)\{y/x\} & \equiv & (\lambda y.M\{y/x\}) & \\ (\lambda z.M)\{y/x\} & \equiv & (\lambda z.M\{y/x\}) & \text{if } x \neq z \end{array}$$

Note that this kind of renaming replaces all occurrences of x by y, whether free, bound, or binding. We will only apply it in cases where y does not already occur in M.

Definition 3.5 [α–equivalence] *We define α–equivalence to be the smallest congruence relation $=_\alpha$ on lambda terms, such that for all terms M and all variables y that do not occur in M,*

$$\lambda x.M =_\alpha \lambda y.(M\{y/x\})$$

Remark 3.2 [congruence] Recall that a relation on lambda terms is an equivalence relation if it is reflexive, symetric and transitive. It is a conruence if it also satifies the following rules:

$$\begin{array}{lll} (cong) & M =_\alpha M' \wedge N =_\alpha N' & \Rightarrow \quad MN =_\alpha M'N' \\ (\xi) & M =_\alpha M' & \Rightarrow \quad \lambda x.M =_\alpha \lambda x.M' \end{array}$$

It is easy to prove by induction that any lambda term is α–equivalent to another term in which the names of all bound variables are distinct from each other and from any free variables. Thus, when we manipulate lambda terms in theory and in practice, we can (and will) always assume without loss of generality that bound variables have been renamed to be distinct. This convention is called *Barendregt's* variable convention.

Exercise 3.3 [α–equivalence] *Analize (in the following style) each of these λ expressions to identify its free and bound variables, and those in its subexpressions.*

a) $\lambda x.\lambda y.(\lambda x.y\lambda y.x)$

$$\begin{array}{llll} x \text{ bound at} & \{x\} & in & \lambda x.\lambda y.(\lambda x.y\lambda y.\{x\}) \\ x \text{ free at} & \{x\} & in & \lambda y.(\lambda x.y\lambda y.\{x\}) \\ & & & (\lambda x.y\lambda y.\{x\}) \\ \ldots & & & \end{array}$$

b) $\lambda x.(x(\lambda y.(\lambda x.xy)x))$

c) Use α–conversion to ensure unique names in the original expressions given in a) and b)

3.2.3 β-reduction and β-equivalence

First we will consider an operation, called *substitution*, which allows us to replace a variable by a lambda term. We will write $M[N/x]$ for the result of replacing x by N in M. The definition of substitution is complicated by two circumstances:

1. We should only replace free variables. This is because the names of bound variables are considered immaterial, and should not affect the result of a substitution. Thus, $x(\lambda xy.x)[N/x] is N(\lambda xy.x)$, and not $N(\lambda xy : N)$.

2. We need to avoid unintended "capture" of free variables. Consider for example the term $M \equiv \lambda x.yx$, and let $N \equiv \lambda z.xz$. Note that x is free in N and bound in M. What should be the result of substituting N for y in M? If we do this naively, we get

   $$M[N/y] = (\lambda x.yx)[N/y] = \lambda x.Nx = \lambda x.(\lambda z.xz)x$$

   However, this is not what we intended, since the variable x was free in N and during the substitution, it got bound. We need to account for the fact that the x that was bound in M was not the "same" x as the one that was free in N. The proper thing to do is to rename the bound variable before the substitution:

   $$M[N/y] = (\lambda x'.yx')[N/y] = \lambda x'.Nx' = \lambda x'.(\lambda z.xz)x'$$

The best thing is to pick a variable from $\mathcal{V}$) that has not been used yet as the name of the bound variable. This is always possible, because the set of variable $\mathcal{V}$ is infinite. A variable that is currently unused is called *fresh*. We can now give a formal definition of substitution.

Definition 3.6 [Substitution]

The (capture-avoiding) substitution of N for free occurrences of x in M, in symbols $M[N/x]$, is defined as follows:

$$\begin{array}{lcll} x[N/x] & \equiv & N & \\ y[N/x] & \equiv & y & \text{if } x \neq y \\ (MP)[N/x] & \equiv & (M[N/x])(P[N/x]) & \\ (\lambda x.M)[N/x] & \equiv & M & \\ (\lambda y.M)[N/x] & \equiv & \lambda y(M[N/x]) & \text{if } x \neq y \text{ and } y \notin FV(M) \\ (\lambda y.M)[N/x] & \equiv & \lambda y'.(M[y/y'])[N/x] & \text{if } x \neq y, y \in FV(M) \text{ and } y' \text{ fresh} \end{array}$$

Remark 3.3 [α–equivalence]

From now on, unless stated otherwise, we identify lambda terms up to α–equivalence. This means, when we speak of lambda terms being "equal", we mean that they are α–equivalent. Formally, we regard lambda terms as equivalenceclasses modulo α–equivalence. We will often use the ordinary equality symbol $M = N$ to denote α–equivalence.

The process of evaluating lambda terms is called *β–reduction*. A term of the form $(\lambda x.M)N$ wich consists of a lambda abstraction applied to another term is called a *β–redex* . We say that it reduces to $M[N/x]$ and we call the later term the *reduct* . We reduce lambda terms by finding a subterm that is a redex, and then replacing that redex by its reduct. As long as a redex exists in the term we can continue to reduce that term. A lambda term without any β–redex is said to be in *β–normalform* .

Example 3.14 [β–reduction] For example, the lambda term $(\lambda x.y)((\lambda z.zz)(\lambda w.w))$ can be reduced as follows. Here, we underline each redex just before reducing it:

$$\begin{array}{lll} (\lambda x.y)(\underline{(\lambda z.zz)(\lambda w.w)}) & \to_\beta & (\lambda x.y)(\underline{(\lambda w.w)(\lambda w.w)}) \\ & \to_\beta & \underline{(\lambda x.y)(\lambda w.w)} \\ & \to_\beta & y \end{array}$$

The last term y has no more redex and is thus in normal form. But the same term can also be reduced in one step by the reduction

$$\underline{(\lambda x.y)((\lambda z.zz)(\lambda w.w))} \to_\beta y$$

As we can see from the example

- reducing a redex can create new redexes,
- reducing a redex can delete some other redexes,
- the number of steps it takes to reduce a term to a normalform may differ depending on the order the redexes are choosen
- We can also see that the final result, y, does not seem to depend on the order in which the redexes are reduced. In fact, this is true in general, as we will prove later.

If M and M' are terms such that $M \twoheadrightarrow_\beta M'$, and if M' is in normal form, then we say that M *evaluates* to M'. So we can say the term $(\lambda x.y)(\underline{(\lambda z.zz)(\lambda w.w)})$ evaluates to y .

Not every lambda term has a normalform. Some terms can be reduced forever without reaching a normalform. Here is an example.

$$\begin{array}{lll} (\lambda x.xx)(\lambda y.yyy) & \to_\beta & (\lambda y.yyy)(\lambda y.yyy) \\ & \to_\beta & (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ & \to_\beta & (\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy)(\lambda y.yyy) \\ & \to_\beta & \ldots \end{array}$$

This example also shows that the size of a lambda term need not decrease during reduction; it can increase, or remain the same like the term $(\lambda x.xx)(\lambda x.xx)$, A term with an infinite reduction sequence is calle divergent

We are now ready to define β–reduction and β–equivalence fromaly

Definition 3.7 [Single step β–reduction] *We define single step β–reduction to be the smallest relation $\to_\beta$ on terms satisfying:*

$$\begin{array}{lllll} (beta) & (\lambda x.M)N \to_\beta M[N/x] & & & \\ (cong_1) & M \to_\beta M' & \Rightarrow & MN \to_\beta M'N \\ (cong_2) & N \to_\beta N' & \Rightarrow & MN \to_\beta MN' \\ (\xi) & M \to_\beta M' & \Rightarrow & \lambda x.M \to_\beta \lambda x.M' \end{array}$$

Thus, $M \to_\beta M'$ iff M' is obtained from M by reducing a single λ–redex of M.

Definition 3.8 [β–reduction] *Formaly we define β–reduction written $M \twoheadrightarrow_{beta} M'$, to be the reflexive transitive closure of $\to_\beta$, i.e the smallest reflexive transitive relation containing $\to_\beta$.*

$M \twoheadrightarrow_{beta} M'$ means, that M reduces to M' in zero or more single steps. Finally, β–equivalence is obtained by allowing reduction steps as well as inverse reduction steps, i.e., by making $\to_\beta$ symmetric:

Definition 3.9 [β–equivalence] *We write $M =_\beta M'$ if M can be transformed into M' by zero or more reduction steps and/or inverse reduction steps. Formally, $=_\beta$ is defined to be the reflexive symmetric transitive closure of $\to_\beta$, i.e., the smallest equivalence relation containing $\to_\beta$.*

3.3 Programming in lambda calculus

One of the amazing facts about the untyped lambda calculus is that we can use it to encode data, such as booleans and natural numbers, as well as programs that operate on the data. This can be done purely within the lambda calculus, without adding any additional syntax or axioms.

We will often have occasion to give names to particular lambda terms; we willusually use boldface letters for such names.

3.3.1 Booleans

Because we have no constants in the pure lambda calculus we must define some lambda terms for them. We begin by defining two lambda terms to encode the truth values "true" and "false"

$$\begin{array}{lll} \mathbf{T} & = & \lambda xy.x \\ \mathbf{F} & = & \lambda xy.y \end{array}$$

Now lets define the logical operators **And**, **Or** and **Not**

$$\begin{array}{lcl} \textbf{And} & = & \lambda ab.ab\textbf{F} \\ \textbf{Or} & = & \lambda ab.a\textbf{T}b \\ \textbf{Not} & = & \lambda z.z\textbf{FT} \end{array}$$

Note that **T** and **F** are normal forms, so we can really say that **And TT** evaluates to **T**.We say that **And** encodes the boolean function "and". It is understood that this coding is with respect to the particular coding of "true" and "false". We don't claim that **And** MN evaluates to anything meaningful if M or N are terms other than **T** and **F**. As an Example we will verify the operator **Not**:

Example 3.15 [Evaluate NOT]

Not T:

$$\begin{array}{lcl} (\lambda z.z(\lambda xy.y)(\lambda xy.x))(\lambda xy.x) & \to_\beta & (\lambda xy.x)(\lambda xy.y)(\lambda xy.x) \\ & \to_\beta & (\lambda yxy.y)(\lambda xy.x) \\ & \to_\beta & \lambda xy.y \\ & = & \textbf{F} \end{array}$$

Not F:

$$\begin{array}{lcl} (\lambda z.z(\lambda xy.y)(\lambda xy.x))(\lambda xy.y) & \to_\beta & (\lambda xy.y)(\lambda xy.y)(\lambda xy.x) \\ & \to_\beta & (\lambda y.y)(\lambda xy.x) \\ & \to_\beta & \lambda xy.x \\ & = & \textbf{T} \end{array}$$

We can define the operator **ifThenElse** $= \lambda x.x$. This term behaves like an "if-then-else" function – specifically, we have:

$$\begin{array}{lcl} \textbf{ifThenElse T}MN & \twoheadrightarrow_\beta & M \\ \textbf{ifThenElse F}MN & \twoheadrightarrow_\beta & N \end{array}$$

for all terms M and N. We will verify this:

ifThenElse TMN

$$\begin{array}{lcl} (\lambda x.x)(\lambda xy.x)MN & \to_\beta & (\lambda xy.x)MN \\ & \to_\beta & (\lambda y.M)N \\ & \to_\beta & M \end{array}$$

ifThenElse FMN

$$\begin{array}{lcl} (\lambda x.x)(\lambda xy.y)MN & \to_\beta & (\lambda xy.y)MN \\ & \to_\beta & (\lambda y.y)N \\ & \to_\beta & N \end{array}$$

Exercise 3.4 [Booleans]

*a) Verify the boolean operators **And** and **Or**.*

b) Find a lamda term for the implication ($\Rightarrow$) and verify your result.

3.3.2 Natural Numbers

If f and x are lambda terms, and $n > 0$ a natural number, write $f^n x$ for the term $f(f(\ldots(fx)\ldots))$, where f occurs n times. For each natural number n, we define a lambda term $\bar{n}$, called the *nth Church numeral*, as $\bar{n} = \lambda fx.f^n x$.

Example 3.16 [Church numeral] Here are the first few Church numerals:

$$\begin{array}{lcl} \bar{0} & = & \lambda fx.x \\ \bar{1} & = & \lambda fx.fx \\ \bar{2} & = & \lambda fx.f(fx) \\ \bar{3} & = & \lambda fx.f(f(fx)) \\ \ldots & & \end{array}$$

This encoding of the natural numbers is due to Alonzo Church who was also the inventor of the lambda calculus.

First we will show, that it is possible to define the successor function $\mathbf{succ}(\bar{n}) = \overline{n+1}$. Suppose we have a Church numeral $\bar{n}$, and want to add $\bar{1}$ to it. All we need to do is apply f one more time to it. The only thing we need to remember to do is to strip the λ's from $\bar{n}$ (this is realized by the lambda term $\bar{n}fx$), and shove them back on again at the end.

$$\mathbf{succ} = \lambda nfx.f(nfx)$$

Example 3.17 [Evaluating succ $\bar{n}$]

In this example we will evaluate $\mathbf{succ}\ \bar{n}$. The result should be $\overline{n+1}$.

$$\begin{array}{rcl} \mathbf{succ}\ \bar{n} & = & (\lambda nfx.f(nfx))(\lambda fx.f^n x) \\ & \to_\beta & \lambda fx.f((\lambda f'x.f'^n x)fx) \\ & \to_\beta & \lambda fx.f((\lambda x'.f^n x')x) \\ & \to_\beta & \lambda fx.f(f^n x) \\ & \to_\beta & \lambda fx.f^{n+1}x \\ & = & \overline{n+1} \end{array}$$

Adding two numbers together is nice and easy, since we just need to replace the x in one of them by the whole of the second number. Don't forget to strip the λ's, and to shove them on again at the end. So the following term will add $\bar{n}$ and $\bar{m}$:

$$\mathbf{add} = \lambda nmfx.nf(mfx)$$

Example 3.18 [Evaluating add $\bar{n}$ $\bar{m}$]

We will show now, that $\mathbf{add}\ \bar{n}\bar{m} \twoheadrightarrow_\beta \overline{n+m}$.

$$\begin{array}{rcll} \mathbf{add}(\bar{n})(\bar{m}) & = & (\lambda nmfx.nf(mfx)) & (\lambda fx.f^n x)\ (\lambda fx.f^m x) \\ & \to_\beta & (\lambda mfx.(\lambda f'x.f'^n x)f(mfx)) & (\lambda fx.f^m x) \\ & \to_\beta & (\lambda mfx.(\lambda x.f^n x)(mfx)) & (\lambda fx.f^m x) \\ & \to_\beta & (\lambda mfx.(f^n(mfx))))) & (\lambda fx.f^m x) \\ & \to_\beta & (\lambda fx.(f^n(\lambda f'x.f'^m x)fx))))) & \\ & \to_\beta & (\lambda fx.(f^n(\lambda x'.f^m x')x)) & \\ & \to_\beta & (\lambda fx.(f^n(f^m x) & \\ & = & \overline{n+m} & \end{array}$$

Multiplication also turns out to be easy, as in order to get $\bar{n}$ copies of $\bar{m}$, just put $\bar{m}$ into each f in $\bar{n}$. The idea is simple - we just need to juggle things round a bit so that the x in each copy of $\bar{m}$ disappears (this is done by passing in $\bar{m}f$, which is a function that takes one argument, x, and so chains the new number together by itself):

$$\textbf{mult} = \lambda nmf.n(mf)$$

The predecessor function defined by **pred** $\bar{n} = \overline{n-1}$ for a positive integer n and **pred** $\bar{0} = \bar{0}$ is considerably more difficult. First we consider the Term $\mathbf{t} = (\lambda gh.h(g\ f))$. Now we can show that $\mathbf{t}^n(\lambda u.x) = (\lambda h.h(f^{n-1}(x)))$ for $n > 0$.

proof: We will proof this by induction:

n = 1:

$\mathbf{t}^1(\lambda u.x) = (\lambda gh.h(g\ f))(\lambda u.x) \rightarrow_\beta (\lambda h.h((\lambda u.x)f) \rightarrow_\beta (\lambda h.h(x)) = (\lambda h.h(f^0(x)))$

Assume it is true for n then

$$\begin{array}{rcll} \mathbf{t}^{n+1}(\lambda u.x) & = & \mathbf{t}(\mathbf{t}^n(\lambda u.x)) & \\ & = & \mathbf{t}((\lambda h.h(f^{n-1}(x))) & \text{by induction hypothesis} \\ & = & (\lambda gh.h(g\ f))((\lambda h.h(f^{n-1}(x))) & \\ & \rightarrow_\beta & (\lambda h.h(((\lambda h.h(f^{n-1}(x)))\ f)) & \\ & \rightarrow_\beta & (\lambda h.h(f(f^{n-1}(x)))) & \\ & = & (\lambda h.h(f^n(x))) & \end{array}$$

□

Now we can use the term **t** to define the function **pred**.

$$\textbf{pred} = \lambda nfx.n\ (\lambda gh.h(g\ f))(\lambda u.x)(\lambda u.u)$$

With the predecessor function, subtraction is straightforward. Defining

$$\textbf{sub} := \lambda mn.n(\textbf{pred}\ m)$$

sub $\bar{n}$ $\bar{m}$ yields $\overline{n-m}$ when $n > m$ and $\bar{0}$ otherwise.

Often handy is the function **iszero** from natural numbers to booleans, which is defined by

$$\begin{array}{rcl} \textbf{iszero}(\bar{0}) & = & true \\ \textbf{iszero}(\bar{n}) & = & false, \text{if } n \neq 0. \end{array}$$

The following term is a representation of this function:

$$\textbf{iszero} = \lambda nxy.n(\lambda z.y)x$$

Exercise 3.5 [Church number]

a) Evaluate the terms ***add*** $\bar{2}$ $\bar{3}$ *and* ***mult*** $\bar{2}$ $\bar{3}$ *manually.*

b) Prove that ***mult*** $\bar{n}\ \bar{m} = \overline{n \cdot m}$ *for all natural numbers* n, m.

c) Find lambda terms that represent each of the following functions

i) $f = (x+3)^2$

ii) $\boldsymbol{exp}(n, m) = n^m$

iii) $\boldsymbol{signum}(n) = \begin{cases} 0 & n = 0 \\ 1 & n > 0 \end{cases}$

d) Find lambda terms that represent each of the following relations

i) $m = n$

ii) $m < n$

iii) $m \leq n$

3.3.3 Fixpoint and recursive functions

We have seen how to encode some simple boolean and arithmetic functions. However, we do not yet have a systematic method of constructing such functions. What we need is a mechanism for defining more complicated functions from simple ones. Consider for example the factorial function, defined by:

$$\begin{array}{lcl} 0! & = & 1 \\ n! & = & n \cdot (n-1)! \quad \text{if } n > 0 \end{array}$$

The encoding of such functions in the lambda calculus is the subject of this section. It is related to the concept of a fixpoint. Suppose f is a function. We say that x is a fixpoint of f if $f(x) = x$. For instance, $f(x) = x^2$ has two fixpoints 0 and 1, whereas $f(x) = x + 1$ has no fixpoints. Some functions have infinitely many fixpoints, notably $f(x) = x$.

We apply the notion of fixpoints to the lambda calculus. If F and N are lambda terms, we say that N is a fixpoint of F if $FN =_\beta N$. The lambda calculus contrasts with arithmetic in that every lambda term has a fixpoint.

Theorem 3.1 [Fixpoint] *In the untyped lambda calculus, every term* F *has a fixpoint .*

> **Proof:** Let $A = \lambda xy.y(xxy)$ and define $\Theta = AA$. Now suppose F is any lambda term, and let $N = \Theta F$. We claim that N is a fixpoint of F. This is shown by the following calculation:
>
> $$\begin{array}{rcl} N & = & \Theta\ F \\ & = & A\ A\ F \\ & = & (\lambda xy.y(xxy))\ A\ F \\ & \rightarrow_\beta & (\lambda y.y\ (A\ A\ y)\ F \\ & \rightarrow_\beta & F(A\ A\ F) \\ & = & F(\Theta F) \\ & = & F\ N \end{array}$$
>
> □

The term Θ used in the proof is called *Turing's fixpoint combinator* .

The importance of fixpoints lies in the fact that they allow us to solve equations. After all, finding a fixpoint for f is the same thing as solving the equation $x = f(x)$. This covers equations with an arbitrary right-hand side, whose lefthand side is x. From the above theorem, we know that we can always solve such equations in the lambda calculus.

To see how to apply this idea, consider the question, how to define the factorial function. The most natural definition of the factorial function is recursive, and we can write it in the lambda calculus as follows:

$$\mathbf{fact}\ n = \mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n(\mathbf{fact}(\mathbf{pred}\ n)))$$

This term is not a lambda term, because **fact** on the righthandside is not defined yet. The evident problem with a recursive definition such as this one is that the term to be defined, **fact** , appears both on the left- and the right-hand side. In other words, to find **fact** requires solving an equation!

We now apply our newfound knowledge of how to solve fixpoint equations in the lambda calculus. We start by rewriting the problem slightly:

$$\begin{array}{rcl} \mathbf{fact} & = & (\lambda n.\mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n(\mathbf{fact}(\mathbf{pred}\ n)))) \\ \mathbf{fact}' & = & (\lambda f.\lambda n.\mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n\ f(\mathbf{pred}\ n)))) \end{array}$$

Now $\mathbf{fact}'$ is not recursively defined and therefore is a valid lambda term. Also note, that the fixpoint equation $\mathbf{fact} = \mathbf{fact}'(\mathbf{fact})$ with fixpoint **fact** holds. So we define **fact** with the help of the fixpoint operator.

$$\begin{array}{rcl} \mathbf{fact} & = & \Theta\mathbf{fact}' \\ & = & \Theta(\lambda f.\lambda n.\mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n\ f(\mathbf{pred}\ n)))) \end{array}$$

Note that **fact** has disappeared from the right-hand side. The right-hand side is a closed lambda term that represents the factorial function. (A lambda term is called closed if it contains no free variables).

Example 3.19 [Evaluating fact] To see how this definition works in practice, let us evaluate $\mathbf{fact}\ \bar{2}$. Recall from theorem 3.1, that $\Theta\mathbf{fac}' \twoheadrightarrow_\beta \mathbf{fac}'(\Theta\mathbf{fac}')$.

$$\begin{array}{rcl} \mathbf{fact}\ \bar{2} & = & \Theta\mathbf{fact}'\ \bar{2} \\ & \twoheadrightarrow_\beta & \mathbf{fact}'(\Theta\mathbf{fact}')\ \bar{2} \\ & = & (\lambda f n.\mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n\ f(\mathbf{pred}\ n)))(\Theta\mathbf{fact}')\ \bar{2} \\ & \rightarrow_\beta & (\lambda n.\mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n((\Theta\mathbf{fact}')(\mathbf{pred}\ n))))\bar{2} \\ & \rightarrow_\beta & (\mathbf{ifThenElse}(\mathbf{iszero}\ \bar{2})(\bar{1})(\mathbf{mult}\ \bar{2}((\Theta\mathbf{fact}'))(\mathbf{pred}\ \bar{2}))) \\ & \twoheadrightarrow_\beta & (\mathbf{mult}\ \bar{2}((\Theta\mathbf{fact}')(\bar{1}))) \\ & \twoheadrightarrow_\beta & (\mathbf{mult}\ \bar{2}(\mathbf{fact}'(\Theta\mathbf{fact}')(\bar{1}))) \\ & = & (\mathbf{mult}\ \bar{2}(\lambda f n.\mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n\ f(\mathbf{pred}\ n)))(\Theta\mathbf{fact}')\ \bar{1}) \\ & \twoheadrightarrow_\beta & (\mathbf{mult}\ \bar{2}(\mathbf{ifThenElse}(\mathbf{iszero}\ \bar{1})(\bar{1})(\mathbf{mult}\ \bar{1}((\Theta\mathbf{fact}')(\mathbf{pred}\ \bar{1}))))) \\ & \twoheadrightarrow_\beta & (\mathbf{mult}\ \bar{2}(\mathbf{mult}\ \bar{1}((\Theta\mathbf{fact}')(\bar{0})))) \\ & \twoheadrightarrow_\beta & (\mathbf{mult}\ \bar{2}(\mathbf{mult}\ \bar{1}(\mathbf{fact}'(\Theta\mathbf{fact}')(\bar{0})))) \\ & = & (\mathbf{mult}\ \bar{2}(\mathbf{mult}\ \bar{1}(\lambda f n.\mathbf{ifThenElse}(\mathbf{iszero}\ n)(\bar{1})(\mathbf{mult}\ n\ f(\mathbf{pred}\ n)))(\Theta\mathbf{fact}')(\bar{0})))) \\ & \twoheadrightarrow_\beta & (\mathbf{mult}\ \bar{2}(\mathbf{mult}\ \bar{1}(\mathbf{ifThenElse}(\mathbf{iszero}\ \bar{0})(\bar{1})(\mathbf{mult}\ \bar{0}(\Theta(\mathbf{fact}')(\mathbf{pred}\ \bar{0})))))) \\ & \rightarrow_\beta & (\mathbf{mult}\ \bar{2}(\mathbf{mult}\ \bar{1}\ \bar{1})) \\ & \twoheadrightarrow_\beta & \bar{2} \end{array}$$

The construction we used to define **fact** can also be used to construct other recursive functions. To define a lamda term that represents a recursive function we will follow this steps:

1. First we define the function recursively e.g.

$$F = T \quad \text{a term containing F}$$

2. Then we construct the non recursive function $F' = \lambda f.(T[f/F])$ (f is a fresh variable).

3. Now we apply the fixpoint operator to F' and will obtain a function with the same behavior than F.

$$F = \Theta F'$$

Exercise 3.6 [Recursion]

a) Write a lambda term that represents the Fibonacci function, defined by

$$\begin{array}{lcl} f(0) & = & 1 \\ f(1) & = & 1 \\ f(n) & = & f(n-1) + f(n-2) \text{ for } n \geq 2 \end{array}$$

b) The first fixpoint combinator for the lambda calculus was discovered by Curry. Curry's fixpoint combinator, which is also called the paradoxical fixpoint combinator, is the term $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

i) Prove that this is indeed a fixpoint combinator, i.e., that YF *is a fixpoint of* F*, for any term* F*.*

ii) Turing's fixpoint combinator not only satisfies $\Theta F =_\beta F(\Theta F)$*, but also* $\Theta F \twoheadrightarrow F(\Theta F)$*. We used this fact in evaluating* ***fact*** $\bar{2}$*. Does an analogous property hold for* Y*? Does this affect the outcome of the evaluation of* ***fact*** $\bar{2}$*?*

c) Can you find another fixpoint combinator, besides Curry's and Turing's?

3.4 Church-Rosser Theorem

3.4.1 Extensionality, η-equivalence, and η-reduction

In the untyped lambda calculus, any term can be applied to another term. Therefore, any term can be regarded as a function. Consider a term M, not containing the variable x, and consider the term $M' = \lambda x.Mx$. Then for any argument A, we have $MA =_\beta M'A$. So in this sense, M and M' define "the same function".

Should M and M' be considered equivalent as terms? The answer depends on whether we want to accept the principle that "if M and M' define the same function, then M and M' are equal". This is called the principle of *extensionality*. Formally, the extensionality rule is the following:

$$(ext) \quad Mx = M'x \wedge x \notin FV(M, M') \Rightarrow M = M'$$

Note that we can apply the extensionality rule in particular to the case where $M' = \lambda x.Mx$, where x is not free in M. As we have remarked above, $Mx =_\beta M'x$, and thus extensionality implies that $M = \lambda x.Mx$. This last equation is called the $\eta - -$law (eta-law) :

$$(\eta) \quad M = \lambda x.Mx \text{ where } x \notin FV(M)$$

We note that the η–law does not follow from the axioms and rules of the lambda calculus that we have considered so far. In particular, the terms x and $\lambda y.xy$ are not β–equivalent, although they are clearly η–equivalent. We will prove that $x \neq_\beta \lambda y.xy$ in theorem 3.7.

Definition 3.10 [Single step η–reduction] *We define single step η–reduction to be the smallest relation $\rightarrow_\eta$ on terms satisfying:*

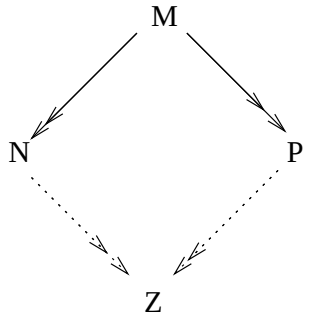
(eta)	$\lambda x.Mx \rightarrow_\eta M$	*where* $x \notin FV(M)$	
$(cong_1)$	$M \rightarrow_\eta M'$	$\Rightarrow$	$MN \rightarrow_\eta M'N$
$(cong_2)$	$N \rightarrow_\eta N'$	$\Rightarrow$	$MN \rightarrow_\eta MN'$
(ξ)	$M \rightarrow_\eta M'$	$\Rightarrow$	$\lambda x.M \rightarrow_\eta \lambda x.M'$

Single-step $\beta\eta$–reduction $\rightarrow_{\beta\eta}$ is defined as the union of the single-step β– and η–reductions, i.e., $M \rightarrow_{\beta\eta} M'$ iff $M \rightarrow_\beta M'$ or $M \rightarrow_\eta M'$. Multi-step η–reduction $\twoheadrightarrow_\eta$, multi-step $\beta\eta$–reduction $\twoheadrightarrow$, as well as η–equivalence $=_\eta$ and $\beta\eta$–equivalence $=_{\beta\eta}$ are defined in the obvious way as we did for β–reduction and equivalence. We also get the evident notions ofη–normal form and $\beta\eta$–normal form.

3.4.2 Statement of the Church-Rosser Theorem, and some consequences

Theorem 3.2 [Church and Rosser, 1936] *Let $\twoheadrightarrow$ denote either $\twoheadrightarrow_\beta$ or $\twoheadrightarrow_\eta$. Suppose M, N, and P are lambda terms such that $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$. Then there exists a lambda term Z such that $N \twoheadrightarrow Z$ and $P \twoheadrightarrow Z$.* □

In pictures, the theorem states that the following diagram can always be completed:



This property is called the *Church-Rosser property*, or *confluence* . We will not prove the Church-Rosser Theorem, but let us highlight some of its consequences.

Theorem 3.3 [Confluence 1] *If $M =_\beta N$ then there exists some Z with $M \twoheadrightarrow_\beta Z$ and $N \twoheadrightarrow_\beta Z$. Similarly for $\beta\eta$.*

> **Proof:** Suppose that $M =_\beta N$. Then there exists a $N \geq 0$ and $M_0, \ldots, M_n$ with $M = M_0$, $N = M_n$.and for all $i = 1 \ldots n$, either $M_{i-1} \rightarrow_\beta M_i$ or $M_i \rightarrow_\beta M_{i-1}$. We will proof the Theorem by induction on n.

$n = 0$:
In this case $M = N$ and there is nothing to show.

Suppose the claim has been proven for $n - 1$:
Then by induction hypothesis there exists a term Z', with $M \twoheadrightarrow_\beta Z'$ and $M_{n-1} \twoheadrightarrow_\beta Z'$. Now we have two cases:

case $N \rightarrow_\beta M_{n-1}$**:** In this case $N \twoheadrightarrow_\beta Z'$ and we are done.

case $M_{n-1} \rightarrow_\beta N$**:** In this case $Z' =_\beta N$ and we can apply the Church Rosser theorem to find Z with $Z' \twoheadrightarrow_\beta Z$ and $N \twoheadrightarrow_\beta Z$. Since $M \twoheadrightarrow_\beta Z' \twoheadrightarrow_\beta Z$ we are done

The proof in the case of $\beta\eta$–reduction is identical. □

Theorem 3.4 [Confluence 2] *If N is a β–normal form and $N =_\beta M$, then $M \twoheadrightarrow_\beta N$, and similarly for $\beta\eta$.*

Proof: By Theorem 3.3 there exists some Z with $M \twoheadrightarrow_\beta Z$ and $N \twoheadrightarrow_\beta Z$. But N is a normal form, thus $N =_\alpha Z$. □

Theorem 3.5 [Confluence 3] *If M and N are β–normal forms such that $M =_\beta N$, then $M =_\alpha N$, and similarly for $\beta\eta$.*

Proof: By Theorem 3.4 we have $M \twoheadrightarrow_\beta N$, but since M is a normal form, we have $M =_\beta N$. □

Theorem 3.6 [Confluence 4] *If $M =_\beta N$, then neither or both have a β–normal form. Similarly for $\beta\eta$.*

Proof: Suppose that $M =_\beta N$, and that one of them has a β–normal form. Say, for instance, that M has a normal form Z. Then $N =_\beta Z$, hence $N \twoheadrightarrow_\beta Z$ by theorem 3.4. □

Theorem 3.7 [Confluence 5] *The terms x and $\lambda y.xy$ are not β–equivalent. In particular, the η–rule does not follow from the β–rule.*

Proof: The terms x and $\lambda y.xy$ are both β–normal forms, and they are not α–equivalent. It follows by theorem 3.5 that $x \neq_\beta \lambda y.xy$. □

3.4.3 Reduction Strategy and Normal Form

We know from theorems 3.2 and 3.5, that if a lambda term has a β–normalform M, then M is unique up to α–equivalence. We also know, that there exist lambda terms, which have no normal form. Even if a term has a normal form, then there may be an inifite reduction sequence for the term, that never reaches a normal form. For example:

Example 3.20 [term with infinite reduction sequence]

$(\lambda yz.z)((\lambda x.xx)(\lambda w.ww))$	$\rightarrow_\beta$	$\lambda z.z$	normal form
$(\lambda yz.z)((\lambda x.xx)(\lambda w.ww))$	$\rightarrow_\beta$	$(\lambda yz.z)((\lambda w.ww)(\lambda w.ww))$	
	$\rightarrow_\beta$	$(\lambda yz.z)((\lambda w.ww)(\lambda w.ww))$	
	$\rightarrow_\beta$	$\ldots$	same term forever

Thus, theorem 3.2 guarantees that at most one normal form exists, but we do not yet have a way to find it. Intuitively, the problem with the non-terminating reduction above is that we are evaluating the argument of a function that will not be used in the function body. This suggests a strategy where we always apply the leftmost β–reduction in an term to reach a normal form.

Definition 3.11 [Reduction strategy] *The normal-order-reduction strategy (NOR for short) always reduces the leftmost outermost redex of a term.*

The aplicative-order-redution strategy (AOR for short) always reduces the leftmost innermost redex of a term first.

Example 3.21 [Reduction strategy] We will reduce the term $(\lambda x.(\lambda y.y\ y)x)((\lambda x.x)z)$ with NOR and AOR. The redex is underlined.

NOR-Strategy:

$$\begin{array}{rcl} \underline{(\lambda x.(\lambda y.y\ y)x)((\lambda x.x)z)} & \to_\beta & \underline{(\lambda y.y\ y)((\lambda x.x)z)} \\ & \to_\beta & (\underline{(\lambda x.x)z})((\lambda x.x)z) \\ & \to_\beta & z\underline{((\lambda x.x)z)} \\ & \to_\beta & z\ z \end{array}$$

AOR-Strategy:

$$\begin{array}{rcl} (\lambda x.\underline{(\lambda y.y\ y)x})((\lambda x.x)z) & \to_\beta & (\lambda x.xx)(\underline{(\lambda x.x)z}) \\ & \to_\beta & \underline{(\lambda x.xx)z} \\ & \to_\beta & z\ z \end{array}$$

Theorem 3.8 [NOR Strategy] *If a lambda term M has a normal form, then the normal-order-reduce strategy will always reduce M to it' normal form.* □

We will not proof this theorem formally. Note however, that if there exists an infinite reduction sequence for a lambda term M, then M contains at least one minimal subterm with an infinite reduction sequence. With the NOR strategy this term will disappear if M has a normal form. See 3.20 where NOR leads to the normal form but AOR leads to an infinite reduction sequence.

3.5 Lambda definable functions and recursiv functions

We have seen that the function plus, times and exponentiation on $\mathbb{N}$ can be represented in the λ–calculus using Church's numerals. We will now show that all computable (recursive) functions can be represented in the λ–calculus.

Definition 3.12 [Lambda definability]

(i) A numeric function is a map

$$f : \mathbb{N}^p \to \mathbb{N}$$

for some p. In this case f is called p-ary.

(ii) A numeric p-ary function f is called λ–definable if for some lambda term F

$$F\bar{n_1}\bar{n_2}\ldots\bar{n_p} = \overline{f(n_1, n_2, \ldots, n_p)} \quad (*)$$

for all $n_1, \ldots n_p \in \mathbb{N}$. If $(*)$ holds, then f is said to be λ–defined by F.

3.5.1 Primitive recursive functions

In section 2.1 we defined the primitive recursive functions. In this section we will show, that all primitive recursive functions are λ–definable .

Theorem 3.9 [Primitive recursive functions] *The basic primitive recursive functions are λ–definable.*

Proof: take as defining terms:

$$\begin{array}{lcll} C_i^n & \equiv & \lambda x_1 x_2 \ldots x_n.\bar{i} & \\ S(k) & \equiv & \lambda k f x.f(k\ f\ x) & \textbf{succ} \\ P_i^n & \equiv & \lambda x_1 x_2 \ldots x_n.x_i & \end{array}$$

□

Theorem 3.10 [Composition] *The λ–definable functions are closed under composition.*

Proof: Let g be a m-ary primitive recursive function λ–defined by G and let $h_1, \ldots h_m$ be n-ary primitive recursive functions λ–defined by $H_1, \ldots, H_m$ then

$$SUB(g, h_1, \ldots, h_m)(\vec{x}) =_\beta g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$$

is λ–defined by

$$F \equiv \lambda \vec{x}.G(H_1\vec{x}) \ldots (H_n\vec{x})$$

□

Theorem 3.11 [Primitive recursion] *The λ–definable functions are closed under primitive recursion.*

Proof: Let g and h be primitive recursive functions λ–defined by G and H respectivly and

$$\begin{array}{lcl} PR(g,h)(\vec{x},0) & = & g(\vec{x}) \\ PR(g,h)(\vec{x},S(y)) & = & h(\vec{x},y,PR(g,h)(\vec{x},y)) \end{array}$$

We have to find a term F such that

$$F\vec{x}\ y = \textbf{ifThenElse}(\textbf{iszero}\ y)(G\vec{x})(H\vec{x}(\ \textbf{pred}\ y)(F\vec{x}(\textbf{pred}\ y)))$$

Now use the fixpoint operator. First define

$$F' = (\lambda f \vec{x} y.\textbf{ifThenElse}(\textbf{iszero}\ y)(G\vec{x})(H\vec{x}(\textbf{pred}\ y)(f\vec{x}(\textbf{pred}\ y))))$$

Now define

$$F = \Theta F'$$

$PR(g,h)$ is λ–defined by $\Theta F'$. □

3.5.2 Recursive functions

We will show now, that recursiv functions are also λ–definable

Theorem 3.12 [Minimisation] *The λ–definable functions are closed under minimisation.*

Proof: Let g be a recursive function defined by G and

$$f(\vec{x}) = \mu(g)(\vec{x}) = min\{y : g(\vec{x}, y) = 0\}$$

We have to find a term F such that

$$F\vec{x}\ y = \textbf{ifThenElse}(\textbf{iszero}\ G\vec{x}\ y)(y)(F\vec{x}(\textbf{succ}\ \ \text{x})$$

Now use the fixpoint operator. First define F'

$$F' = (\lambda f\vec{x}y.\textbf{ifThenElse}(\textbf{iszero}\ G\vec{x}\ y)(y)(f\vec{x}(\textbf{succ}\ \ \text{x})$$

now define

$$F = \Theta F'$$

□

Theorem 3.13 [lambda definable] *Every recursiv function is λ–definable.*

Proof: By theorems 3.9, 3.10, 3.11 and 3.12. □

We can extend the concept of λ–definable to partial recursive functions.

Definition 3.13 [partial recursive function] *A partial recursive p-ary function f is called λ–definable if for some term F:*

$$F\bar{n_1}, \ldots \bar{n_p} =_\beta \left\{ \begin{array}{ll} \overline{f(\vec{n})} & \text{if } f(\vec{n}) \text{ is defined} \end{array} \right.$$

Theorem 3.14 [partial recursive function] *Every partial recursive function is λ–definable.*
□

We will not proof this theorem.

We have proofed that every partial recursiv function is λ–definable. The next theorem states that every λ–definable function is a partial recursive function.

Theorem 3.15 [Partial recursive function] *Every λ–definable function is a partially recursive.*

Proof sketch: Let f be a p-ary λ–definable function and let f be defined by P then:

$$\begin{array}{lcll} f(\vec{n}) & = & k & \text{with k such that} P\bar{n_1} \ldots \bar{n_p} = \bar{k} \text{ if P has a normal form} \\ f(\vec{n}) & = & undefined & \text{if P has no normal form} \end{array}$$

With an appropriate Gödel numbering, we can proof, that f is partially recursive Gödel Numbering.

□

Chapter 4

The 3 Computational Models

4.1 The Curch's Thesis

All the attempts to formalise mechanically computable functions give the same class of functions. This leads one to believe that we have captured the right notion of computability and this belief is usually referred to as Church's[1] thesis. Let us state it for future reference.

Church's Thesis: *The class of recursive functions is the class of mechanically computable functions, and any reasonable definition of mechanically computable will give the same class of functions.*

Observe that Church's thesis is not a mathematical theorem but a statement of experience. Thus we can use such imprecise words as *reasonable*. Church's thesis is very convenient to use when arguing about computability. Since any high level computer language describes a reasonable model of computation the class of functions computable by high level programs is included in the class of recursive functions. Thus as long as our descriptions of procedures are detailed enough so that we feel certain that we could write a high level program to do the computation, we can draw the conclusion that we can do the computation on a Turing machine or by a recursive function. In this way we do not have to worry about actually programming the Turing machine.

4.2 Turing Machines and Recursive Functions

A Turing machine defines only a partial function[2] since it is not clear that the machine will halt or not for all inputs. But whenever a Turing machine halts for all inputs, it corresponds to a total function.

Definition 4.1 [Turing Computable] *A function f is called* Turing Computable *if it is the function associated with a Turing machine halts for all inputs.*

The Turing computable functions is a reasonable definition of the mechanically computable functions and thus the first interesting question is how this new class of functions relates to

[1] Alonzo Church (1903 - 1995) was an American mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science. He is best known for the lambda calculus, Church's-Turing thesis, Frege-Church ontology, and the Church-Rosser theorem.

[2] We will see later in this section how it is possible to associate a function to a Turing machine.

the recursive functions.

Theorem 4.1 [Turing Computable] *A function is Turing computable if and only if it is recursive.*

Proof: We will only sketch the proof here. The easier part of the theorem is to prove that if a function is recursive the it is Turing computable. It is easy to write a program (e.g. a JAVA Program) that would compute a recursive function. It is harder to program Turing machines but it is still feasible since they are somehow equivalent to computers.

To prove that any Turing computable function is recursive, we associate a set of equations to a Turing machine. This can be done in the following way:

1. At each step of the computation, a TM can be described by a tuple $(i, p, X_1, X_2, \ldots, X_n)$ where i is the tape position, p is the actual symbol and $X_1, X_2, \ldots, X_n$ contains all non blank tape symbols. Since it has varying length, we will code the tuple $X_1, X_2, \ldots, X_n$ as a number z.
2. Consider the transition function $\delta(q, X) = (p, Y, D)$. Using the transition table, we can define three functions $ns(p, X) = q$ (next state), $nt(p, X) = Y$ (next tape symbol) and $nm(p, X) = D$ (next move). Since they are defined using the transition table, the three functions are primitive recursive (see exercise 2.3). Now we have $\delta(q, X) = (ns(p, X), nt(p, X), nm(p, X))$.
3. Note that using the function exponent $(z)_i$ defined in section 2.1.1, we can extract any component of the tuple z, e.g. $X_i = (z)_i$.
4. At transition $\delta(q, X)$, the TM will change from configuration (i, p, z) to a new configuration $(j, q, z') = f(i, p, z)$. The function f is again primitive recursive (it is a composition of primitive recursive functions), i.e. using the tuple notation

$$\begin{aligned} f(i, p, X_1, X_2, \ldots, X_n) &= (j, q, X_1, X_2, \ldots, X_{i-1}, Y, X_{i+1}, \ldots, X_n) \\ &= (npos(p, X_i), ns(p, X_i), X_1, X_2, \ldots, X_{i-1}, nt(p, X_i), X_{i+1}, \ldots, X_n) \end{aligned}$$

   Where

$$npos(p, X_i) = \begin{cases} i \perp 1 & \text{if} \quad nm((p, X_i) = L \\ i + 1 & \text{if} \quad nm((p, X_i) = R \end{cases}$$

   Note that f needs to be defined differently if the position corresponds to the first or the last symbol of the tape.
5. Now we can test if the TM accepts with following partial recursive function *accept*:

$$\begin{aligned} accept(0, 1, q_0, z) &= 1 \,\text{if}\, q_0 \in F \,\text{else}\, 0 \\ accept(i + 1, j, p, z) &= accept(i, f(j, p, z)) \end{aligned}$$

6. If a function g is Turing computable then the TM will eventually halt. We can compute g using

$$ACCEPT(z) = \mu n : accept(n, z)$$

and looking at the output.

If g is total (partial), then $ACCEPT$ is a total (partial) recursive function.

□

It is important to remember that, while any member of A will eventually be listed, the members of A are not necessarily listed in order and that M_A will probably never halt since A is infinite most of the time. Thus if we want to know whether $x \in A$ it is not clear how to use M_A for this purpose. We can watch the output of M and if x appears we know that $x \in A$, but if we have not seen x we do not know whether $x \notin A$ or we have not been waiting long enough.

If we would require that A was listed in order we could check whether $x \in A$ since we would only have had to wait until we had seen x or a number greater than x. Thus in this case we can conclude that A is recursive, but in general this is not true.

4.2.1 The Universal Turing Machine

Theorem 4.2 [Recursive/Recursively Enumerable] *If a set A is recursive, then it is recursively enumerable.*

Proof: We can print the elements of A using following procedure:

$i = 1$
while (true) {
    if ($i \in A$) {
        print(i) {
        $i = i + 1$ {
}

□

Codes for Turing machines

We now want to represent Turing machines with input alphabet $\{0, 1\}$ by binary strings, so that we can identify Turing machines with integers and refer to the ith Turing machine as M_i.

To represent a Turing machine

$$M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F)$$

as a binary string, we must first assign integers to the states, tape symbols, and directions L and R:

- Assume the states are $q_1, q_2, \ldots, q_r$ for some r. The start state is q_1, and the only accepting state is q_2.
- Assume the tape symbols are $X_1, X_2, \ldots, X_s$ for some s and $X_1 = 0$, $X_2 = 1$, and $X_3 = B$
- $L = D_1$ and $R = D_2$.

Encode the transition rule $\delta(q_i, X_j) = (q_k, X_l, D_m)$ by $0^i10^j10^k10^l10^m$. Note that there are no two consecutive 1s.

Encode an entire Turing machine by concatenating, in any order, the codes C_i of its transition rules, with leading, trailing and separating 11: $11C_111C_211\ldots C_{n-1}11C_n11$.

Example 4.22 [TM Code]

Let $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$ where δ is defined by: $\delta(q_1, 1) = (q_3, 0, R)$, $\delta(q_3, 0) = (q_1, 1, R)$, $\delta(q_3, 1) = (q_2, 0, R)$, and $\delta(q_3, B) = (q_3, 1, L)$.

Codes for the transitions:
0100100010100, 0001010100100, 00010010010100 and 0001000100010010

Code for M:
1101001000101001100010101001001100010010010100110001000100010001001011

Definition 4.2 [ith Turing Machine] *Let M be a Turing machine with code w_i. We associate an integer i to w_i (w_i is a binary number). The ith Turing machine is referred to as M_i.*

Many integers do not correspond to any Turing machine, e.g. 1101. If w_i is not a valid TM code, then we shall take M_i to be the Turing machine (with one state and no transitions) that immediately halts on any input. Hence $L(M_i) = \emptyset$ if w_i is not a valid TM code.

The Universal Turing machine

Theorem 4.3 [Universal Turing machine] *There is a universal Turing machine which on input (x, y, z) simulates z computational steps of M_y on input x. By this we mean that if M_y halts with output w on input x within z steps then also the universal machine outputs w. If M_y does not halt within z steps then the universal machine gives output 0.* □

We will sometimes allow z to take the value ∞. In such a case the universal machine will simulate M_y until it halts or go on for ever without halting if M_y does not halt on input x. The output will again agree with that of M_y.

In a more modern language,the universal Turing machine is more or less an interpreter since it takes as input a Turing machine program together with an input and then runs the program.

Diagonalisation

Theorem 4.4 [Recursive/Recursively Enumerable] *There are recursively enumerable sets that are not recursive.*

Proof: Consider following function V_T:

$$V_T(i) = \begin{cases} 1 & \text{if } M_i \text{ halts on input } i \text{ with output } 0 \\ 0 & \text{otherwise} \end{cases}$$

V_T is the characteristic function of a set we will denote by K_D. K_D is the set of Turing machines which halt with output 0 when given their own encoding as input. We claim that K_D is recursively enumerable but not recursive. To prove the first claim observe that K_D can be enumerated by the following procedure

$i = 1$
while (true) {

for $(j = 1, \ldots, i)$ {
If M_j is legal, run M_j , i steps on input j, if it
halts within these i steps and gives output 0 and we have not
listed j before print(j)
}
$i = i + 1$
}

Observe that this is an recursive procedure using the universal Turing machine. The only detail to check is that we can decide whether j has been listed before. The easiest way to do this is to observe that j has not been listed before precisely if $j = i$ or M_j halted in exactly i steps. The procedure lists K_D since all numbers ever printed are by definition members in K_D and if $k \in K_D$ and M_k halts in T steps on input k then k will be listed for $i = max(k, T)$ and $j = k$.

To see that K_D is not recursive,suppose that V_T can be computed by a Turing machine M. We know that $M = M_n$ for some n. Consider what happens when M is fed input n. If it halts with output 0 then $V_T(n) = 1$. On the other hand if M does not halt with output 0 then $V_T(n) = 0$. In either case M_n makes an error and hence we have reached a contradiction. □

We have proved slightly more than was required by the theorem. We have given an explicit function which cannot be computed by a Turing machine.

Theorem 4.5 [Non Turing] *The function V_T cannot be computed by a Turing machine, and hence is not recursive.*

Proof: If there is such a B then A can be enumerated by the following program:

$i = 1$
while (true) {
for $(j = 1, \ldots, i)$ {
If for some $k \leq i$ we have $(j, k) \in B$ and
$(j, l) \notin B$ for $l < k$ and j has not been
printed before then print(j)
}
$i = i + 1$
}

First observe that j has not been printed before if either j or l is equal to i. By the relation between A and B this program will list only members of A and if $j \in A$ and k is the smallest number such that $(j, k) \in B$ then j is listed for $i = max(j, k)$.

To see the converse, let M_A be the Turing machine which enumerates A. Define B to be the set of pairs (j, k) such that j is output by M_A in at most k steps. By the existence of the universal Turing machine it follows that B is recursive and by definition $\exists k : (j, k) \in B$ precisely when j appears in the output of M_A, i.e. when $j \in A$. □

The last theorem says that r.e.sets are just recursive sets plus an existential quantifier.

4.2.2 The Halting Problem

The halting problem is a decision problem about properties of computer programs on a fixed Turing-complete model of computation. The problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations of memory or time on the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

For example, in pseudo code, the program

```
while (true);
```

does not halt; rather, it goes on forever in an infinite loop. On the other hand, the program

```
print ("Hello World!");
```

halts very quickly.

The halting problem is famous because it was one of the first problems proven algorithmically undecidable. This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops when run with that input.

Definition 4.3 [Halting Set] *The halting set K is defined by*

$$K = \{(j,k) : M_k \textit{ is legal and halts on input } j\}$$

To determine whether a given pair $(j,k) \in K$ is for natural reasons called the *halting problem*. This is closely related to the diagonal halting problem which we have already proved not to be recursive in the last section. Intuitively this should imply that the halting problem also is not recursive and in fact this is the case.

Theorem 4.6 [Halting Problem] *The halting problem is not recursive.*

> Suppose K is recursive i.e. that there is a Turing machine M which on input (j,k) gives output 1 precisely when M_k is legal and halts on input j. We will use this machine to construct a machine that computes V_T using M as a subroutine. Since we have already proved that no machine can compute V_T this will prove the theorem.
>
> Now consider an input j and that we want to compute $V_T(j)$. First decide whether M_j is a legal Turing machine. If it is not we output 0 and halt. If M_j is a legal machine we feed the pair (j,j) to M. If M outputs 0 we can safely output 0 since we know that M_j does not halt on input j. On the other hand if M outputs 1 we use the universal machine on input (j,j,∞) to determine the output of M_j on input j. If the output is 0 we give the answer 1 and otherwise we answer 0. This gives a mechanical procedure that computes V_T and we have reached the desired contradiction. □

Chapter 5

FOR & WHILE Programs

We will introduce two kinds of programs, and prove some mathematical statements about them.

5.1 FOR Programs

5.1.1 Syntax

Definition 5.1 [FOR Program]
*Let $V = \{x_i : i \in \mathbb{N}^+)$ be a set of variables. A **FOR** Program is an element of the set W defined recursively as follows:*

1. **Addition:** *If $x_i \in V$, $x_j \in V$ and $x_k \in V$ then $ADD(x_i, x_j, x_k) \in W$.*
2. **Subtraction:** *If $x_i \in V$, $x_j \in V$ and $x_k \in V$ then $SUB(x_i, x_j, x_k) \in W$).*
3. **Constant:** *If $x_i \in V$ and $c \in \mathbb{N}$ then $CONST(x_i, c) \in W$.*
4. **Sequence:** *If $P_1 \in W$ and $P_2 \in W$ then $SEQ(P_1, P_2) \in W$.*
5. **For:** *Let $P_1 \in W$ and $x_i \in V$ then $FOR(x_i, P_1) \in W$.*

You can think of W to be the set of all programs defined with following context free grammar:

```
             <ID> ::=   x[1-9][0-9]*

            <NUM> ::=   [0-9]+

      forProgram ::=   simpleStatement
                     | forStatement
                     | sequence

simpleStatement ::=   <ID> "=" <ID> "+" <ID>
                     | <ID> "=" <ID> "-" <ID>
                     | <NUM>
```

```
forStatement ::=   "for" <ID> "do" forProgram "od"

    sequence ::=   forProgram  ";"  forProgram
```

Remark 5.1 [FOR Program] Formally, the set W is defined as

$$W = \bigcup_{i \geq 1} W_i$$

where

$$\begin{aligned}
W_1 \;=\; & \{ \; P : \\
& \quad \exists x_i, x_j, x_k \in X : P = ADD(x_i, x_j, x_k) \vee \\
& \quad \exists x_i, x_j, x_k \in X : P = SUB(x_i, x_j, x_k) \vee \\
& \quad \exists x_i \in X, c \in \mathbb{N} : P = CONST(x_i, c) \\
& \} \\
W_n \;=\; W_{n-1} \; & \cup \\
& \{ \; P : \\
& \quad \exists P_1 \in W_{n-1}, x_i \in X : P = FOR(x_i, P_1) \vee \\
& \quad \exists P_1 \in W_j, \exists P_2 \in W_k, j + k \leq n - 1 : P = SEQ(P_1, P_2) \\
& \}
\end{aligned}$$

5.1.2 Semantic

A program P gets a number of inputs $\alpha_1, \ldots, \alpha_n \in \mathbb{N}$. The input is stored in the variables $x_1, \ldots, x_n$.

Definition 5.2 [Output] *Let P be a program using variables $x_1, \ldots, x_n$. The* output *is the content of x_1 after the execution of the program.*

The set $V = \{x_i : i \in \mathbb{N}^+)$ of possible variables is infinite, but each FOR program P always uses a finite number of variables. We may always assume that the set of used variables is $X = \{x_1, \ldots, x_n\}$.

Definition 5.3 [State] *A* state *is a vector $S \in \mathbb{N}^n$. It describes the content of the variables in P.*

If $\sigma_1, \ldots, \sigma_n$ is our input, then the initial state S_0 will be $(\sigma_1, \ldots, \sigma_n)$ (initial values that are not used have value 0).

Given a state S and a program $P \in W_n$, we will now describe what happens when running P starting in S.

We will use the function $\Phi_P : \mathbb{N}^n \to \mathbb{N}^n$ to describe the states of a program. $\Phi_P(S) = (\alpha_1, \ldots, \alpha_n)$ denote the state that is reached after running P on state $S = (\sigma_1, \ldots, \sigma_n)$.

1. Addition:

$$\Phi_{ADD(x_i,x_j,x_k)}(S) = (\sigma_1, \ldots, \sigma_{i-1}, \sigma_j + \sigma_k, \sigma_{i+1}, \ldots, \sigma_n)$$

2. Subtraction:

$$\Phi_{SUB(x_i,x_j,x_k)}(S) = (\sigma_1, \ldots, \sigma_{i-1}, max\{\sigma_j - \sigma_k, 0\}, \sigma_{i+1}, \ldots, \sigma_n)$$

3. **Constant:**

$$\Phi_{CONST(x_i,c)}(S) = (\sigma_1, \ldots, \sigma_{i-1}, c, \sigma_{i+1}, \ldots, \sigma_n)$$

4. **Sequence**

$$\Phi_{SEQ(P_1,P_2)}(S) = \Phi_{P_2}(\Phi_{P_1}(S))$$

5. **For:**

$$\Phi_{FOR(x_i,P_1)}(S) = \Phi_{P_1}^{(\sigma_i)}(S)$$

The semantics of $ADD(x_i, x_j, x_k)$ corresponds to $x_i = x_j + x_k$. This operation takes the values of x_j and x_k, adds them and stores the result in x_i. $SUB(x_i, x_j, x_k)$ corresponds to $x_i = x_j - x_k$, if the result is negative, we will set it to 0 instead, since we can only store values from $\mathbb{N}$. This operation is also called modified difference. $CONST(x_i, c)$ corresponds to $x_i = c$, the assignment of the constant value c to the variable x_i.

A for loop $FOR(x_i, P_1)$ executes P_1 σ_i times, where σ_i is the value of x_i before the execution of the for loop. This means that changing the value of x_i during the execution of the for loop does not have any effect. In particular, for loops always terminate.

$SEQ(P_1, P_2)$ is the concatenation of P_1 and P_2, we first execute P_1 on S and then P_2 on the state produced by P_1 provided that P_1 is defined.

5.2 WHILE Programs

5.2.1 Syntax

Definition 5.4 [WHILE Program]

*Let $V = \{x_i : (i \in \mathbb{N}$ be a set of variables. A **WHILE** Program is an element of the set W defined recursively as follows:*

1. **Addition:** *If $x_i \in V$, $x_j \in V$ and $x_k \in V$ then $ADD(x_i, x_j, x_k) \in W$.*
2. **Subtraction:** *If $x_i \in V$, $x_j \in V$ and $x_k \in V$ then $SUB(x_i, x_j, x_k \in W)$.*
3. **Constant:** *If $x_i \in V$ and $c \in \mathbb{N}$ then $CONST(x_i, c) \in W$.*
4. **Sequence:** *If $P_1 \in W$ and $P_2 \in W$ then $SEQ(P_1, P_2) \in W$.*
5. **While:** *Let $P_1 \in W$ and $x_i \in V$ then $WHILE(x_i, P_1) \in W$.*

You can think of W to be the set of all programs defined with following context free grammar:

```
            <ID> ::=   x[1-9][0-9]*

           <NUM> ::=   [0-9]+

   whileProgram ::=   simpleStatement
                    | whileStatement
                    | sequence

simpleStatement ::=   <ID> "=" <ID> "+" <ID>
```

```
                    | <ID> "=" <ID> "-" <ID>
                    | <NUM>

whileStatement ::=   "while" <ID> "do" whileProgram "od"

      sequence ::=   whileProgram  ";"  whileProgram
```

5.2.2 Semantic

We will use here the notation of section 5.1.2. The semantic rules for `WHILE` programs are the same as the ones for `FOR` programs except rules 4 and 5.

4. Sequence:

$$\Phi_{SEQ(P_1,P_2)}(S) = \begin{cases} \Phi_{P_2}(\Phi_{P_1}(S)) & \text{if } \Phi_{P_1}(S) \text{ and } \Phi_{P_2}(\Phi_{P_1}(S)) \text{ are defined} \\ undef & \text{otherwise} \end{cases}$$

5. While: Let $P = WHILE(x_i, P_1)$ and let r be the smallest $r \in \mathbb{N}$ such that $\Phi_{P_1}^{(r)}(S)$[1] is either undefined, which means that P_1 does not terminate, or the ith position in $\Phi_P^{(r)}(S)$ equals 0.

$$\Phi_{WHILE(x_i,P_1)}(S) = \begin{cases} \Phi_{P_1}^{(r)}(S) & \text{if } r \text{ exists and } \Phi_{P_1}^{(r)}(S) \text{ is defined} \\ undef & \text{otherwise} \end{cases}$$

The semantics of a while loop $WHILE(x_i, P_1)$ is again as expected: We execute P_1 as long as the value of x_i does not equal 0. If the loop does not terminate, the result is undefined.

Exercise 5.1 [Simulation] *Show that for every `FOR` program P there is an equivalent `WHILE` program Q, i.e. $\Phi_P = \Phi_Q$.*

5.3 Computable Functions

Definition 5.5 [`FOR` Computed Function] *Let P be a `FOR` program with n variables. The function $\varphi_P : \mathbb{N}^n \to \mathbb{N}$ computed by P is defined by*

$$\varphi_P(x_1, \ldots, x_n) = U_1^l(\Phi_P(x_1, \ldots, x_n)$$

For all $(x_1, \ldots, x_n) \in \mathbb{N}^N$.

Definition 5.6 [`WHILE` Computed Function] *Let P be a `WHILE` program with n variables. The function $\varphi_P : \mathbb{N}^n \to \mathbb{N}$ computed by P is defined by*

$$\varphi_P(x_1, \ldots, x_n) = \begin{cases} U_1^l(\Phi_P(x_1, \ldots, x_n) & \textit{if } \Phi_P(x_1, \ldots, x_n) \textit{ is defined} \\ undef & \textit{otherwise} \end{cases}$$

[1] $\Phi_P^{(r)}$ denotes here the rth iteration of the function Φ_P, i.e.

$$\Phi_P^{(r)}(x_1, \ldots, x_n) = \begin{cases} (x_1, \ldots, x_n) & \text{if } r = 0 \\ \Phi_P(\Phi_P^{(r)}(x_1, \ldots, x_n)) & \text{otherwise} \end{cases}$$

For all $(x_1, \ldots, x_n) \in \mathbb{N}^n$.

Definition 5.7 [`FOR` Computable Function] *A function* $f : \mathbb{N}^n \to \mathbb{N}$ `FOR` computable *if there is a* `FOR` *program* P *such that* $f = \varphi_P$.

The set of all `FOR` *computable functions is denoted by* $F(FOR)$.

Definition 5.8 [`WHILE` Computable Function] *A (partial) function* $f : \mathbb{N}^n \to \mathbb{N}$ `WHILE` computable *if there is a* `WHILE` *program* P *such that* $f = \varphi_P$.

The set of all `WHILE` *computable functions is denoted by* $F(WHILE)$.

By exercise 5.1, we already know that $FOR \subseteq WHILE$. We will show that $F(PRIM) = F(FOR)$ and $F(REC) = F(WHILE)$ but we need to introduce functions into our programing languages.

Remark 5.2 [Computable Function] According to definitions 5.5 and 5.6, we can associate n functions $p_1, \ldots, p_n$ with each `FOR` or `WHILE` program P with n variables such that $p_i(\sigma_1, \ldots, \sigma_n) = \alpha_i$ where $S_b = (\sigma_1, \ldots, \sigma_n)$ is the state of the system before execution of P and $S_a = (\alpha_1, \ldots, \alpha_n)$ is the state of the system after execution of P.

5.3.1 Functions & Assignment

We will use the notation $SET(x_i, x_j)$ for the sequence $SEQ(CONST(x_k, 0), ADD(x_i, x_j, x_k))$.

Let $h : \mathbb{N}^t \to \ltimes$ be a `WHILE` or `FOR` computable function, respectively, with m variables and let P be a `WHILE` or `FOR` program for h. We can enrich any `WHILE` or `FOR` program with n variables by statements of the form $CALL(x_i, h(x_{j_1}, \ldots, x_{j_m}))$ and can always revert back to ordinary `WHILE` or `FOR` as follows:

$$
\begin{array}{l}
SET(x_{n+1}, x_{j_1}); \\
\ldots \\
SET(x_{n+m}, x_{j_m}); \\
\hat{P}; \\
SET(x_i, x_{m+1});
\end{array}
$$

$\hat{P}$ is the program obtained from P by replacing every variable x_i by x_{i+m}. Basically, we replace every occurrence of h by a program that computes h and avoid any interferences by choosing new variables.

5.3.2 `FOR` Computable Functions

Theorem 5.1 [`FOR` Computable Functions] *A function* f *is* `FOR` *computable if and only if* f *is primitive recursive, i.e.*

$$F(FOR) = F(PRIM)$$

Proof: We first show that every primitive recursive function is `FOR` computable.

1. Constant function: We compute $C_i^n(\alpha_1, \ldots \alpha_n)$ as follows

$$
\begin{array}{l}
CONST(x_1, \alpha_1); \\
\ldots \\
CONST(x_n, \alpha_n); \\
CONST(x_1, i);
\end{array}
$$

2. Successor function: We compute $S(\alpha)$ as follows

$$Set(x_1, \alpha);$$

3. Projection function: We compute $P_i^n(\alpha_1, \ldots \alpha_n)$ as follows

$$
\begin{array}{l}
CONST(x_1, \alpha_1); \\
\ldots \\
CONST(x_n, \alpha_n); \\
SET(x_1, x_i);
\end{array}
$$

4. Composition: We compute $SUB(g, h_1, \ldots, h_m)(\alpha_1, \ldots \alpha_n)$ as follows

$$
\begin{array}{l}
CALL(x_1, h_1(\alpha_1, \ldots \alpha_n); \\
\ldots \\
CALL(x_m, h_m(\alpha_1, \ldots \alpha_n); \\
CALL(x_1, g(x_1, \ldots x_m)
\end{array}
$$

5. Primitive recursion: We compute $PR(g, h)(\alpha_1, \ldots \alpha_n, \alpha_{n+1})$ as follows (see also remark 2.6):

$$
\begin{array}{l}
CALL(x_{n+2}, g(\alpha_1, \ldots \alpha_n); \\
CONST(x_{n+1}, \alpha_{n+1}); \\
CONST(x_{n+3}, 1); \\
CONST(x_{n+4}, 1); \\
FOR(x_{n+1}, P);
\end{array}
$$

where P

$$
\begin{array}{l}
CALL(x_{n+2}, h(\alpha_1, \ldots \alpha_n, x_{n+3}, x_{n+2}))); \\
ADD(x_{n+3}, x_{n+3}, x_{n+4});
\end{array}
$$

Now we show that `FOR` computable functions are primitive recursive. Note that we may need to show that each element of a state S can be computed with a primitive recursive function.

1. Addition: Let $P = ADD(x_1, x_2, x_3)$ recall that

$$
\begin{array}{rcl}
\Phi_{ADD(x_1,x_2,x_3)}(\sigma_1, \sigma_2, \sigma_3) & = & (\sigma_1 + \sigma_2, \sigma_2, \sigma_3) \\
& = & (add(\sigma_1, \sigma_2), \sigma_2, \sigma_3)
\end{array}
$$

thus

$$
\begin{array}{rcl}
\varphi_{ADD(x_1,x_2,x_3)}(\sigma_1, \sigma_2, \sigma_3) & = & U_1^3(add(\sigma_1, \sigma_2), \sigma_2, \sigma_3) \\
& = & add(\sigma_1, \sigma_2)
\end{array}
$$

See also definition 2.2.

2. **Subtraction:** Let $P = SUB(x_1, x_2, x_3)$ recall that

$$\begin{aligned}\Phi_{SUB(x_1,x_2,x_3)}(\sigma_1, \sigma_2, \sigma_3) &= (\sigma_1 - \sigma_2, \sigma_2, \sigma_3) \\ &= (sub(\sigma_1, \sigma_2), \sigma_2, \sigma_3)\end{aligned}$$

thus

$$\begin{aligned}\varphi_{SUB(x_1,x_2,x_3)}(\sigma_1, \sigma_2, \sigma_3) &= U_1^3(sub(\sigma_1, \sigma_2), \sigma_2, \sigma_3) \\ &= sub(\sigma_1, \sigma_2)\end{aligned}$$

See also definition 2.3.

3. **Constant:** Let $P = CONST(x_1, c)$ recall that

$$\begin{aligned}\Phi_{CONST(x_1,c)}(\sigma_1) &= (c) \\ &= (C_c^1(\sigma_1))\end{aligned}$$

thus

$$\begin{aligned}\varphi_{CONST(x_1,c)}(\sigma_1) &= U_1^1(C_c^1(\sigma_1)) \\ &= C_c^1(\sigma_1)\end{aligned}$$

4. **Sequence:** Let $P = SEQ(P_1, P_2)$, we may assume (induction) that there is a primitive recursive function g that computes φ_{P_2} and there are n primitive recursive functions $p_{1_1}, \ldots, p_{1_n}$ such that $p_i(x_1, \ldots, x_n) = \alpha_i$ where $S = (\alpha_1, \ldots, \alpha_n)$ is the state of the system after execution of P_1. Now

$$\varphi_{SEQ(P_1,P_2)} = SUB(g, p_{1_1}, \ldots, p_{1_n})$$

5. **While:** Recall that

$$FOR(x_i, P) = P^{(\sigma_i)}$$

i.e.

$$FOR(x_i, P) = SEQ(P, FOR(x_i - 1, P))$$

If we want to compute $FOR(x, P)$ we may assume (induction) that there are primitive recursive functions p_i that compute the components of $\Phi(P)$. Note that $FOR(m, P) = SEQ(SEQ(\cdots, P), P)$ $(m - 1$ times$)$.
We define g_i and h_i $(1 \leq i \leq n)$ as

$$g_i(\vec{x}) = P_i^n(\vec{x})$$

and

$$h_1(\vec{x}, m, z_1, \ldots, z_n) = P_i^n(p_1(z_1, \ldots, z_n), \ldots, p_n(z_1, \ldots, z_n))$$

Claim that

$$(f_1, \ldots, f_k) = SPR(g_1, \ldots, g_k, h_1, \ldots, h_k)$$

computes

$$\Phi_{FOR(x_i,P)}$$

More precisely,

$$\Phi_{FOR(i,P)}(\vec{x}) = (f_1(\vec{x}, i), \ldots, f_n(\vec{x}, i))$$

□

Exercise 5.2 [`WHILE` Computable Functions] *Prove that a (partial) function f is `WHILE` computable if and only if f is recursive, i.e.*

$$F(WHILE) = F(REC)$$

Chapter 6

Mathematical background

6.1 Axiom

The word *axiom* comes from the Greek word $\alpha\xi\iota\omega\mu\alpha$ (axioma), a verbal noun from the verb $\alpha\xi\iota o\epsilon\iota\nu$ (axioein), meaning *to deem worthy*, but also *to require*, which in turn comes from $\alpha\xi\iota o\zeta$ (axios), meaning *being in balance*, and hence *having (the same) value (as)*, *worthy*, *proper*. Among the ancient Greek philosophers an axiom was a claim which could be seen to be true without any need for proof.

In traditional logic, an *axiom* or *postulate* is a proposition that is not proved or demonstrated but considered to be either self-evident, or subject to necessary decision. Therefore, its truth is taken for granted, and serves as a starting point for deducing and inferring other (theory dependent) truths.

Definition 6.1 [Axiom] *In mathematics, the term* axiom *is used in two related but distinguishable senses:* logical axioms *and* non-logical axioms*. In both senses, an axiom is any mathematical statement that serves as a starting point from which other statements are logically derived. Unlike* theorems*, axioms (unless redundant) cannot be derived by principles of deduction, nor are they demonstrable by mathematical proofs, simply because they are starting points; there is nothing else from which they logically follow (otherwise they would be classified as theorems).*

Logical axioms are usually statements that are taken to be universally true (e.g., $A \wedge B \longrightarrow A$), while non-logical axioms (e.g., $a + b = b + a$) are actually defining properties for the domain of a specific mathematical theory (such as arithmetic). When used in that sense, axiom, postulate, and "assumption" may be used interchangeably. In general, a non-logical axiom is not a self-evident truth, but rather a formal logical expression used in deduction to build a mathematical theory. To axiomatise a system of knowledge is to show that its claims can be derived from a small, well-understood set of sentences (the axioms).

6.2 The natural Numbers

6.2.1 The Peano axioms

The Peano[1] axioms define the properties of natural numbers, usually represented as a set $\mathbb{N}$.

Axiom 6.1 [Peano Axioms] We begin with a set $\mathbb{N}$ and a mapping S of $\mathbb{N}$ into $\mathbb{N}$. The

[1]Giuseppe Peano (27 August 1858 - 20 April 1932) was an Italian mathematician, whose work was of exceptional philosophical value.He was a founder of mathematical logic and set theory, to which he contributed

elements of $\mathbb{N}$ we call *natural numbers*. For $n \in \mathbb{N}$ we call $S(n)$ the *successor* of n.

1. 0 is a natural number.
2. For every natural number n, $S(n)$ is a natural number.
3. For every natural number n, $S(n) \neq 0$. There is no natural number whose successor is 0.
4. For all natural numbers m and n, if $S(m) = S(n)$, then $m = n$. That is, S is one-to-one (injection).
5. If ϕ is a unary predicate such that:
   - $\phi(0)$ is true, and
   - for every natural number n, if $\phi(n)$ is true, then $\phi(S(n))$ is true,

   then $\phi(n)$ is true for every natural number n.

The first two axioms define the properties of the natural numbers. The constant 0 is assumed to be a natural number, and the naturals are assumed to be closed under a *successor* function S. We denote also $S(n)$ by $n+$.

The two next axioms together imply that the set of natural numbers is infinite, because it contains at least the infinite subset $\{0, S(0), S(S(0)), \ldots\}$, each element of which differs from the rest.

The final axiom, sometimes called the *axiom of induction*, is a method of reasoning about all natural numbers; it is the only second-order[2] axiom.

6.2.2 Mathematical induction

Mathematical induction is a method of mathematical proof typically used to establish that a given statement is true of all natural numbers. It is done by proving that the first statement in the infinite sequence of statements is true, and then proving that if any one statement in the infinite sequence of statements is true, then so is the next one.

Mathematical induction is a direct consequence of the fifth Peano axiom.

The simplest and most common form of mathematical induction proves that a statement involving a natural number n holds for all values of n. The proof consists of two steps:

Induction base showing that the statement holds when $n = 0$.

Inductive step showing that if the statement holds for some n, then the statement also holds when $n + 1$ is substituted for n

much notation. The standard axiomatisation of the natural numbers is named in his honor. As part of this axiomatisation effort, he made key contributions to the modern rigorous and systematic treatment of the method of mathematical induction. He spent most of his career teaching mathematics at the University of Turin.

[2] *First-order logic* uses only discrete variables (eg. the variable x represents a person) whereas *second-order logic* uses variables that range over sets of individuals. For example, the second-order sentence $\forall P \forall x (x \in P \vee x \notin P)$ says that for every set P of people and every person x, either x is in P or it is not (this is the principle of bivalence). Second-order logic also includes variables quantifying over functions, and other variables. Both first-order and second-order logic use the idea of a domain of discourse (often called simply the *domain* or the *universe*). The domain is a set of individual elements which can be quantified over.

The assumption in the inductive step that the statement holds for some n is called the induction hypothesis (or inductive hypothesis). To perform the inductive step, one assumes the induction hypothesis and then uses this assumption to prove the statement for $n + 1$.

The description above of the basis applies when 0 is considered a natural number, as is common in the fields of combinatorics and mathematical logic. If, on the other hand, 1 is taken to be the first natural number, then the base case is given by $n = 1$.

This method works by first proving that the statement is true for a starting value, and then proving that the process used to go from one value to the next is valid. If these are both proved, then any value can be obtained by performing the process repeatedly. It may be helpful to think of the domino effect; if one is presented with a long row of dominoes standing on end, one can be sure that:

1. The first domino will fall
2. Whenever a domino falls, its next neighbor will also fall,

so it is concluded that all of the dominoes will fall, and that this fact is inevitable.

6.2.3 Addition

Definition 6.2 [Addition] *The* addition *of natural numbers is the mapping A of $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$ such that*

1. *$A(m, 0) = m$ for all $m \in \mathbb{N}$*
2. *$A(m, S(n)) = S(A(m, n)$ for all $m, n \in \mathbb{N}$*

Notation 6.1 [Addition] We write $m + n$ for $A(m, n)$

Remark 6.1 [Uniqueness] We can prove directly from the axioms that the addition of natural numbers as defined above is unique.

We show now that some properties of the addition can be prover directly from the axioms (mainly using mathematical induction).

Theorem 6.1 [Properties of addition] *Let m, n and p be natural numbers, then*

1. *$m + 0 = 0 + m = m$ for all $m \in \mathbb{N}$. i.e. 0 is the neutral element for the addition*
2. *$(m + n) + p = m + (n + p)$ for all $m, n, p \in \mathbb{N}$, i.e. the addition is associative.*
3. *$m + n = n + m$ for all $m, n \in \mathbb{N}$, i.e. the addition is commutative.*

Proof:

1. Since $m + 0 = m$ by definition, we only need to show that $m = 0 + m$ for all $m \in \mathbb{N}$. We use hereby induction over m

   Basis: $0 + 0 = 0$ (by definition)

   Induction: Suppose that $m = 0+m$. By definition $0+S(m) = S(0+m)$. By hypothesis $S(0 + m) = S(m)$ $\square$

2. Proof (by induction over p):

   Basis: $(m+n)+0 = m+n = m+(n+0)$

   Induction: Suppose that $(m+n)+p = m+(n+p)$. By definition of the addition, $(m+n)+S(p) = S((m+n)+p)$. By hypothesis $S((m+n)+p) = S(m+(n+p))$. By definition of the addition, $S(m+(n+p)) = m+S(n+p) = m+(n+S(p))$ $\square$

3. We first prove that $S(m)+n = m+S(n)$ for all $m, n \in \mathbb{N}$. We use an induction proof over n.

   Basis: $S(m)+0 = S(m) = S(m+0) = m+S(0)$ (neutral element).

   Induction: Suppose that $S(m)+n = m+S(n)$.By definition, $S(m)+S(n) = S(S(m)+n)$. By hypothesis, $S(S(m)+n) = S(m+S(n))$. By definition of the addition $S(m+S(n)) = m+S(S(n))$

   Now we can show that $m+n = n+m$ using induction over m.

   Basis: $0+n = n+0$ since 0 is neutral element.

   Induction: Suppose that $m+n = n+m$. By definition $m+S(n) = S(m+n)$. By hypothesis $S(m+n) = S(n+m)$. By definition $S(n+m) = n+S(m)$. Now it follows from the result above that $n+S(m) = S(n)+m$. $\square$

6.2.4 Multiplication

Definition 6.3 [Multiplication] *The* multiplication *of natural numbers is the mapping M of $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$ such that*

1. *$M(m,0) = 0$ for all $m \in \mathbb{N}$*
2. *$M(m,S(n)) = M(m,n)+m$ for all $m, n \in \mathbb{N}$*

Notation 6.2 [Multiplication] We write $m*n$ for $M(m,n)$ and 1 for $S(0)$

Remark 6.2 [Uniqueness] We can prove directly from the axioms that the multiplication of natural numbers as defined above is unique.

The properties of the multiplication can be proving directly from the axioms (mainly using mathematical induction).

Theorem 6.2 [Properties of multiplication] *Let m, n and p be natural numbers, then*

1. *$m*1 = 1*m = m$ for all $m \in \mathbb{N}$. i.e. 0 is the neutral element for the addition*
2. *$(m*n)*p = m*(n*p)$ for all $m, n, p \in \mathbb{N}$, i.e. the addition is associative.*
3. *$m*n = n*m$ for all $m, n \in \mathbb{N}$, i.e. the addition is commutative.*

Proof: Exercise $\square$

Exercise 6.1 [Distributivity] *Prove that*

$$m*(n+p) = m*n+m*p \quad \forall m, n, p \in \mathbb{N}$$

Hint: use induction over p.

Exercise 6.2 [Multiplication] *Prove theorem 6.2.*

6.2.5 Order

If one number is less than the other, then the second can be obtained by adding a number to the first. For the set $\mathbb{N}$ of natural numbers we have

Theorem 6.3 [Order in $\mathbb{N}$] *If T is the subset of $\mathbb{N} \times \mathbb{N}$ consisting of all (m, n) such that $m + q = n$ for some $q \in \mathbb{N}$, then T is an order relation in $\mathbb{N}$, i.e.*

1. *$(m, m) \in T$ for all $m \in \mathbb{N}$ (reflexivity)*
2. *If $(m, n) \in T$ and $(n, m) \in T$ then $m = n$ (symmetry)*
3. *If $(m, n) \in T$ and $(n, p) \in T$ then $(m, p) \in T$*

Notation 6.3 [Order in $\mathbb{N}$] We write $m \leq n$ if $(m, n) \in T$.

Proof:

1. $m + 0 = m$ for all $m \in \mathbb{N}$, i.e. $(m, m) \in T$ for all $m \in \mathbb{N}$
2. We first prove that $m + q = m$ if and only if $q = 0$.

   Basis: Let $m = 0$. **(if):** if $q = 0$, then $0 + 0 = 0$. **(only if):** $0 + q = q = 0$.

   Induction: Suppose that $m + q = m$ if and only if $q = 0$. We need to prove that $S(m) + q = S(m)$ if and only if $q = 0$. **(if)**: if $q = 0$ then $S(m) + 0 = S(m)$. **(only if):** if $S(m) + q = S(m)$ then $S(m) + q = S(m + q) = S(m)$, i.e. $m + qm$ the proof follows by induction hypothesis. □

   Now suppose that if $(m, n) \in T$ and $(n, m) \in T$, i.e. $m+p = n$ for some p and $n+q = m$ for some q. It follows that $(m + p) + q = n + q = m$, i.e. $m + (p + q) = m$. If follows that $p + q = 0$, i.e. $p = q = 0$. Finally, since $p = q = 0$, $m = n$ □
3. If $(m, n) \in T$ and $(n, p) \in T$, then $m + q = n$ for some q and $n + r = p$ for some r, i.e. $m + (q + r) = n + r = p$. It follows that $(m, p) \in T$ □

Definition 6.4 [Initial segment] *For $n \in \mathbb{N}$, the* initial segment *I_n is the set of all $m \in \mathbb{N}$ such that $m \leq n$.*

6.3 Counting

The familiar process of counting uses $\mathbb{N} - \{0\}$ as a standard set of tags. If the elements of a set can be tagged with the whole numbers form 1 to n, the set is said to have n elements. The usefulness of this process lies in the fact that a set which can be tagged with the whole numbers from 1 to n cannot also be tagged with the whole numbers from 1 to m unless $m = n$.

Theorem 6.4 [Counting] *There is no 1-1 mapping of any initial segment I_n onto a proper subset of I_n.*

Proof: Let M be the set of all $n \in \mathbb{N}$ such that there is no 1-1 mapping of the initial segment I_n onto any of its proper subsets. We prove by induction than $M = \mathbb{N}$.

Basis: $1 \in M$, since the only proper subset of $I_1 = \{1\}$ is the empty set, and the empty set is not the range of any mapping.

Induction: Suppose that $n \in M$, and let F be a 1-1 mapping of $I_{S(n)}$ onto some proper subset K of $I_{S(n)}$. Then exactly one of the following is true:

(1) $S(n) \in K$ and $F(S(n)) = S(n)$
(1) $S(n) \in K$ and $F(S(n)) \neq S(n)$
(2) $S(n) \notin K$

If (1) is true, let

$$F' = \{(m, F(m)) : m \in I_n\}$$

Then F' is a 1-1 mapping of I_n onto $K - \{S(n)\}$, which is a proper subset of I_n, since k is a proper subset of $I_{S(n)}$. This is impossible. since $n \in M$.

If (2) is true, the $F(S(n)) = t \neq S(n)$ and $F(k) = S(n)$ for some $k \neq S(n)$. Then set

$$F'' = \{(m, F(m)) : m \neq k, S(n)\} \cup \{(k, t), (S(n), S(n)\}$$

is a 1-1 mapping of $I_{S(n)}$ onto K for which (1) is true. This has been shown to be impossible.

If (3) is true, then $K \subset I_n$ ant the set

$$F''' = F - \{(S(n), F(S(n)))\}$$

is a 1-1 mapping of I_n onto $K - \{F(S(n))\}$. But $K - \{F(S(n))\}$ is a proper subset of I_n, since $F(S(n)) \in K$, by (3), $K \subset I_n$. This is impossible since $n \in M$ □

Theorem 6.5 [Counting] *If a set X is not finite, then there is a 1-1 mapping of $\mathbb{N}$ onto some subset of X.*

Proof: This theorem cannot be prove using the axioms stated so far. A proper proof requires usage of the *Axiom of Choice.* □

Definition 6.5 [Infinite Set] *A set X is* infinite *if there is a 1-1 mapping of X onto a proper subset of itself.*

Definition 6.6 [Denumerable Set] *A set X is called* denumerable *if there is a 1-1 mapping of $\mathbb{N}$ onto X.*

Theorem 6.6 [$\mathbb{Z}$ is denumerable] *$\mathbb{Z}$ is denumerable.*

Proof: Let

$$F(n) = \begin{cases} \frac{n}{2} & \text{if} \quad n \quad \text{is even} \\ -\frac{n+1}{2} & \text{if} \quad n \quad \text{is odd} \end{cases}$$

Then F is a 1-1 mapping of $\mathbb{N}$ onto $\mathbb{Z}$ □

Theorem 6.7 [$\mathbb{N} \times \mathbb{N}$ is denumerable] *$\mathbb{N} \times \mathbb{N}$ is denumerable.*

Proof: For each $(p, q) \in \mathbb{N} \times \mathbb{N}$ there is exactly one $n \in \mathbb{N}$ such that $p + q = n$ Let

$$R_n = \{(p, q) : p + q = n\}$$

then R_n contains $n + 1$ elements, i.e. $(0, n), (1, n - 1), (2, n - 2), \ldots (n - 1, 1), (n, 0)$. Let $p + q = n$ we define:

$$G(p, q) = p + \sum_{i=0}^{n} |R_i| = p + \sum_{i=0}^{n} (i + 1) = p + \frac{(n + 1)(n + 2)}{2}$$

Then $F = G^{-1}$ is a 1-1 mapping of $\mathbb{N}$ onto $\mathbb{N} \times \mathbb{N}$ $\square$

Theorem 6.8 [$\mathbb{Z} \times \mathbb{Z}$ is denumerable] *$\mathbb{Z} \times \mathbb{Z}$ is denumerable.* $\square$

Theorem 6.9 [$\mathbb{Q}$ is denumerable] *$\mathbb{Q}$ is denumerable.* $\square$

Proof: Follows directly from theorem 6.7 and theorem 6.8 $\square$

Theorem 6.10 [$P(\mathbb{N})$ is not denumerable] *$P(\mathbb{N})$ the power set of $\mathbb{N}$ is not denumerable.*

Proof: Suppose there is a mapping F of $\mathbb{N}$ onto $P(\mathbb{N})$. Since the set $A = \{n : n \notin F(n)\} \in P(\mathbb{N})$ and F maps $\mathbb{N}$ onto $P(\mathbb{N})$, there is some $m \in \mathbb{N}$ such that $F(m) = A$. Thus $m \notin F(m)$ if and only if $m \in A$. This is impossible since $F(m) = A$. Hence there is no mapping of $\mathbb{N}$ onto $P(\mathbb{N})$ $\square$

Exercise 6.3 [$\mathbb{R}$ is not denumerable] *Prove that $\mathbb{R}$ is not denumerable.*

Hint: *Suppose that the interval $]0, 1] \subset \mathbb{R}$ is denumerable. List the elements of $]0, 1]$ in their binary representation:*

$$\begin{aligned} x_1 &: 0.a_{11}a_{12}a_{13}a_{14}a_{15}\ldots \\ x_2 &: 0.a_{21}a_{22}a_{23}a_{24}a_{25}\ldots \\ x_3 &: 0.a_{31}a_{32}a_{33}a_{34}a_{35}\ldots \\ x_4 &: 0.a_{41}a_{42}a_{43}a_{44}a_{45}\ldots \end{aligned}$$

Construct a non-terminating decimal $0.b_1b_2b_3b_4b_5\ldots$ such that $b_n \neq a_{nn}$

Exercise 6.4 [$\mathbb{Z} \times \mathbb{Z}$ is denumerable] *Prove theorem 6.8.*

Exercise 6.5 [$\mathbb{Q}$ is denumerable] *Prove theorem 6.9.*

Chapter 7

Gödel's Incompleteness Theorem

7.1 Introduction

In mathematical logic, a theory is a set of sentences expressed in a formal language. Some statements in a theory are included without proof (these are the axioms of the theory), and others (the theorems) are included because they are implied by the axioms.

Because statements of a formal theory are written in symbolic form, it is possible to mechanically verify that a formal proof from a finite set of axioms is valid. This task, known as automatic proof verification.

Many theories of interest include an infinite set of axioms, however. To verify a formal proof when the set of axioms is infinite, it must be possible to determine whether a statement that is claimed to be an axiom is actually an axiom. This issue arises in first order theories of arithmetic, such as Peano arithmetic, because the principle of mathematical induction is expressed as an infinite set of axioms (an axiom schema).

A formal theory is said to be *effectively generated* if its set of axioms is a recursively enumerable set. This means that there is a computer program that, in principle, could enumerate all the axioms of the theory without listing any statements that are not axioms. This is equivalent to the ability to enumerate all the theorems of the theory without enumerating any statements that are not theorems. For example, each of the theories of Peano arithmetic and Zermelo-Fraenkel set theory has an infinite number of axioms each is effectively generated.

In choosing a set of axioms, one goal is to be able to prove as many correct results as possible, without proving any incorrect results. A set of axioms is *complete* if, for any statement in the axioms' language, either that statement or its negation is provable from the axioms. A set of axioms is *consistent* if there is no statement so that both the statement and its negation are provable from the axioms. In the standard system of first-order logic, an inconsistent set of axioms will prove every statement in its language (this is sometimes called the principle of explosion), and is thus automatically complete. A set of axioms that is both complete and consistent, however, proves a maximal set of non-contradictory theorems. Gödel's incompleteness theorems show that in certain cases it is not possible to obtain an effectively generated, complete, consistent theory.

7.2 Gödel's First Incompleteness Theorem

Theorem 7.1 [Gödel's First Incompleteness Theorem] *Any effectively generated theory*

capable of expressing elementary arithmetic cannot be both consistent and complete. In particular, for any consistent, effectively generated formal theory that proves certain basic arithmetic truths, there is an arithmetical statement that is true, but not provable in the theory [Gö31].

The true but unprovable statement referred to by the theorem is often referred to as *the Gödel sentence* for the theory. It is not unique; there are infinitely many statements in the language of the theory that share the property of being true but unprovable.

For each consistent formal theory T having the required small amount of number theory, the corresponding Gödel sentence G asserts: *G cannot be proved to be true within the theory T*. If G were provable under the axioms and rules of inference of T, then T would have a theorem, G, which effectively contradicts itself, and thus the theory T would be inconsistent. This means that if the theory T is consistent then G cannot be proved within it. This means that G's claim about its own unprovability is correct; in this sense G is not only unprovable but true. Thus provability-within-the-theory-T is not the same as truth; the theory T is incomplete.

If G is true: G cannot be proved within the theory, and the theory is incomplete. If G is false: then G can be proved within the theory and then the theory is inconsistent, since G is both provable and refutable from T.

Each theory has its own Gödel statement. It is possible to define a larger theory T' that contains the whole of T, plus G as an additional axiom. This will not result in a complete theory, because Gödel's theorem will also apply to T', and thus T' cannot be complete. In this case, G is indeed a theorem in T', because it is an axiom. Since G states only that it is not provable in T, no contradiction is presented by its provability in T'. However, because the incompleteness theorem applies to T': there will be a new Gödel statement G' for T', showing that T' is also incomplete. G' will differ from G in that G' will refer to T', rather than T.

To prove the first incompleteness theorem, Gödel represented statements by numbers. Then the theory at hand, which is assumed to prove certain facts about numbers, also proves facts about its own statements. Questions about the provability of statements are represented as questions about the properties of numbers, which would be decidable by the theory if it were complete. In these terms, the Gödel sentence states that no natural number exists with a certain, strange property. A number with this property would encode a proof of the inconsistency of the theory. If there were such a number then the theory would be inconsistent, contrary to the consistency hypothesis. So, under the assumption that the theory is consistent, there is no such number.

7.3 Proof of Gödel's First Incompleteness Theorem

7.3.1 The System

Gödel builds a formal system called P defined over the alphabet Σ defined as follows:

Constants: $\neg$ (not), $\vee$ (or), $\forall$ (for all), 0 (zero), S (the successor of), (,) (parentheses).

Variables of Type One: $x_1, y_1, z_1 \ldots$ for natural numbers.

Variables of Type Two: $x_2, y_2, z_2 \ldots$ for subsets of natural numbers.

Variables of Type Three: $x_3, y_3, z_3 \ldots$ for sets of subsets of natural numbers.

Then he defines a set of axioms for P:

Peano Axioms: which give fundamental properties for natural numbers.

1. $\neg(S(x_1) = 0 =$
2. $S(x_1) = S(x_2) \Rightarrow x_1 = x_2$
3. $(x_2(0) \wedge \forall x_1 : x_2(x_1)) \Rightarrow x_2(S(x_1)) \Rightarrow \forall x_1 : x_2(x_1)$. We can prove a predicate x_2 on natural numbers by natural induction.

Proposition Axioms: Every formula obtained by inserting arbitrary formulae for p, q, r in the following schemata.

1. $p \vee p \Rightarrow p$
2. $p \Rightarrow p \vee q$
3. $p \vee q \Rightarrow q \vee p$
4. $(p \Rightarrow q) \Rightarrow (r \vee p \Rightarrow r \vee q)$

Quantor Axioms: Every formula obtained from the two schemata

1. $(\forall v : a) \Rightarrow SUBST\, a \begin{pmatrix} v \\ c \end{pmatrix}$
2. $(\forall v : b \vee a)) \Rightarrow (b \vee \forall v : a)$

Insert an arbitrary formula for a, an arbitrary variable for v, any formula where v does not occur free[1] for b, and for c a sign of the same type as v with the additional requirement that c does not contain a free variable that would be bound in a position in a where v is free.

Reducibility Axiom: Every formula obtained from the schema

1. $\exists u : \forall v : (U(v) \Leftrightarrow a)$

by inserting for v and u any variables of type n and $n+1$ respectively and for a a formula that has no free occurrence of u.

Set Axiom: Any formula obtained from the following by type-lift (and the formula itself)

1. $(\forall x_1 : (x_2(x_1) \Leftrightarrow y_2(x_1))) \Rightarrow x_2 = y_2$

This axiom states that a set is completely determined by its elements.

Definition 7.1 [Statement] *A* statement *is a formula that has no free variables.*

Note that any statement can be interpreted as a string over Σ.

We are now preparing for the definition of theorem. Recall that in any theory there are axioms and rules of inference. Axioms are carefully chosen statements and rules of inference are certain operations on strings.

Definition 7.2 [Inference] *A formula c is called the* immediate consequence *of a and b (of a) if a is the formula $\neg b \vee c$ or if c is the formula $\forall v.a$, where v is any variable*[2]

[1] A *free variable* is a notation that specifies places in an expression where substitution may take place.
[2] Note that $\neg b \vee c$ is equivalent to $b \rightarrow c$.

The set of provable formulae is defined as the smallest class of formulae that contains the axioms and is closed under the relation *immediate consequence.*

Now we are ready to define theorem.

Definition 7.3 [Theorem] *p is a theorem if and only if*

1. *p is an axiom or*
2. *there is a sequence $p_1, \ldots, p_n$ where $p_1, \ldots, p_n$ are theorems and we can derive p from $p_1, \ldots, p_n$ by using rules of immediate consequence.*

7.3.2 Gödel Numbers

At this point, Gödel introduces his famous Gödel numbers[3], identifying each mathematical expression with a unique natural number.

$\ulcorner 0 \urcorner$	0
$\ulcorner S \urcorner$	1
$\ulcorner \neg \urcorner$	2
$\ulcorner \vee \urcorner$	3
$\ulcorner \forall \urcorner$	4
$\ulcorner (\urcorner$	5
$\ulcorner) \urcorner$	6
$\ulcorner x_n \urcorner$	$6 + n + 3k$ for each variable x_n of type n

.

Any formula $s = s_1, \ldots, s_n$ is a string of symbols, we encode s as $\ulcorner s \urcorner = \langle s_1, \ldots, s_n \rangle$

7.3.3 Primitive Recursive Functions

In the next step, Gödel defines a lot of primitive recursive functions (see also section 2.1.1). Furthermore, Gödel shows the existence of a primitive recursive predicate *isTheorem*. This predicate is definable inside the system P.

7.3.4 Provability

A formula $F(x)$ that contains exactly one free variable x is called a *statement form.* As soon as x is replaced by a specific number, the statement form turns into a *statement*, and it is then either provable in the system, or not. For certain formulas one can show that for every natural number n, $F(n)$ is true if and only if it can be proven. In particular, this is true for every specific arithmetic operation between a finite number of natural numbers, such as $2 * 3 = 6$.

Statement forms themselves are not statements and therefore cannot be proved or disproved. But every statement form $F(x)$ can be assigned with a Gödel number which we will denote by $G(F)$. The choice of the free variable used in the form $F(x)$ is not relevant to the assignment of the Gödel number $G(F)$.

Now comes the trick: The notion of provability itself can also be encoded by Gödel numbers, in the following way. Since a proof is a list of statements which obey certain rules, we can define

[3]The original Gödel numbers have been defined differently.

the Gödel number of a proof. Now, for every statement p, we may ask whether a number x is the Gödel number of its proof. The relation between the Gödel number of p and x, the Gödel number of its proof, is an arithmetical relation between two numbers. Therefore there is a statement form $Proof(x)$ that uses this arithmetical relation to state that a Gödel number of a proof of x exists:

> $Proof(y) = \exists x : y$ is the Gödel number of a formula and x is the Gödel number of a proof of the formula encoded by y.

An important feature of the formula $Proof(y)$ is that if a statement p is provable in the system then $Proof(G(p))$ is also provable. This is because any proof of p would have a corresponding Gödel number, the existence of which causes $Proof(G(p))$ to be satisfied.

7.3.5 Diagonalisation

The next step in the proof is to obtain a statement that says it is unprovable. Although Gödel constructed this statement directly, the existence of at least one such statement follows from the diagonal lemma, which says that for any sufficiently strong formal system and any statement form F there is a statement p such that the system proves

$$p \leftrightarrow F(G(p))$$

We obtain p by letting F be the negation of $Proof(x)$; thus p roughly states that its own Gödel number is the Gödel number of an unprovable formula.

The statement p is not literally equal to $\neg Proof(G(p))$; rather, p states that if a certain calculation is performed, the resulting Gödel number will be that of an unprovable statement. But when this calculation is performed, the resulting Gödel number turns out to be the Gödel number of p itself. This is similar to the following sentence in English:

> ", when preceded by itself in quotes, is unprovable.", when preceded by itself in quotes, is unprovable.

This sentence does not directly refer to itself, but when the stated transformation is made the original sentence is obtained as a result, and thus this sentence asserts its own unprovability. The proof of the diagonal lemma employs a similar method.

7.3.6 Proof of independence

We will now assume that our axiomatic system is consistent. We let p be the statement obtained in the previous section.

If p were provable, then $Proof(G(p))$ would be provable, as argued above. But p asserts the negation of $Proof(G(p))$. Thus our system would be inconsistent, proving both a statement and its negation. This contradiction shows that p cannot be provable.

If the negation of p were provable, then $Proof(G(p))$ would be provable (because p was constructed to be equivalent to the negation of $Proof(G(p))$). However, for each specific number x, x cannot be the Gödel number of the proof of p, because p is not provable (from the previous paragraph). Thus on one hand the system proves there is a number with a

certain property (that it is the Gödel number of the proof of p), but on the other hand, for every specific number x, we can prove that it does not have this property. This is impossible. Thus the negation of p is not provable.

Thus the statement p is undecidable: it can neither be proved nor disproved within the system.

Note that if one tries to *add the missing axioms* to avoid the undecidability of the system, then one has to add either p or $\neg p$ as axioms. But then the definition of *being a Gödel number of a proof* of a statement changes. Which means that the statement form $Proof(x)$ is now different. Thus when we apply the diagonal lemma to this new form $Proof$, we obtain a new statement p, different from the previous one, which will be undecidable in the new system if it is consistent.

7.4 Gödel's Second Incompleteness Theorem

Theorem 7.2 [Gödel's Second Incompleteness Theorem] *For any formal effectively generated theory T including basic arithmetical truths and also certain truths about formal provability, T includes a statement of its own consistency if and only if T is inconsistent [Gö31].*

This strengthens the first incompleteness theorem, because the statement constructed in the first incompleteness theorem does not directly express the consistency of the theory. The proof of the second incompleteness theorem is obtained, essentially, by formalising the proof of the first incompleteness theorem within the theory itself.

7.5 Proof of Gödel's Second Incompleteness Theorem

The main difficulty in proving the second incompleteness theorem is to show that various facts about provability used in the proof of the first incompleteness theorem can be formalised within the system using a formal predicate for provability. Once this is done, the second incompleteness theorem essentially follows by formalising the entire proof of the first incompleteness theorem within the system itself.

Let p stand for the undecidable sentence constructed above, and assume that the consistency of the system can be proven from within the system itself. We have seen above that if the system is consistent, then p is not provable. The proof of this implication can be formalised within the system, and therefore the statement "p is not provable", or "$\neg P(p)$" can be proven in the system.

But this last statement is equivalent to p itself (and this equivalence can be proven in the system), so p can be proven in the system. This contradiction shows that the system must be inconsistent.

Chapter 8

Solutions of the Exercises

8.1 Turing machines

Exercise 1.1

We need to take a program P and modify it so it:

1. Never halts unless we explicitly want it to, and
2. Halts whenever it prints `hello, world`.

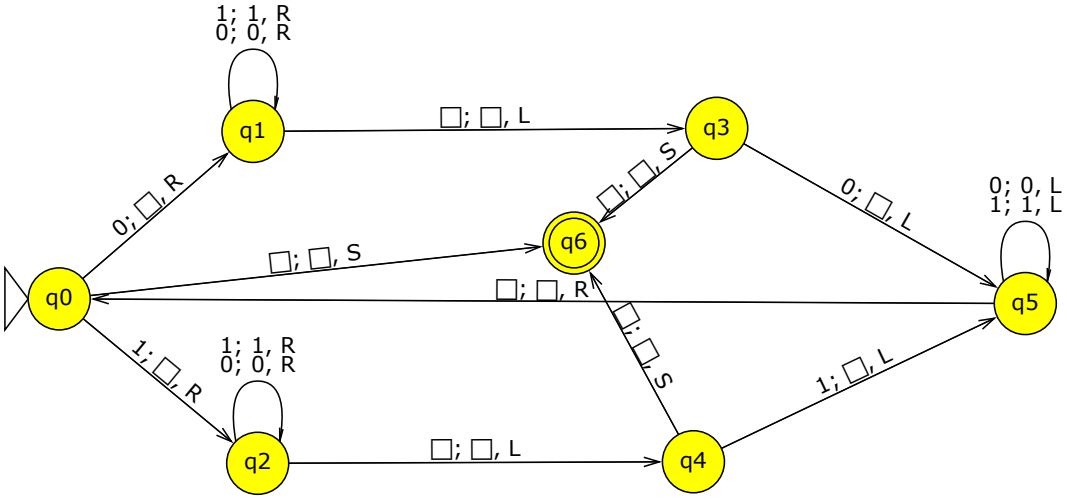
For (1), we can add a loop such as `while(true){x=x;}` to the end of main, and also at any point where main returns. That change catches the normal ways a program can halt, although it doesn't address the problem of a program that halts because some exception such as division by `0` or an attempt to read an unavailable device. Technically, we'd have to replace all of the exception handlers in the run-time environment to cause a loop whenever an exception occurred.

For (2), we modify P to record in an array the first 12 characters printed. If we find that they are `hello, world`, we halt by going to the end of main (past the point where the while-loop has been installed).

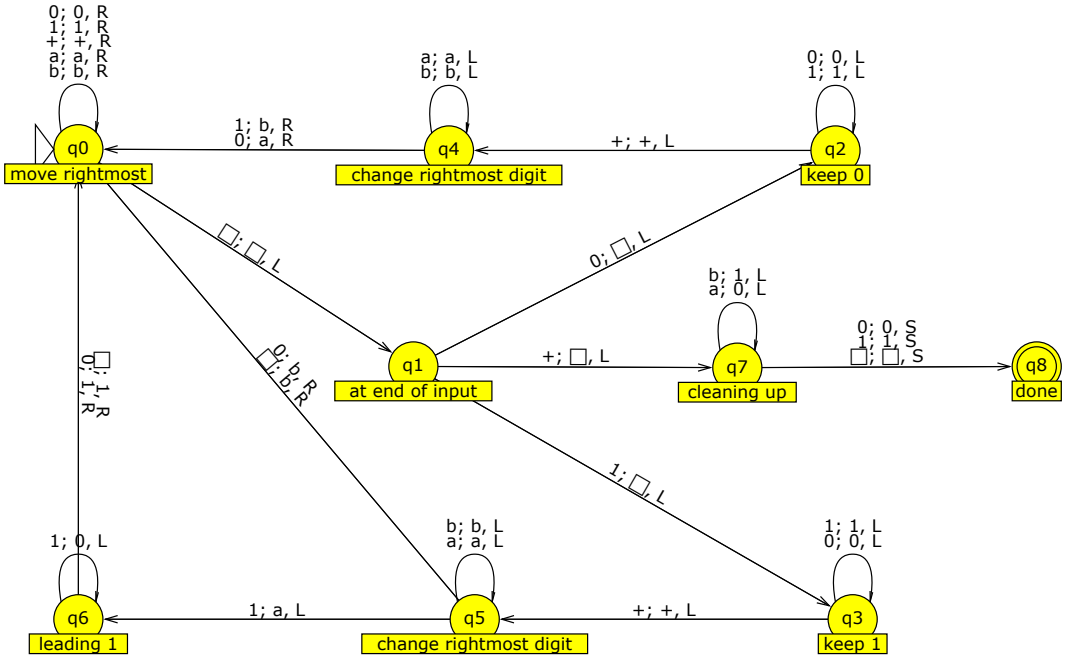
8.1.1 Problems that Computers Cannot Solve

8.1.2 Turing Machines

Exercise 1.2



Exercise 1.3



8.2 Recursive Functions

8.2.1 Primitive Recursive Functions

Exercise 2.1

$mul = PR(C_0^1, add \circ (U_1^3, U_3^3))$

Exercise 2.2

$p \rightarrow q = sg((p \perp 1) + q)$
$p \leftrightarrow q = sg((sg((p \perp 1) + q) + sg(p + (q \perp 1)) \perp 1)$
$p \oplus q = sg((p \perp q) + (q \perp p))$

Exercise 2.3

$f(n) = \sum_{i=1}^{k} eq(a_i, n)n_i$

8.2.2 Recursive Functions

Exercise 2.4

1.
$$\begin{aligned} a(2,1) &= a(a(1,1),0) \\ &= a(a(a(0,1),0),0) \\ &= a(a(1,0),0) \\ &= a(2,0) \\ &= 4 \end{aligned}$$

2. **Basis:** $a(1,1) = a(a(0,1),0) = a(1,0) = 2$.
   Induction: $a(n+1,1) = a(a(n,1),0) = a(2n,0) = 2n+2 = 2(n+1)$

3. **Basis:** $a(0,2) = 1 = 2^0$.
   Induction: $a(n+1,2) = a(a(n,2),1) = a(2^n,1) = 2 \cdot 2^n = 2^{n+1}$.

4. $a(4,3) = \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n \quad \text{terms}}$.
   Since $a(0,3) = 1$ and $a(n+1,3) = a(a(n,3),2) = 2^{a(n,3)}$

Exercise 2.5

Basis: $a_0(n) = a(n,0) = 1 =$ so $a_0 \in F(PRIM)$

Induction: We have

$$\begin{aligned} a_{n+1}(0) &= a(0,n+1) = 1 = a_n(1) \\ a_{n+1}(m+1) &= a(m+1,n+1) = a(a(m,n+1),n) = a(a_{n+1}(m),n) = a_n(a_{n+1}(m)) \end{aligned}$$

8.2.3 Gödel Numbers

Exercise 2.6

$$\begin{aligned} \ulcorner SUB(S,S) \urcorner &= \langle 4, 1, \ulcorner S \urcorner, \langle \ulcorner S \urcorner \rangle \rangle \\ &= \langle 4, 1, \langle 1,1 \rangle, \langle \langle 1,1 \rangle \rangle \rangle \\ &= \langle 4, 1, \langle 1,1 \rangle, \langle 1,1 \rangle \rangle \\ &= \langle 4, 1, 20, \langle 20 \rangle \rangle \\ &= \langle 4, 1, 20, 231 \rangle \\ &= 829280947762162434748904053324349 7 \end{aligned}$$

8.2.4 Computability

8.2.5 Functions, Sets and Languages

Exercise 2.7

Consider following recursive predicate:

$$\begin{aligned} even(0) &= 1 \\ even(n+1) &= 1 - even(n) \end{aligned}$$

even is primitive recursive and $\chi_{2\mathbb{N}} = even$

8.2.6 Universal Function

Exercise 2.8

For example 0. Each recursive function is coded as a tuple. The first component of the tuple is its type. Here we get the type with $\pi_1(0) = 0$. But there is no recursive construction of type 0.

8.2.7 The Recursion Theorem

Exercise 2.9

Should be written on a single line of code without whitespaces.

```
import java.text.*;class a{public static void main(String x[])
{char b[]={34};char c[]={123};String s[]=new String[3];s[0]=
"import java.text.*;class a{2}public static void main(String
x[]){2}char b[]={2}34};char c[]={2}123};String s[]=new String[3];
s[0]={1}{0}{1};s[1]=new String(b);s[2]=new String(c);
System.out.println(MessageFormat.format(s[0],s));}}";
s[1]=new String(b);s[2]=new String(c);System.out.println(
MessageFormat.format(s[0],s));}}
```

See also `http://www.nyx.net/~gthompso/self_java.txt` for more solutions.

8.2.8 Problem Reduction

Exercise 2.10

Let B be a set that is r.e. but not recursive (e.g. the halting problem) then by the second property of being r.e.-complete it follows that $B <_m A$. Now if A was recursive then by theorem 2.20 we could conclude that B is recursive, contradicting the initial assumption that B is not recursive.

8.3 The 3 Computational Models

8.4 Mathematical Background

8.4.1 The natural Numbers

Exercise 6.1

Basis: $m * (n + 0) = m * n = m * n + 0 = m * n + m0$

Induction:

$$\begin{aligned} m(n + S(p)) &= mS(n + p) \\ & \quad m * (n + p) + m \\ & \quad (m * n + m * p) + m \\ & \quad m * n + (m * p + m) \\ & \quad m * n + m * S(p) \end{aligned}$$

Exercise 6.2

1. Recall that $1 = S(0)$. Thus $m*1 = m*S(0) = m*0+m = m$. We prove that $1*m = m$ by induction on m:

   Basis: $1 * 0 = 0$

   Induction: $1 * S(m) = 1 * m + m = m + 1 = m + S(0) = S(m + 0) = S(m)$

2. Induction over p:

   Basis: $(m * n) * 0 = 0$ $m * (n * 0) = m * 0 = 0$

   Induction:

$$\begin{aligned} (m * n) * S(p) &= (m * n) * p + m * n \\ &= m * (n * p) + m * n \\ &= m * (n * p + n) \\ &= m * (n * S(p)) \end{aligned}$$

3. Induction over n:

   Basis: $m * 0 = 0 = 0 * m$ Prove it by induction over m.

   $n * S(n) = m * n * m = n * m + 1 * m = (n + 1) + m = S(n) * m$

8.4.2 Counting

Exercise 6.3

The real number $0.b_{11}b_{22}b_{33}\ldots$ where $b_{ii} = 1 - a_{ii} \quad \forall i \in \mathbb{N}$ is not in the list. It differs at position i from x_i.

This method is called Cantor's diagonal argument[1].

[1]Cantor's diagonal argument, also called the diagonalisation argument, the diagonal slash argument or the diagonal method, was published in 1891 by Georg Cantor as a proof that there are infinite sets which cannot be put into one-to-one correspondence with the infinite set of natural numbers. Such sets are now known as uncountable sets, and the size of infinite sets is now treated by the theory of cardinal numbers which Cantor began.

Exercise 6.4

Since $\mathbb{Z}$ is denumerable, there is a one to one function ϕ from $\mathbb{Z}$ onto $\mathbb{N}$. Now $\mathbb{Z} \times \mathbb{Z}$ is the same than $\mathbb{N} \times \mathbb{N}$.

Exercise 6.5

There is a one to one function ϕ from $\mathbb{Z} \times (\mathbb{Z} - \{0\})1$ to $\mathbb{Q}$ defined by $\phi(p, q) = \frac{p}{q}$ (this function is not onto. Why?). Thus $\mathbb{Q}$ is denumerable.

8.5 FOR & WHILE Programs

8.5.1 WHILE Programs

Exercise 5.1

The for loop $FOR(x_i, P)$ can be written as $SEQ(SET(x_{n+1}, x_i), WHILE(x_{n+1}, P)$

8.5.2 Computable Functions

Exercise 5.2

If $P = WHILE(x_i, P_1)$, then $\Phi_P(S) = \mu r : P_i^n(\Phi_P^{(r)}(S))$ according to the semantics of a WHILE while loop in section 5.2.2.

It follows that WHILE computable functions are recursive.

Bibliography

[ALSU08] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compiler*. Pearson Studium, München, 2008.

[AS09] K. Ambos-Spies. Theoretische informatik. Technical report, Ruprecht-Karls Universität, Heidelberg, 2009.

[ASU07] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, Boston, 2nd edition, 2007.

[CE63] L.W. Cohen and G. Ehrlich. *The Structure of the Real Number System*. The University Series in Undergraduate Mathematics. D. van Nostrand, Princeton, 1963.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. `http://www.chomsky.info/`.

[Fle06] P. Flener. Formal languages and automata theory, 2006. `http://user.it.uu.se/`.

[Gö31] K. Gödel. On formally undecidable propositions of principia mathematica and related systems. *Monatshefte für Mathematik und Physik*, 38:173–98, 1931. Translated by M. Hirzel `http://www.research.ibm.com`.

[HMU07] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Automata Theory, Languages and Computation*. Pearson Education, Boston, 3rd edition, 2007.

[Hå08] J. Håstad. Complexity theory. Technical report, Royal Institute of technology, Stockholm, 2008. `http://www.nada.kth.se`.

[JCC96] JavaCC, java compiler compiler [tm], 1996. `https://javacc.dev.java.net`.

[Les75] M.E. Lesk. Lex – a lexical analyser generator. Technical Report Computer Science Technical Report 39, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[Nip98] Tobias Nipkow. Lambda-kalkül, 1998. `http://www4.informatik.tu-muenchen.de/lehre/vorle`

[Poh93] W. Pohlers. Handbuch der informatik. Technical report, Westfälische Wilhelm-Universität, Münster, 1993.

[Rm] S. Rodger and many. Jflap. `http://jflap.org`.

[Sel07] Peter Selinger. Lecture notes on the lambda calculus, 2007. `http://www.mscs.dal.ca/~selinger/papers/lambdanotes.pdf`.

[Smi04] Michael Smith. λ calculus – church numerals and lists, 2004. `http://lanther.co.uk/compsci/LambdaList.pdf`.

[Tur37] A.M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1937. `http://www.comlab.ox.ac.uk`.

[vL90a] J. van Leeuwen. *Handbook of Theoretical Computer Science, Algorithms and Complexity*, volume A. Elsevier, Amsterdam, 1990.

[vL90b] J. van Leeuwen. *Handbook of Theoretical Computer Science, Formal Models and Semantics*, volume B. Elsevier, Amsterdam, 1990.

[Wik] Wikipedia. `http://en.wikipedia.org`.

[Wir86] N. Wirth. *Compilerbau*. Teubner, Stuttgart, 1986.

[Wir96] N. Wirth. *Compiler Construction*. Addison-Wesley, Reading, 1996. See also `http://www-old.oberon.ethz.ch`.