# Short Lecture Notes — Computability (2008)

Roberto Zunino
Dipartimento di Ingegneria e Scienza dell'Informazione
Università degli Studi di Trento
zunino@disi.unitn.it

Preliminary Version — 27 May 2009

## General Information

These notes are meant to be a short summary of the topics covered in my *Computability* course kept in 2008 at Trento. Students are welcome to use these notes, provided they understand the following.

- These notes are *work in progress*. I will update and expand them, so at any time (but the very end of the course) they do not comprise all the topics which are needed for the exam. As a consequence, please do not rely on an *old* version of these notes.
- Even when these notes will be completed, you might need to refer to the books for some parts. I will try to point to these parts in the notes.
- Reporting errors in these notes will be awarded.

Roberto Zunino

# Contents

# Chapter 1

# Basics

## 1.1 Introduction

## 1.2 Logic Notation

The following exercises are meant to check your formula-related skills.

**Exercise 1.** *Describe the meaning of the formulas below.*

$$
\begin{aligned}
&p \vee \neg p \\
&\neg(p \vee q) \iff (\neg p \wedge \neg q) \\
&(p \implies q) \iff (\neg p \vee q) \\
&(p \wedge q \implies r) \iff (p \implies (q \implies r)) \\
&(p \implies q) \iff (\neg q \implies \neg p) \\
&(p \implies (\forall x.\, q(x))) \iff (\forall x.\, p \implies q(x)) \\
&((\forall x.\, p(x)) \implies q) \iff (\exists x.\, p(x) \implies q) \\
&\exists y. \forall x.\, p(x, y) \implies \forall x. \exists y.\, p(x, y) \\
&\forall x. \exists y.\, p(x, y) \not\Longrightarrow \exists y. \forall x.\, p(x, y) \\
&\exists! x.\, p(x) \iff \exists c.\, (\forall x.\, p(x) \iff x = c) \\
&\exists! x.\, p(x) \iff (\exists x.\, p(x)) \wedge (\forall x, y.\, p(x) \wedge p(y) \implies x = y)
\end{aligned}
$$

**Exercise 2.** *Prove (or refute) the formulas above.*

## 1.3 Set Theory

Let $A, B, \ldots, X, Y, Z$ be sets.

$$\forall x \in X.\, p(x) \iff \forall x.\, x \in X \implies p(x)$$
$$\exists x \in X.\, p(x) \iff \exists x.\, x \in X \wedge p(x)$$
$$\bigcup X = \bigcup_{Y \in X} Y = \{y | \exists Y \in X.\, y \in Y\}$$
$$\bigcup \{\{1,2,3\},\{4,5\},\emptyset\} = \{1,2,3,4,5\}$$
$$A \cup B = \bigcup \{A, B\} = \{x | x \in A \vee x \in B\}$$
$$\bigcap X = \bigcap_{Y \in X} Y = \{y | \forall Y \in X.\, y \in Y\}$$
$$\bigcap \{\{1,2,3\},\{3,4,5\}\} = \{3\}$$
$$A \cap B = \bigcap \{A, B\} = \{x | x \in A \wedge x \in B\}$$
$$A \setminus B = \{x | x \in A \wedge x \notin B\}$$
$$X \subseteq Y \iff \forall x \in X.\, x \in Y$$
$$\mathcal{P}(A) = \{B | B \subseteq A\}$$

We shall use ordered pairs $\langle x, y \rangle$, as well as ordered tuples.

$$\langle x, y \rangle = \langle x', y' \rangle \iff (x = x' \wedge y = y')$$
$$X \times Y = \{\langle x, y \rangle | x \in X \wedge y \in Y\}$$

**Exercise 3.** *Define* $\forall \langle x, y \rangle \in Z.\, p(x, y)$ *using the notation seen above.*

The disjoint union of two sets: we use 0 and 1 as tags to keep the two sets disjoint.

$$A \uplus B = \{\langle 0, a \rangle | a \in A\} \cup \{\langle 1, b \rangle | b \in B\}$$

**Definition 4.** *To our purposes, the set of* functions *from a set* $A$ *to a set* $B$*, written* $(A \to B)$ *is defined as*

$$(A \to B) = \{f | f \subseteq A \times B \wedge \forall a \in A. \exists! b \in B.\, \langle a, b \rangle \in f\}$$

*The* domain *of* $f \in (A \to B)$ *is* $\mathsf{dom}(f) = \{a | \langle a, b \rangle \in f\} = A$*. The* range *of* $f \in (A \to B)$ *is* $\mathsf{ran}(f) = \{b | \langle a, b \rangle \in f\} \subseteq B$*.* □

So, a function is a *set of pairs*, mapping each element $a$ of its domain $A$ to exactly one element $f(a)$ of its range (some subset of $B$).

**Definition 5.** *A function $f$ is injective when*

$$\forall x, y \in \mathsf{dom}(f).\, f(x) = f(y) \implies x = y$$

□

**Exercise 6.** *Prove the following to be equivalent to $f$ being injective.*

$$f^{-1} \in (\mathsf{ran}(f) \to \mathsf{dom}(f)) \quad \textit{where} \quad f^{-1} = \{\langle b, a\rangle | \langle a, b\rangle \in f\}$$

We shall often deal with *partial functions*.

**Definition 7.** *The set of partial functions $(A \rightsquigarrow B)$ is defined as*

$$(A \rightsquigarrow B) = \{f | \exists A' \subseteq A.\, f \in (A' \to B)\}$$

□

The domain of partial function $f \in (A \rightsquigarrow B)$ is therefore a *subset* of $A$. This means that the expression $f(a)$ when $a \in A$ is actually *undefined* whenever $a$ is not in $\mathsf{dom}(f)$. In informal terms, a partial function is a function that might fail to deliver any result. Formally, while a "true" function returns exactly one result, a partial function returns *at most* one result.

Sometimes we shall use the term *total function* for a function $f \in (A \to B)$ to stress the fact that $f$ is completely defined on $A$, i.e. $\mathsf{dom}(f) = A$.

**Exercise 8.** *Try to classify the following operations as "partial" or "total". Be precise on what $A$ and $B$ are in your model.*

- *addition,subtraction,multiplication,division on natural numbers*
- *compiling a Java program*
- *compiling a Java program, then running it and taking its output*
- *downloading a file from a server*
- *executing a* `COMMIT` *SQL statement*

**Definition 9.** *A function $f \in (A \to B)$ is said to be* surjective *when $\mathsf{ran}(f) = B$. An injective and surjective function is said to be a* bijection. □

**Note.** If $f$ is a partial function, arguing whether $f$ is a total function is meaningless unless the set $A$ is clear from the context: every partial $f$ is a *total* function in $(\mathsf{dom}(f) \to \mathsf{ran}(f))$, for instance.

**Note 2.** Similarly, if $f$ is a function, arguing whether $f$ is surjective is meaningless unless the set $B$ is clear from the context: every $f$ is *surjective* in $(\mathsf{dom}(f) \to \mathsf{ran}(f))$.

**Note 3.** The same holds for *bijections*.

### 1.3.1 Further Notation

For this course, we shall use

$$\begin{aligned}
&\mathbb{N} = \{0, 1, 2, \ldots\} \\
&\bar{A} = \mathbb{N} \setminus A \\
&\chi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

The function $\chi_A$ is called the *characteristic function* of the set $A$.

## 1.4 Induction

We briefly recall some facts about induction. Consider the usual induction principle on $\mathbb{N}$:

**Theorem 10** (Induction Principle)**.** *Given a predicate $p$ on $\mathbb{N}$, we have $\forall n \in \mathbb{N}. p(n)$ iff both of these hold*

$$p(0)$$

$$\forall m \in \mathbb{N}. \quad p(m) \implies p(m+1)$$

**Exercise 11.** *Prove* $\forall n \in \mathbb{N}. 0 + 1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$.

Note that naturals can be *inductively* defined as the smallest set containing a constant 0, and closed under a successor function $s(-)$. This is basically a syntactic definition, defining e.g. 3 as the syntactic expression $s(s(s(0)))$. Usually, inductive definitions such as this one are written using inference rules:

$$\frac{}{0 \in N} \qquad \frac{n \in \mathbb{N}}{s(n) \in \mathbb{N}}$$

In other terms,

$$\mathbb{N} = \bigcap \{X | 0 \in X \wedge \forall n. n \in X \implies s(n) \in X\}$$

Note that the induction principle (Th.10) closely matches this definition.

We can also express the same inductive definition in a more set-theoretic fashion, using a recursive equation

$$\mathbb{N} \simeq 1 \uplus \mathbb{N}$$

where 1 here denotes a set with one element (the constant 0 in this case). The equation above unfolds as $\mathbb{N} = 1 \uplus (1 \uplus (1 \uplus \cdots))$ where each 1 here denotes a unique constant: indeed, they are forced to be unique by the disjoint union ($\uplus$). The smallest set that satisfies the recursive equation is indeed the set of naturals.

If you recall *context free grammars*, you will find the above recursive equation similar to

$$N \leftarrow 0 \,|\, s(N)$$

Indeed, grammars are a kind of inductive definitions.

**Exercise 12.** *Starting from the grammar of binary trees (of naturals)*

$$T \leftarrow N \,|\, b(T, T)$$

*write a recursive set definition for the set of binary trees. You can use* $\mathbb{N}, \times, \uplus$ *for this.*

**Exercise 13.** *Express the set* $T$ *of Ex. 12 using* $\bigcap$*:*

$$\mathbb{T} = \bigcap \{X \,|\, \cdots\}$$

**Exercise 14.** *Define* $A^*$*, the set of finite sequences (i.e. strings) of elements of the set* $A$*.*

**Exercise 15.** *(For the logically minded people)*
*Write an induction principle for* $\mathbb{T}$*.*

## 1.5 Cardinality

### 1.5.1 Bijections of $\mathbb{N} \uplus \mathbb{N}, \mathbb{N} \times \mathbb{N}, \mathbb{N}^*, \ldots$ in $\mathbb{N}$

For $\mathbb{N} \uplus \mathbb{N}$:

$$\mathsf{encode}_{\uplus}(x) = \begin{cases} 2n & \text{if } x = \langle 0, n \rangle \\ 2n+1 & \text{if } x = \langle 1, n \rangle \end{cases}$$

**Exercise 16.** *Write the inverse function* $\mathbb{N} \to \mathbb{N} \uplus \mathbb{N}$*.*

Sometimes, we use

$$\mathsf{inL}(n) = \mathsf{encode}_{\uplus}\langle 0, n\rangle \qquad\qquad \mathsf{inR}(n) = \mathsf{encode}_{\uplus}\langle 1, n\rangle$$

For $\mathbb{N} \times \mathbb{N}$:

$$\mathsf{encode}_{\times}(\langle n, m\rangle) = \frac{(n+m)(n+m+1)}{2} + n$$

**Exercise 17.** *Describe the inverse function* $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$. *This is usually seen as* two *projection functions* proj1 *and* proj2.

**Theorem 18.** *There a bijection between* $\mathbb{N}$ *and* $\mathbb{N}^*$

*Proof.* Left as an exercise. □

**Exercise 19.** *Describe how to use these encodings to construct the following bijections:*

- *the language of arithmetic expressions* $\leftrightarrow \mathbb{N}$
- *the set of all files* $\leftrightarrow \mathbb{N}$
- *the language of logic formulas* $\leftrightarrow \mathbb{N}$

**Exercise 20.** *Define a bijection between* $\mathbb{N}$ *and* $\mathbb{Q}$.

**Exercise 21.** *Prove that*

$$A \cap B = \emptyset \implies \exists f \in (A \cup B \leftrightarrow A \uplus B)$$

**Lemma 22.**

$$\mathsf{encode}_{\times}(\langle n, m\rangle) \geq n$$
$$\mathsf{encode}_{\times}(\langle n, m\rangle) \geq m$$

*Proof.* The first part is trivial:

$$\mathsf{encode}_{\times}(\langle n, m\rangle) = \frac{(n+m)(n+m+1)}{2} + n \geq n$$

For the second part

$$\mathsf{encode}_{\times}(\langle n, m\rangle) = \frac{(n+m)(n+m+1)}{2} + n \geq \frac{(n+m)(n+m+1)}{2} =$$
$$= \frac{n^2 + m^2 + 2nm + n + m}{2} \geq \frac{m^2 + m}{2} \geq \frac{m+m}{2} = m$$

where the last steps follow from $m^2 \geq m$, which holds for all $m \in \mathbb{N}$. □

## 1.6 Paradoxes and Related Techniques

### 1.6.1 Russell's Paradox

Here's a famous version of this paradox:

> There is a (male) barber $b$ in a City who is shaving each (and only) man in the City who is not shaving himself.

Apparently, one might think that this is a possible scenario. In formulas, we could write:

$$\forall m \in \mathsf{City}.\, b \ \mathsf{shaves}\ m \iff \neg(m \ \mathsf{shaves}\ m)$$

But if this is true for *all* men $m$, we can take $m = b$ and have

$$b \ \mathsf{shaves}\ b \iff \neg(b \ \mathsf{shaves}\ b)$$

which is clearly false. That is, we are unable to answer "does the barber shave himself?".

Russell used a similar argument to find a contradiction to naïve set theory. Assume there is a set $X = \{x|p(x)\}$ for each predicate $p$ we can think of. We clearly must have

$$\forall y.\, \big(y \in X \iff p(y)\big)$$

How can we make this resemble the paradox seen before? We want $X$ to play the rôle of the barber. So, $y$ must play the man $m$, and shaves relation must be $\in$ (the membership relation). Then $p(y)$ becomes $y \notin y$. So, the above becomes

$$\forall y.\, \big(y \in \{x|x \notin x\} \iff (y \notin y)\big)$$

which is indeed a contradiction, since if $X = \{x|x \notin x\}$, we now have (choosing $y = X$, as we did before for $m = b$)

$$X \in X \iff X \notin X$$

Russell used this argument to show that the set $X$ above actually must regarded as non well-defined, so to avoid the logical fallacy. The same argument however can be used to prove a number of interesting facts.

### 1.6.2 Diagonalisation

**Theorem 23** (Cantor)**.** *There is no bijection between a set $A$ and its parts $\mathcal{P}(A)$.*

*Proof.* By contradiction, assume $f \in (A \leftrightarrow \mathcal{P}(A))$. We now proceed as for Russell's paradox. Let

$$X = \{x \in A | x \notin f(x)\}$$

Clearly, $X \in \mathcal{P}(A)$, so $f^{-1}(X) \in A$. We now have,

$$f^{-1}(X) \in X \iff f^{-1}(X) \notin f(f^{-1}(X)) \iff f^{-1}(X) \notin X$$

which is a contradiction. □

This kind of argument is also known as a *diagonalisation* argument. This is because the set X is constructed by looking at the *diagonal* of this matrix:

| | $x$ | $y$ | $z$ | ... | (all the elements of $A$) |
|---|---|---|---|---|---|
| $f(x)$ | $\underline{yes}$ | $no$ | $no$ | ... | |
| $f(y)$ | $no$ | $\underline{no}$ | $no$ | ... | |
| $f(z)$ | $no$ | $yes$ | $\underline{yes}$ | ... | |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | |

Given $a \in A$, the matrix above has a "yes" at coordinates $f(a), a$ iff $x_j$ belongs to $X_i$ (and a "no" otherwise). How do we build a set $X$ different from all the $f(a)$'s ? We take the diagonal $(yes, no, yes, \ldots)$ and *complement* it: $(no, yes, no, \ldots)$

| | $x$ | $y$ | $z$ | ... | (all the elements of $A$) |
|---|---|---|---|---|---|
| $X$ | $no$ | $yes$ | $no$ | ... | |

So, $X$ is clearly distinct from all the $f(a)$.

**Exercise 24.** *Construct a bijection from $\mathbb{R}$ to the interval $[0, 1)$.*
*(Hint: start from* $\arctan(x)$*)*

**Theorem 25.** *There is no bijection between $\mathbb{N}$ and $\mathbb{R}$.*

*Proof.* By contradiction, there is a bijection $f$ between $\mathbb{N}$ and $[0, 1)$. Every real $x \in [0, 1)$ can be written in a unique way as an infinite sequence of decimal digits

$$x = 0.\, d_0 d_1 d_2 \ldots$$

with $0 \leq d_i \leq 9$, and such that digits $0, \ldots, 8$ occur infinitely often (no periodic 9's). In other words, there is a bijection between $[0, 1)$ and such infinite sequences.

So, for all $n \in N$, we can write $f(n) = 0.d_{n,0}d_{n,1}\ldots$, hence we have a bijection between $\mathbb{N}$ and these infinite sequences.

We proceed by Russell's argument (diagonalisation). We construct a sequence different from all the ones generated by $f(n)$ for all $n \in \mathbb{N}$. We let

$$d_i = \begin{cases} 1 & \text{if } d_{i,i} = 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that this is indeed a legal sequence (each digit in the $0 \ldots 9$ range, no periodic 9's). Hence, there is no $n$ such that $f(n) = 0.\, d_0 d_1 d_2 \ldots$, contradicting $f$ being a bijection. □

Another example of the same technique:

**Theorem 26.** *There is no bijection $f$ between $\mathbb{N}$ and $(\mathbb{N} \to \mathbb{N})$.*

*Proof.* By contradiction, take $f$. Define $g(n) = f(n)(n) + 1$. Since $f$ is a bijection, and $g$ a function in its range, for some $i \in \mathbb{N}$ we must have $g = f(i)$. But then $f(i)(i) = g(i) = f(i)(i) + 1$. □

Actually, the above proof proved a slightly more general fact: we can extend the theorem to a surjective $f$. Also, we can use partial functions as $\mathsf{ran}(f)$, exploiting $(\mathbb{N} \to \mathbb{N}) \subseteq (\mathbb{N} \rightsquigarrow \mathbb{N})$.

**Theorem 27.** *There is no surjective function between $\mathbb{N}$ and $(\mathbb{N} \rightsquigarrow \mathbb{N})$.*

*Proof.* Left as an exercise. □

## 1.7 Cardinality Argument for Uncomputability

We can now state a first, strong, computability result.

Namely, we compare the set of functions $(\mathbb{N} \to \mathbb{N})$ with the set of programs in an *unspecified* language. We merely assume the following very reasonable assumptions:

- each program can be written in a file — i.e. it can be represented by a (possibly very long, but finite) string
- each program has an associated semantic partial function, mapping the input (a file) to the output (another file)

**Theorem 28.** *There is a function (from input to output) that can not be computed by a program.*

*Proof.* There is a bijection between files and $\mathbb{N}$ (Ex. 19). So a program just corresponds to a natural in $\mathbb{N}$, while the function mapping input to output can be seen as some partial function in $(\mathbb{N} \rightsquigarrow \mathbb{N})$. Since the mapping from programs to their semantics is in $(\mathbb{N} \to (\mathbb{N} \rightsquigarrow \mathbb{N}))$, by Th .27 it can not be surjective. □

Note that the proof above actually hints to one of these uncomputable functions. Let us forget files, and just assume that programs get some natural as input and can output a natural as output. Similarly, we can identify programs with naturals as well, i.e. we fix some enumeration and use $n$ to denote the $n$-th program. So, we can write $\varphi_x(y)$ for the output of the $x$-th program when run using $y$ as input. Then, the proof suggests this function:

$$f(i) = \varphi_i(i) + 1$$

However, we should be careful here: the function $\varphi_i$ is a *partial* function, and therefore $\varphi_i(i)$ might be undefined. So, we change the above definition of $f$ to:

$$f(i) = \begin{cases} \varphi_i(i) + 1 & \text{if } \varphi_i(i) \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

And this indeed is not a computable function.

**Exercise 29.** *Prove that $f$ is not computable.*

**Exercise 30.** *What happens if we change the 0 in the definition of $f$ to some other natural? Does the uncomputability argument still hold? What if we change it to "undefined", thus defining $f$ to be a partial function?*

# Chapter 2

# The $\lambda$ Calculus

For the full gory details, see the introduction of [Barendregt].

## 2.1 Syntax

**Definition 31** ($\lambda$-terms)**.** *Let* $\mathsf{Var} = \{x_0, x_1, \ldots\}$ *be a denumerable set of variables. The syntax of the* $\lambda$*-terms is*

$$\begin{array}{rcll} M := & x & & \textit{variable (with } x \in \mathsf{Var}\textit{)} \\ | & (M\,M) & & \textit{application} \\ | & \lambda x.\, M & & \textit{abstraction (with } x \in \mathsf{Var}\textit{)} \end{array}$$

*The set of all* $\lambda$*-terms is written as* $\Lambda$*.*

Intuitively, a $\lambda$-term represents a function, e.g. we can write

$$f = \lambda x.\, x^2 + 5$$

instead of

$$\forall x.\, f(x) = x^2 + 5$$

**Note.** While we shall often use an extended syntax in our examples, involving arithmetic operators, naturals, and so on, we do this to guide intuition, only. In the $\lambda$ calculus there is *no other syntax* other than that shown in Def. 31. Later, we shall see how we can express things like 5 and $x^2$ in the calculus.

**Exercise 32.** *Rewrite the definition of* $\Lambda$*, providing a recursive equation of the form* $\Lambda \simeq \cdots$*. Use only the following constructs:* $\mathsf{Var}, \times, \uplus$*.*

As a convention, we write chains of applications such as

$$(((xy)z)w)$$

in the more natural form

$$xyzw$$

**Warning.** Note that applications such as $(x(y(zw)))$ still need all the parentheses, otherwise we have $(x(y(zw))) = xyzw = (((xy)z)w)$. These, in general, are not equal, as we shall prove later.

## 2.2 Curry's Isomorphism

How to express functions with more than one parameter? Instead of taking two arguments $x, y$ and return the result, we instead take only $x$, and *return a function*. This function will take $y$, and return the actual result.

$$\lambda x.\, (\lambda y.\, x^2 + y)$$

Note that this way of expressing binary functions also allows *partial application*: we can just pass the first argument $x$, only, and use the resulting function as we want. For instance, we could use the resulting function on several different $y$'s.

We write $\lambda xy.\, \cdots$ as a shorthand for $\lambda x.\, \lambda y.\, \cdots$.

## 2.3 $\alpha$-conversion

As in computer programs, the name of variables is immaterial, so

$$\lambda x.\, x^2 + 5 = \lambda y.\, y^2 + 5$$

This renaming of variables is known as *$\alpha$-conversion*. Sometimes, we use $=_\alpha$ to denote this renaming, e.g.

$$\lambda x.\, x^2 + 5 =_\alpha \lambda y.\, y^2 + 5$$

Note that, while we can rename variables, we must avoid clashes. For instance,

$$\lambda x.\, \lambda y.\, x + y \neq \lambda x.\, \lambda x.\, x + x$$

In the latter expression, both occurrences of $x$ in $x+x$ are bound by the inner $\lambda x$. This follows the same *static scoping* conventions found in programming languages: each occurrence of a variable is bound by the nearest definition.

You should probably do the following exercises together. See also [Barendregt 2.1.11].

**Exercise 33.** *Define formal rules for variable $\alpha$-conversion.*

The free variables $\mathsf{free}(M)$ of a $\lambda$-term are those not under a $\lambda$ binder.

**Definition 34.** *The* free variables *of a $\lambda$-term $M$ are inductively defined as follows:*

$$\begin{aligned}&\mathsf{free}(x_i) = \{x_i\}\\ &\mathsf{free}(NO) = \mathsf{free}(N) \cup \mathsf{free}(O)\\ &\mathsf{free}(\lambda x_i.N) = \mathsf{free}(N) \setminus \{x_i\}\end{aligned}$$

**Exercise 35.** *Define the result of the substitution $M\{N/x\}$, substituting all the free occurrences of $x$ in $M$ with the term $N$. (Watch out for variable clashes!)*

**Definition 36.** *The result of applying a substitution $M\{N/x\}$ is defined as follows.*

$$\begin{aligned}&x_i\{N/x_i\} = N && \\ &x_i\{N/x_j\} = x_i && \text{when } i \neq j\\ &(MM')\{N/x_i\} = (M\{N/x_i\})(M'\{N/x_i\}) && \\ &(\lambda x_i.\, M)\{N/x_i\} = (\lambda x_i.M) && \\ &(\lambda x_j.\, M)\{N/x_i\} = \lambda x_k.\, (M\{x_k/x_j\}\{N/x_i\}) && \text{when } i \neq j \text{ and } x_k \notin \mathsf{free}(N)\\ & && \text{and } x_k \notin \mathsf{free}(\lambda x_j.\, M)\end{aligned}$$

In the last line we avoid variable clashes. First, we rename $x_j$ to $x_k$, a "fresh" variable, picked so that it does not occur (free) in $N$ and $\lambda x_j.\, M$. Then, we can apply the substitution in the body of the function.

## 2.4 $\beta$ and $\eta$ Rules

**Definition 37** ($\beta$ rule)**.** *Here's the $\beta$ rule, used to compute the result of function application.*

$$(\lambda x.\, M)N = M\{N/x\}$$

Example:

$$(\lambda x.x^2 + x + 1)5 = 5^2 + 5 + 1$$

The meaning is straightforward: we can apply a function $(\lambda x.\, M)$ by taking its body $(M)$ and replacing $x$ with the actual argument $(N)$.

**Definition 38** ($\eta$ rule)**.** *Here's the $\eta$ rule, used to remove redundant $\lambda$'s.*

$$(\lambda x.\, Mx) = M \qquad \text{if } x \notin \mathsf{free}(M)$$

When $x$ is not free in $M$, it is obvious that $(\lambda x.\, Mx)$ denotes the same function as $M$: it just forwards its argument $x$ to $M$.

**Exercise 39.** *Can you state the $\eta$ rule in Java (or another procedural language), at least in some loose form?*

Note that one can apply the $\beta$ and $\eta$ rules even to *subterms* of the $\lambda$-term at hand, e.g.

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

**Exercise 40.** *Use the $\eta$ rule to prove the* **ext** *rule.*

$$Mx = Nx \wedge x \notin \mathsf{free}(MN) \implies M = N \qquad (\mathbf{ext})$$

**Exercise 41.** *Show that the $\eta$ rule is actually equivalent to the* **ext** *rule above.*

Usually, one applies these rules by reading them *left to right*:

$$\begin{array}{ll} (\lambda x.\, M)N \to_\beta M\{N/x\} & \\ (\lambda x.\, Mx) \to_\eta M & \text{if } x \notin \mathsf{free}(M) \end{array}$$

In this case, we speak of $\beta$ and $\eta$ *reduction relations*.

**Definition 42** (Normal form)**.** *Given a reduction relation $\to_R$ (e.g. with $R = \beta$ or $R = \eta$), we say that a term $M$ is in $R$-normal form iff $M \not\to_R$.*

Intuitively, when we start $\beta$-reducing a term $M$, we form a sequence like

$$M \to_\beta M_1 \to_\beta M_2 \to_\beta M_3 \cdots$$

This sequence is called a *reduction*[1]. One of the following might happen:

- The reduction stops: that is, we reach some $M_k$ which is a $\beta$-normal form. Intuitively, this is the result of running $M$. We say that the reduction above *halts*.
- The reduction never stops: that is, it is infinite. So, the reduction is non-halting.

Note that a term might have *many reductions*, since we allow $\to_\beta$ to be applied in any subterm as well.

[1] We follow the terminology of [Barendregt] here. Reductions as the above are also called *runs* for $M$.

**Exercise 43.** *Construct different reductions for*

$$(\lambda x.\, x)((\lambda y.y)5)$$

**Exercise 44.** *Show that* $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ *has no halting reduction.*

**Exercise 45.** *After having done the exercise above, define a* $\lambda$*-term* $M$ *having a single non-halting reduction, where the terms* $M_i$ *are completely distinct.*

Note that a term might have both a halting *and* a non-halting reduction.

**Exercise 46.** *Prove the above using* $(\lambda x.5)\Omega$.

Fortunately, there is a simple strategy that always finds a normal form when there is a halting reduction.

**Definition 47** (Leftmost reduction)**.** *A leftmost* $\beta$*-reduction is a reduction where, at every step,* $\to_\beta$ *is applied as to the left as possible, i.e. to the first occurrence of an applied* $\lambda$ *binder, reading the* $\lambda$*-term left-to-right.*

**Exercise 48.** *Devise a procedure to construct a leftmost* $\beta$*-reduction.*

We shall use the following facts. Alas, we omit these proofs.

**Theorem 49** (Normalization)**.** *The leftmost strategy is normalizing, i.e. it finds a* $\beta$*-normal form whenever there is one.*

[Barendregt 13.2.2 — no proof]

Another very important result, stating that even if there can be many $\beta$-reductions, the result is unique (if it exists).

**Definition 50.** *A relation* $\to$ *is a Church-Rosser relation iff* $\forall M, N_1, N_2$

$$M \to^* N_1 \wedge M \to^* N_2 \implies \exists N.\, N_1 \to^* N \wedge N_2 \to^* N$$

**Theorem 51** (Church-Rosser)**.** *The relation* $\to_\beta$ *is a Church-Rosser relation [Barendregt 3.2.8 — no proof].*
*As a consequence, each* $\lambda$*-term has at most one* $\beta$*-normal form (up-to* $\alpha$*-conversion).*

This also provides a nice link between the equational theory and the $\beta\eta$-reduction relation:

**Theorem 52.** *If* $M =_{\beta\eta} N$ *and* $N$ *is a* $\beta\eta$*-normal form, then* $M \to^*_{\beta\eta} N$.

[Barendregt 3.2.9 — no proof]

## 2.5 Useful Combinators

$$\begin{aligned}\mathbf{I} &= \lambda x.\, x\\ \mathbf{K} &= \lambda xy.\, x\\ \mathbf{S} &= \lambda xyz.\, xz(yz)\\ \mathbf{T} &= \lambda xy.\, x = \mathbf{K}\\ \mathbf{F} &= \lambda xy.\, y\end{aligned}$$

The $\lambda$-terms $\mathbf{T}$ and $\mathbf{F}$ are read "true" and "false".

**Example 53.** *We have the following:*

$$\begin{aligned}\mathbf{KISS} &= ((\mathbf{KI})\mathbf{S})\mathbf{S} = \mathbf{IS} = \mathbf{S}\\ \mathbf{SKK}x &= \mathbf{K}x(\mathbf{K}x) = x = \mathbf{I}x\end{aligned}$$

*so, by the* **ext** *rule*

$$\begin{aligned}\mathbf{SKK} &= \mathbf{I}\\ \mathbf{KI}xy &= \mathbf{I}y = y = \mathbf{F}xy\end{aligned}$$

*so, by the* **ext** *rule*

$$\mathbf{KI}x = \mathbf{F}x$$

*again, by the* **ext** *rule*

$$\mathbf{KI} = \mathbf{F}$$

**Exercise 54.** *Prove that we do not have* $\mathbf{T} =_{\beta\eta} \mathbf{F}$. *See Sol. 195.*

**Exercise 55.** *Define*

$$\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ O = MNO$$

*and check the usual "if-laws" for* $M = \mathbf{T}$ *and* $M = \mathbf{F}$. *This justifies the names for* $\mathbf{T}$ *and* $\mathbf{F}$.

**Exercise 56.** *Define the usual logical operators:* **And**, **Or**, **Not**. *(See Sol. 196)*

**Lemma 57.** *Application is not associative, that is*

$$\neg\forall MNO.\,(MN)O = M(NO)$$

*Proof.* By contradiction,

$$(\mathbf{K}(\mathbf{IT}))\mathbf{F} = \mathbf{IT} = \mathbf{T}$$
$$((\mathbf{KI})\mathbf{T})\mathbf{F} = \mathbf{IF} = \mathbf{F}$$

□

**General Hint.** To prove that some equation do not hold in general under $\beta\eta$, you can show it implies $\mathbf{T} = \mathbf{F}$. To this aim, it is useful to consider simple combinators such as $\mathbf{K}, \mathbf{I}$ first. Also, applying everything to a generic term (to be chosen later) usually helps: for instance, you can proceed like this in the lemma above. First, guess $M = \mathbf{K}$. So, $\mathbf{K}NO = \mathbf{K}(NO)$. Now, the $\mathbf{K}$ on the right hand side expects two arguments, and has only one, so we provide it as a generic term $P$, which we can choose later. We obtain $\mathbf{K}NOP = \mathbf{K}(NO)P$, implying $NP = NO$. Now it is easy to guess $N = \mathbf{I}$, so to obtain $P = O$. Guessing $P, O$ is then made trivial.

**Exercise 58.** *Show that, in general, these laws do not hold*

$$MN = NM$$
$$M(NO) = O(MN)$$
$$M(MO) = MO$$
$$MO = MOO$$
$$MM = M$$
$$MN = \lambda x.\,M(Nx)$$

**Exercise 59.** *Check whether these terms have a $\beta$-normal form*

$$\mathbf{KIK}$$
$$\mathbf{KKI}$$
$$\mathbf{K}(\mathbf{K}(\mathbf{KI}))$$
$$\mathbf{SII}$$
$$\mathbf{SII}(\mathbf{SII})$$
$$\mathbf{KI}\Omega$$
$$(\lambda z.\,(\lambda x.\,xxz)(\lambda x.\,xxz))$$

### 2.5.1 Pairs

Pairs can be encoded as follows:

$$\begin{aligned}\mathbf{Cons} &= \lambda xyc.\, cxy\\ \mathbf{Fst} &= \lambda x.\, x\mathbf{T}\\ \mathbf{Snd} &= \lambda x.\, x\mathbf{F}\end{aligned}$$

**Exercise 60.** *Prove the usual pair laws:*

$$\mathbf{Fst}(\mathbf{Cons}\, M\, N) = M \qquad\qquad \mathbf{Snd}(\mathbf{Cons}\, M\, N) = N$$

**Exercise 61.** *Define $F$ so that (standalone exercises):*

- $F(\mathbf{Cons}\, x\, y) = \mathbf{Cons}\, x\, (\mathbf{Cons}\, y\, x)$
- $F(\mathbf{Cons}\, x\, (\mathbf{Cons}\, y\, z)) = \mathbf{Cons}\, z\, (\mathbf{Cons}\, x\, y)$

## 2.6 Recursive Functions and Fixed Points

Can we build recursive functions? For instance, the factorial function:

$$f = \lambda n.\mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot (f(n-1)) \tag{2.1}$$

Is there some $\lambda$-term $f$ that satisfies the equation above? Of course, the equation itself has $f$ on both sides so it does not define a $\lambda$-term $f$ (unlike e.g. $f = \lambda x.\, x^2 + 5$).

What if we abstract the recursive call?

$$F = \lambda \mathbf{g}.\, \lambda n.\, \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot (\mathbf{g}(n-1)) \tag{2.2}$$

This is now a valid $\lambda$-term, since it is a non-recursive definition. However, we must now force $g$ to act, very roughly, as $f$. A first attempt would be to simply pass a *copy* of $f$ to $f$ itself, as this:

$$f = MM \qquad \text{where} \qquad M = \lambda g.\, \lambda n.\, \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot (g(n-1))$$

This however has a problem: $g$ will be bound to $M$, which is only "half" of $f$. So, the recursive call $g(n-1)$ is actually $M(n-1)$, and that is not $f(n-1)$. However, the latter would be $MM(n-1)$, and we *can* express

this by just writing the recursive call as $gg(n-1)$. So we can *fix*[2] the above definition as follows:

$$f = MM \qquad \text{where} \qquad M = \lambda g.\, \lambda n.\, \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot (gg(n-1))$$

Note that this is a proper definition for a $\lambda$-term $f$.

**Exercise 62.** *Use the above definition of $f$ to compute the factorial of* 3.

**Exercise 63.** *Write a $\lambda$-term for computing $\sum_{i=0}^{n} n^2$.*

It is important to note that the body of *any* recursive function $f$ can be written as in (2.2), that is abstracting all the recursive calls. Writing $F$ for the (abstracted) body, we can see that the key property we are interested in is

$$f = Ff$$

Indeed, by the $\beta$ rule, the above is equivalent to the recursive definition, see e.g. (2.1). So finding such a term $f$ means to find a *fixed point* for $F$.

What if we had a $\lambda$-term $\mathbf{\Theta}$ such that $\mathbf{\Theta}F = F(\mathbf{\Theta}F)$ for *any* $F$? That would be great, because we can use that to express *any* recursive function, just by writing the abstracted body and applying $\mathbf{\Theta}$ to that. Such a $\mathbf{\Theta}$ is called a *fixed point combinator*.

**Exercise 64.** *Write such a $\mathbf{\Theta}$.*
*(Hint. This seems hard, but we know all the tricks now. Start from the equation $\mathbf{\Theta} = \lambda F.\, F(\mathbf{\Theta}F)$.)*
*After you solve this, compare your solution to that in the Appendix, Sol. 194.*

**Exercise 65.** *Check whether these terms have a $\beta$-normal form*

$\mathbf{\Theta}$
$\mathbf{KI\Theta}$
$\mathbf{K\Theta I}$
$\mathbf{\Theta I}$
$\mathbf{\Theta K}$
$\mathbf{\Theta(KI)}$

[2] Oh, the irony...

## 2.7 Church's Numerals

The $\lambda$ calculus does not have any numbers in its syntax. In spite of this, it is possible to *encode* naturals into $\lambda$-terms, and compute with them. That is, we shall pick an infinite sequence of (closed) $\lambda$-terms, and use them to denote naturals in the $\lambda$ calculus. We shall name these $\lambda$-terms the *numerals*.

There are several ways to encode naturals; we shall use a simple way found by Church. Recall the structure of naturals, seen as terms in first-order logic:

$$z, s(z), s(s(z)), s(s(s(z))), \ldots$$

where $z$ is a constant representing zero, and $s$ is the successor function. We just convert that notation to the $\lambda$ calculus by abstracting over $s$ and $z$:

$$\lambda sz.\, z, \lambda sz.\, sz, \lambda sz.\, s(sz), \lambda sz.\, s(s(sz)), \ldots$$

We shall write the above sequence as $\ulcorner 0 \urcorner$,$\ulcorner 1 \urcorner$,$\ulcorner 2 \urcorner$, and so on.

**Definition 66.** *The sequence of Church numerals is inductively defined as follows. Let $s$ and $z$ be variables*[3]*.*

$$\begin{aligned} M_0 &= z \\ M_{n+1} &= s\, M_n \\ \ulcorner n \urcorner &= \lambda sz.\, M_n \end{aligned}$$

Another way to define the same numerals is through the function composition operator:

$$\circ = \lambda fgx.\, f(gx)$$

Then, we can define a "zero" and "successor" $\lambda$-terms as follows

$$\begin{aligned} \mathbf{0} &= \lambda f.\, \mathbf{I} \\ \mathbf{Succ} &= \lambda nf.\, \circ f(nf) \end{aligned}$$

The sequence of Church's numerals indeed satisfies the following.

$$\begin{aligned} \ulcorner 0 \urcorner &= \mathbf{0} \\ \ulcorner n+1 \urcorner &= \mathbf{Succ} \ulcorner n \urcorner \end{aligned}$$

We can check a numeral against zero using the following combinator:

$$\mathbf{IsZero} = \lambda n.\, n(\mathbf{KF})\mathbf{T}$$

[3] Let us pick $s = x_0$ and $z = x_1$.

**Exercise 67.** *Check that* $\mathbf{IsZero}\ulcorner 0\urcorner = \mathbf{T}$ *and* $\mathbf{IsZero}\ulcorner n+1\urcorner = \mathbf{F}$.

The predecessor function. Note that we let $\mathbf{Pred}\,\mathbf{0} = \mathbf{0}$.

$$\begin{aligned}\mathbf{Pred} &= \lambda n.\,\mathbf{Snd}(nM(\mathbf{Cons}\,\mathbf{F0}))\\ M &= \lambda p.\,\mathbf{Cons}\,\mathbf{T}(\mathbf{Fst}\,p\,(\mathbf{Succ}(\mathbf{Snd}\,p))\,\mathbf{0})\end{aligned}$$

**Exercise 68.** *Check that* **Pred** *is correct.*

**Exercise 69.** *Define the usual arithmetic operators and comparisons. Also see the solution in the appendix (Sol. 196).*

**Exercise 70.** *Assume lists of positive naturals such as* $[1,2,3]$ *are encoded as* $\mathbf{Cons}\ulcorner 1\urcorner(\mathbf{Cons}\ulcorner 2\urcorner(\mathbf{Cons}\ulcorner 3\urcorner(\mathbf{Cons}\,\mathbf{0},\Omega)))$, *using* $\mathbf{0}$ *to mark the end of the list. Write the following functions:*

- **Length** *returning the length of a list*
- **Even** *removing from the input list all odd numbers*
- **Append** *appending two lists*
- **Reverse** *reversing a list*
- **Sort** *sorting the list (use e.g. merge-sort)*

*See Sol. 197.*

**Exercise 71.** *Find an encoding for lists of arbitrary (opaque) data, and adapt the functions seen above. What about binary trees?*

## 2.8 λ-definable Functions

We now aim to define partial functions $f \in (\mathbb{N} \rightsquigarrow \mathbb{N})$ through λ-terms. A first attempt would be:

**Definition 72** (naïve λ-definability)**.** *Given a partial function* $f \in (\mathbb{N} \rightsquigarrow \mathbb{N})$, *we say that a closed* λ*-term* $M$ defines $f$ *iff for all* $n \in \mathbb{N}$

$$\begin{array}{ll} M\ulcorner n\urcorner = \ulcorner f(n)\urcorner & \text{if } n \in \mathsf{dom}(f)\\ M\ulcorner n\urcorner \text{ has no } \beta\text{-normal form} & \text{otherwise}\end{array}$$

While we could work with this definition, the following exercise shows a technical problem.

**Exercise 73.** *Show that it is possible that a λ-term $M$ has no β-normal form, while $MN$* has *a β-normal form.*
*(Hint: take $M$ to be one of the combinators we saw before.)*

So, if we for instance want to *compose* two partial functions $f$ and $g$, we cannot simple use $M = \lambda x.\, F(Gx)$ for the composition. This is because, when e.g. $F = \mathbf{K}^{\ulcorner}3^{\urcorner}$, and $G$ does not terminate ($g = \emptyset$), the λ-term $M$ reduces to $\lambda x.\, ^{\ulcorner}3^{\urcorner}$ while $f \circ g = \emptyset$. In other words, $M$ fails to force the *evaluation* of $Gx$. [4]

We need some way to force the evaluation of a λ-term $G$, and still be able to discard its result if we want to. A reasonably simple way to do this is through a *jamming factor*, i.e. a term $J$ such that, whenever $G$ terminates $J$ behaves as the identity $\mathbf{I}$, and otherwise when $G$ does not terminate $J$ "jams" the whole computation. By jamming we mean that, even if $J$ is applied to further λ-terms, it still does *not* produce a normal form. In fact, this works around the problem spotted in Ex. 73. Following [Barendregt], we want a *solvable* $J$.

**Definition 74** (solvability)**.** *A closed λ-term $M$ is solvable if there are some $N_1, \ldots, N_k$, with $k \geq 0$, such that $MN_1 \cdots N_k = \mathbf{I}$.*

[Barendregt 8.3.1]

**Exercise 75.** *Show that if $M$ is unsolvable, then $MN$ is also unsolvable, for any $N$.*

**Exercise 76.** *Show that Church's numerals can be* uniformly solved *by finding $M, N$ such that $\forall n \in \mathbb{N}.\, ^{\ulcorner}n^{\urcorner}MN = \mathbf{I}$.*

**Theorem 77.** *Any closed β-normal form is solvable.*

*Proof.* We leave this as an exercise.
Hint: first, show that the normal form has the form

$$\lambda x_1 \ldots x_n.\, x_i M_1 \ldots M_k$$

for some $i \in \{1..n\}$. (This is called a *head normal form*) □

We can then define λ-definability in a more convenient way:

[4] Indeed, it is inconvenient to lack a simple way to force the complete evaluation of a term in the λ calculus. This is the price to pay to obtain a very minimalistic semantics (the βη rules). There are however many ways to address this issue, as we shall see.

**Definition 78** (λ-definability)**.** *Given a partial function $f \in (\mathbb{N} \rightsquigarrow \mathbb{N})$, we say that a closed λ-term $M$* defines *$f$ iff for all $n \in \mathbb{N}$*

$$\begin{array}{ll} M\ulcorner n\urcorner = \ulcorner f(n)\urcorner & \text{if } n \in \mathsf{dom}(f) \\ M\ulcorner n\urcorner \text{ unsolvable} & \text{otherwise} \end{array}$$

*A partial function $f$ is λ-definable iff it is defined by some $M$. This definition is naturally extended to partial functions $\mathbb{N}^k \rightsquigarrow \mathbb{N}$. Given $A \subset \mathbb{N}$, $A$ is λ-defined iff $\chi_A$ is such.*

In more casual terms, such an $M$ either terminates, returning a numeral, or "jams" the computation.

**Exercise 79.** *Show that if $f, g$ are partial λ-definable functions, then their composition $f \circ g$ is such.*
*Hint: exploit Ex. 76, 75.*

**Definition 80.** *A set $A \subseteq \mathbb{N}$ is λ-defined by $F$ iff*

$$\begin{aligned} n \in A &\implies F\ulcorner n\urcorner = \mathbf{T} \\ n \notin A &\implies F\ulcorner n\urcorner = \mathbf{F} \end{aligned}$$

*A set $\mathcal{L} \subseteq \Lambda$ is λ-defined by $F$ iff $\{\#M | M \in \mathcal{L}\}$ is such.*

**Exercise 81.** *Change* **T** *with* 1 *and* **F** *with* 0 *in the definition above, and prove this alternative notion of λ-definability to be equivalent.*

**Exercise 82.** *Show that finite subsets of $\mathbb{N}$ are λ-definable.*

**Lemma 83.** *λ-definable sets are closed under*

- *union ($\cup$)*
- *complement ($\setminus$)*
- *intersection ($\cap$)*

*Proof.* Left as an exercise. □

**Lemma 84.** *Let $f$ be a total injective λ-definable function. Let $A \subseteq \mathbb{N}$, and let $B = \{f(n) | n \in A\}$. If $B$ is λ-definable, then $A$ is such.*

*Proof.* Let $f, B$ be λ-defined by $F, M_B$. Then let $M_A = \lambda n.\, M_B(Fn)$. Note that $M_A\ulcorner n\urcorner = M_B\ulcorner f(n)\urcorner$. If $n \in A$, then the above evaluates to **T**. If $n \notin A$, then $f(n) \notin B$ since $f$ is injective, and $M_B\ulcorner f(n)\urcorner$ evaluates to **F**. □

## 2.9 Classical Computability Results in the λ calculus

Recall the cardinality argument: $\Lambda$ is a denumerable set, while $\mathbb{N} \to \mathbb{N}$ is larger. So, we expect to find some function which is not $\lambda$-definable. We can indeed define it through a diagonalisation process.

First, we need to enumerate the $\lambda$-terms. To this aim, recall the recursive definition $\Lambda \simeq \mathsf{Var} \uplus (\Lambda \times \Lambda \uplus \mathsf{Var} \times \Lambda)$. We define a bijection between $\Lambda$ and $\mathbb{N}$; we write the natural corresponding to $M$ as $\#M$.

**Definition 85.** *We define the bijection # as follows.*

$$(\#-) \in (\Lambda \leftrightarrow \mathbb{N})$$
$$\#M = \begin{cases} \mathsf{inL}(i) & \text{if } M = x_i \\ \mathsf{inR}(\mathsf{inL}(\mathsf{encode}_\times(\#N, \#O))) & \text{if } M = NO \\ \mathsf{inR}(\mathsf{inR}(\mathsf{encode}_\times(i, \#N))) & \text{if } M = \lambda x_i.\, N \end{cases}$$

We can then represent the natural $\#M$ in the calculus in the usual way.

**Definition 86.** *The function $\ulcorner M \urcorner$ is defined as follows.*

$$\ulcorner - \urcorner \in (\Lambda \to \Lambda^0)$$
$$\ulcorner M \urcorner = \ulcorner \#M \urcorner$$

We can now define a non-computable function, following the diagonalisation argument. We define $f \in (\mathbb{N} \to \mathbb{N})$ as follows

$$f(n) = \begin{cases} 1 & \text{if } M\ulcorner M \urcorner \text{ has a } \beta\text{-normal form, where } n = \#M \\ 0 & \text{otherwise} \end{cases}$$

Note that this is a *total* function, by construction. Also note we are applying a term $M$ to its own numeral index $\ulcorner M \urcorner$. Suppose that the function above is $\lambda$-defined by $F$. Then, define

$$M = \lambda x.\, \mathbf{Eq}\ulcorner 0 \urcorner (Fx)\mathbf{I}\Omega$$

We now consider $f(\#M)$: by definition of $f$, this is either 1 or 0. If $f(\#M)$ were equal to 1, then $M\ulcorner M \urcorner$ would have a normal form, but then

$$M\ulcorner M \urcorner = \mathbf{Eq}\ulcorner 0 \urcorner (F\ulcorner M \urcorner)\mathbf{I}\Omega = \mathbf{Eq}\ulcorner 0 \urcorner\ulcorner 1 \urcorner\mathbf{I}\Omega = \mathbf{FI}\Omega = \Omega$$

which has *not* a normal form — a contradiction. We must conclude that $f(\#M)$ is equal to 0, and that $M\ulcorner M\urcorner$ has no normal form, but then

$$M\ulcorner M\urcorner = \mathbf{Eq}\ulcorner 0\urcorner(F\ulcorner M\urcorner)\mathbf{I}\Omega = \mathbf{Eq}\ulcorner 0\urcorner\ulcorner 0\urcorner\mathbf{I}\Omega = \mathbf{TI}\Omega = \mathbf{I}$$

*has* a normal form — another contradiction.

Hence, such a λ-term $F$ can not exist, i.e. the function $f$ can not be λ-defined.

**Lemma 87.** *The function $f$ defined above is not λ-definable.*

**Exercise 88.** *Compare this result with Exercise 29. You should find the proof to be similar.*

**Nota Bene.** Having $M =_{\beta\eta} N$ does *not* imply that $\#M = \#N$. That is, even if two programs are semantically equivalent, their *source code* may be different!

**Exercise 89.** *Find some closed $M, N$ such that $M =_{\beta\eta} N$ but $\#M \neq \#N$.*

**Nota Bene.** Having $M =_\alpha N$ does *not* imply that $\#M = \#N$. That is, even if two programs only differ because of α-conversion (i.e. choice of variable names), their index is different!

**Exercise 90.** *Show that $\#(\lambda x_0.\, x_0) \neq \#(\lambda x_1.\, x_1)$.*

We can now define one of the most famous sets in computability.

**Definition 91.** $\mathsf{K} = \{\#M | M\ulcorner M\urcorner$ *has a β-normal form*$\}$

Note that $\mathsf{K} \subseteq \mathbb{N}$.

**Lemma 92.** $\mathsf{K}$ *is not λ-definable*

*Proof.* By contradiction, if $\mathsf{K}$ were λ-definable by e.g. $G$, then we could λ-define the function $f$ shown above using this $F$:

$$F = \lambda x.\, G\, x\ulcorner 1\urcorner\ulcorner 0\urcorner$$

Indeed, $f$ is $\chi_{\mathsf{K}}$, the characteristic function of the set $\mathsf{K}$. □

It is often useful to define programs that manipulate the source code of other programs. In the real world, the most common case of these programs are the *optimizers*. An optimizer $O$ could take a program code $\ulcorner M\urcorner$ and output the code of another program $\ulcorner N\urcorner$ which is a more efficient version

of the first one. This usually involves searching the code of $M$ for some known-inefficient patterns[5] and replace them with efficient equivalent code. Other typical cases are compilers and interpreters: they obviously need to examine the structure of the code of a program.

We want to show that these syntactic transformations can be done in the $\lambda$-calculus. To manipulate $\ulcorner M \urcorner$ we basically need to compute the $\mathsf{encode}$ functions and their inverses.

**Exercise 93.** *Show that* $\mathsf{encode}_\times$ *and* $\mathsf{encode}_\uplus$ *can be* $\lambda$*-defined, as well as their inverses. Construct the following functions:*

- $\mathbf{Pair}, \mathbf{Proj1}, \mathbf{Proj2}$ *such that*
  - $\mathbf{Pair}^{\ulcorner} n^{\urcorner\urcorner\ulcorner} m^{\urcorner}$, $\mathbf{Proj1}^{\ulcorner} n^{\urcorner}$ *and* $\mathbf{Proj2}^{\ulcorner} n^{\urcorner}$ *return numerals*
  - $\mathbf{Proj1}(\mathbf{Pair}^{\ulcorner} n^{\urcorner\urcorner\ulcorner} m^{\urcorner}) = \ulcorner n \urcorner$
  - $\mathbf{Proj2}(\mathbf{Pair}^{\ulcorner} n^{\urcorner\urcorner\ulcorner} m^{\urcorner}) = \ulcorner m \urcorner$
  - $\mathbf{Pair}(\mathbf{Proj1}^{\ulcorner} n^{\urcorner})(\mathbf{Proj2}^{\ulcorner} n^{\urcorner}) = \ulcorner n \urcorner$
- $\mathbf{InL}, \mathbf{InR}, \mathbf{Case}$ *such that*
  - $\mathbf{InL}^{\ulcorner} n^{\urcorner}$ *and* $\mathbf{InR}^{\ulcorner} n^{\urcorner}$ *return numerals*
  - $\mathbf{Case}(\mathbf{InL}^{\ulcorner} m^{\urcorner}) M N = M \ulcorner m \urcorner$
  - $\mathbf{Case}(\mathbf{InR}^{\ulcorner} n^{\urcorner}) M N = N \ulcorner n \urcorner$
  - $\mathbf{Case}\, \ulcorner n \urcorner\, \mathbf{InL}\, \mathbf{InR} = \ulcorner n \urcorner$

*Also see Solution 196 in the appendix.*

### 2.9.1 Parameter Lemma

Now we tackle an useful, yet quite simple, code manipulation:

**Lemma 94** (Parameter lemma, s-m-n lemma — simple version)**.** *There exists* $\mathbf{App} \in \Lambda^0$ *such that,* $\forall M, N$

$$\mathbf{App} \ulcorner M \urcorner \ulcorner N \urcorner = \ulcorner MN \urcorner$$

**Exercise 95.** *Prove it.*

[5] We mean *syntactic* patterns, e.g. replacing all the occurrences of $\mathbf{I}N$ with $N$. For this we only need to scan the program code. Instead, rewriting *semantic* patterns, e.g. replacing $ON$ with $M$ whenever $O =_{\beta\eta} \mathbf{I}$ is much harder – in fact it is not computable, as we shall see.

**Exercise 96.** *Define a $G \in \Lambda^0$ such that:*

- $G\ulcorner M\urcorner = \ulcorner MM\urcorner$
- $G\ulcorner M\urcorner = \ulcorner MMM\urcorner$
- $G\ulcorner M\urcorner = \ulcorner M(MM)\urcorner$
- $G\ulcorner MN\urcorner = \ulcorner NM\urcorner$
- $G\ulcorner \lambda x.\, M\urcorner = \ulcorner M\urcorner$
- $G\ulcorner \lambda x.\, \lambda y.\, M\urcorner = \ulcorner \lambda y.\, \lambda x.\, M\urcorner$
- $G\ulcorner \mathbf{I}M\urcorner = \ulcorner M\urcorner$ *and* $G\ulcorner \mathbf{K}M\urcorner = \ulcorner \mathbf{I}\urcorner$
- $G\ulcorner \lambda x_i.\, M\urcorner = \ulcorner \lambda x_{i+1}.\, M\urcorner$
- $G\ulcorner M\urcorner = \ulcorner N\urcorner$ *where $N$ is obtained from $M$ replacing every variable $x_i$ with $x_{i+1}$*
- $G\ulcorner M\urcorner = \ulcorner M\{\mathbf{I}/x_0\}\urcorner$ *(this does not require $\alpha$-conversion)*

*See Solution 198 in the Appendix.*

### 2.9.2 Padding Lemma

Intuitively, many different programs actually have the same semantics. Indeed, recall Ex. 89. We can actually automatically generate an infinite number of equivalent programs.

**Lemma 97** (Padding lemma)**.** *Given $M$, there exists $N$ such that $M =_{\beta\eta} N$ and $\#N > \#M$. Such an $N$ can be effectively computed by a $\lambda$-term* **Pad** *such that*

$$\mathbf{Pad}\ulcorner M\urcorner = \ulcorner N\urcorner$$

*Proof.* Left as an exercise. See Solution 200. □

Using **Pad** we can generate an infinite number of programs equivalent to $M$ by just using $\ulcorner n\urcorner\mathbf{Pad}\ulcorner M\urcorner$, which generates a distinct program for each $n \in \mathbb{N}$.

### 2.9.3 Universal Program

Another useful construction is a "self-interpreter", i.e. a $\lambda$-term $\mathbf{E}$ ("evaluate") that, given the code $\ulcorner M \urcorner$, can run it and behave as $M$. This $\mathbf{E}$ is said to be a universal program, since it can be used to compute anything that can be computed in $\lambda$-calculus. It is, in a sense, "the most general program".

Note that we only allow *closed* $M$ here[6].

**Lemma 98** (Self-interpreter)**.** *There exists* $\mathbf{E} \in \Lambda^0$ *such that*

$$\mathbf{E} \ulcorner M \urcorner = M$$

*for all* closed $M$*.*

*Proof.* We proceed by defining two auxiliary operators.

- $\mathbf{E}' \ulcorner M \urcorner \rho = M'$ where $M'$ is $M$ with each free variable $x_i$ replaced by $\rho \ulcorner i \urcorner$. Here, the rôle of the parameter $\rho$ is to define the meaning of the free variables in $M$, defining the value of $x_i$ as $\rho \ulcorner i \urcorner$. This $\rho$ is called the *environment* function.
- $\mathbf{Upd}\, \rho \ulcorner i \urcorner\, a = \rho'$ where $\rho'$ is the "updated" environment, obtained from $\rho$ by replacing the value of $x_i$ with the new value $a$. Formally,

  $$(\mathbf{Upd}\, \rho \ulcorner i \urcorner\, a) \ulcorner i \urcorner = a$$
  $$(\mathbf{Upd}\, \rho \ulcorner i \urcorner\, a) \ulcorner j \urcorner = \rho \ulcorner j \urcorner \quad \text{where } i \neq j$$

  These equations are satisfied by

  $$\mathbf{Upd} = \lambda \rho i a j.\, \mathbf{Eq}\, j\, i\, a\, (\rho\, j)$$

  We can now formalize the $\mathbf{E}'$ function:

  $$\mathbf{E}' \ulcorner x_i \urcorner \rho = \rho \ulcorner i \urcorner$$
  $$\mathbf{E}' \ulcorner M N \urcorner \rho = \mathbf{E}' \ulcorner M \urcorner \rho (\mathbf{E}' \ulcorner N \urcorner \rho)$$
  $$\mathbf{E}' \ulcorner \lambda x_i.\, M \urcorner \rho = \lambda a.\, \mathbf{E}' \ulcorner M \urcorner (\mathbf{Upd}\, \rho \ulcorner i \urcorner\, a)$$

These equations are satisfied by:

$$\mathbf{E}' = \mathbf{\Theta}\Big(\lambda f m \rho.\, \mathbf{Case}\, m\, (\lambda i.\, \rho i)(\lambda m'.\, \mathbf{Case}\, m'\, A\, B)\Big)$$
$$A = \lambda m''.\, f\, (\mathbf{Proj1}\, m'')\, \rho\, (f\, (\mathbf{Proj2}\, m'')\, \rho)$$
$$B = \lambda m''.\, \lambda a.\, f\, (\mathbf{Proj2}\, m'')\, (\mathbf{Upd}\, \rho\, (\mathbf{Proj1}\, m'')\, a)$$

[6] Unfortunately, extending $\mathbf{E}$ to all the open terms is not possible, since $\mathsf{free}(\mathbf{E})$ would need to be the whole $\mathsf{Var}$ in that case.

After defining $\mathbf{E}'$, we can just let $\mathbf{E} = \lambda m.\, \mathbf{E}'\, m\, \Omega$. Here we use $\Omega$ as the initial environment. Indeed, when $M \in \Lambda^0$, the $\lambda$-term $M$ has no free variables, so the initial environment will never be invoked by $\mathbf{E}'$. That is, we only invoke the environment $\rho$ on variables that have been defined through **Upd**. □

**Exercise 99.** *Check the correctness of* $\mathbf{E}$ *in some concrete (small) cases. For instance check that* $\mathbf{E}\ulcorner\mathbf{I}\urcorner = \mathbf{I}$ *and* $\mathbf{E}\ulcorner\mathbf{K}\urcorner = \mathbf{K}$.

### 2.9.4 Kleene's Fixed Point Theorem

This is also known as the *second recursion theorem*. We establish some preliminary result.

**Lemma 100.** *There exists* $\mathbf{Num} \in \Lambda^0$ *such that for all* $n \in \mathbb{N}$

$$\mathbf{Num}\ulcorner\ulcorner n\urcorner\urcorner = \ulcorner\ulcorner n\urcorner\urcorner$$

*Proof.*

$$\begin{aligned}
&\mathbf{Num} = \lambda y.\, \mathbf{InR}\,(\mathbf{InR}\,(\mathbf{Pair}\,\ulcorner 0\urcorner\, A))\\
&A = \mathbf{InR}\,(\mathbf{InR}\,(\mathbf{Pair}\,\ulcorner 1\urcorner\, B))\\
&B = y\,(\mathbf{App}\,(\mathbf{InL}\ulcorner 0\urcorner))(\mathbf{InL}\ulcorner 1\urcorner)
\end{aligned}$$

Note that $B = n\,(\mathbf{App}\ulcorner x_0\urcorner)\ulcorner x_1\urcorner = \ulcorner x_0(x_0(\cdots(x_0\, x_1)))\urcorner$, where $x_0$ is applied $n$ times, when $y = \ulcorner n\urcorner$. Then, $A = \ulcorner\lambda x_1.\, x_0(x_0(\cdots(x_0\, x_1)))\urcorner$, and so $\mathbf{Num}\ulcorner n\urcorner = \ulcorner\lambda x_0.\, \lambda x_1.\, x_0(x_0(\cdots(x_0\, x_1)))\urcorner = \ulcorner\ulcorner n\urcorner\urcorner$. □

Note that $\mathbf{Num}\ulcorner M\urcorner = \mathbf{Num}\ulcorner\#M\urcorner = \ulcorner\ulcorner\#M\urcorner\urcorner = \ulcorner\ulcorner M\urcorner\urcorner$.

**Theorem 101** (Kleene's fixed point)**.** *For all* $F \in \Lambda$*, there is* $X \in \Lambda$ *such that*

$$F\ulcorner X\urcorner = X$$

*Proof.* A "standard" fixed point such that $FX = X$ could be constructed using

$$X = MM \qquad M = \lambda w.\, F(ww)$$

(compare it with the definition of $\mathbf{\Theta}$). We adapt this to obtain:

$$X = M\ulcorner M\urcorner \qquad M = \lambda w.\, F(\mathbf{App}\, w(\mathbf{Num}\, w))$$

Hence,

$$\begin{aligned} X &= M^{\ulcorner}M^{\urcorner} \\ &= F(\mathbf{App}^{\ulcorner}M^{\urcorner}(\mathbf{Num}^{\ulcorner}M^{\urcorner})) \\ &= F(\mathbf{App}^{\ulcorner}M^{\urcorner\ulcorner}M^{\urcorner\urcorner}) \\ &= F^{\ulcorner}M^{\ulcorner}M^{\urcorner\urcorner} \\ &= F^{\ulcorner}X^{\urcorner} \end{aligned}$$

□

Note the difference between Th. 101 and Lemma 98. Roughly, the former says that $\forall F.\, \exists X.\, F^{\ulcorner}X^{\urcorner} = X$. The latter instead says that $\exists F.\, \forall X.\, F^{\ulcorner}X^{\urcorner} = X$.

**Exercise 102.** *Show whether it is possible to construct a program $P \in \Lambda^0$ such that... (each point below is a standalone exercise)*

- $PM = {}^{\ulcorner}P^{\urcorner}$ *for all* $M$
- $P^{\ulcorner}P^{\urcorner} = {}^{\ulcorner}1^{\urcorner}$ *and* $P^{\ulcorner}n^{\urcorner} = \mathbf{0}$ *otherwise*
- $P\mathbf{0} = {}^{\ulcorner}P^{\urcorner}$ *and* $P^{\ulcorner}n^{\urcorner} = {}^{\ulcorner}\mathbf{E}^{\urcorner}$ *otherwise*
- $P^{\ulcorner}n^{\urcorner} = {}^{\ulcorner}n + 2^{\urcorner}$
- $P^{\ulcorner}n^{\urcorner} = P^{\ulcorner}n + 1^{\urcorner}$
- $P^{\ulcorner}n^{\urcorner} = P^{\ulcorner}n + \#P^{\urcorner}$
- $P^{\ulcorner}n^{\urcorner} = \mathbf{Succ}(P^{\ulcorner}n^{\urcorner})$
- $P^{\ulcorner}n^{\urcorner} = {}^{\ulcorner}P(P^{\ulcorner}n^{\urcorner})^{\urcorner}$
- $\#P = \#P + 1$
- $\#P = \#(P^{\ulcorner}P^{\urcorner})$
- $\#P = \#\mathbf{K}$

**Exercise 103.** *Show that there exists a $G \in \Lambda^0$ such that for all $F \in \Lambda^0$*

$$F^{\ulcorner}G^{\ulcorner}F^{\urcorner\urcorner} = G^{\ulcorner}F^{\urcorner}$$

The set $\mathsf{K}_0$ is related to the set $\mathsf{K}$. As for $\mathsf{K}$, this set is not $\lambda$-definable.

**Definition 104.** $\mathsf{K}_0 = \{\#M |\ M\mathbf{0}$ *has a $\beta$-normal form*$\}$

**Exercise 105.** *Prove that $\mathsf{K}_0$ is not $\lambda$-definable. (See sol. 199)*

### 2.9.5 Rice's Theorem

This is one of the most important results in computability, since it shows that a large class of interesting problems are non-$\lambda$-definable.

**Definition 106.** *A set $\mathcal{L} \subseteq \Lambda$ is closed under $\beta\eta$ iff $\forall M, N$*

$$M \in \mathcal{L} \wedge M =_{\beta\eta} N \implies N \in \mathcal{L}$$

**Theorem 107** (Rice's theorem)**.** *Let $\mathcal{L} \subseteq \Lambda$ be closed under $\beta\eta$, and $\lambda$-defined by $F$. Then, $\mathcal{L}$ is trivial, i.e. either empty or equal to $\Lambda$.*

*Proof.* By contradiction, assume $\mathcal{L}$ non trivial, so $M_1 \in \mathcal{L}$ and $M_0 \notin \mathcal{L}$ for some $M_1$ and $M_0$. Then, by Kleene's fixpoint theorem, for some $G$

$$G = F\ulcorner G \urcorner M_0 M_1$$

Then, if $G \in \mathcal{L}$,

$$G = F\ulcorner G \urcorner M_0 M_1 = \mathbf{T} M_0 M_1 = M_0 \notin \mathcal{L}$$

which is a contradiction since $\mathcal{L}$ is closed under $\beta\eta$. Otherwise, if $G \notin \mathcal{L}$,

$$G = F\ulcorner G \urcorner M_0 M_1 = \mathbf{F} M_0 M_1 = M_1 \in \mathcal{L}$$

which is a contradiction, again. □

[see also Barendregt 6.5.9 to 6.6]

Rice's theorem has a large number of consequences, stating that no non-trivial property about the semantics of the code can be inferred from the code itself.

**Exercise 108.** *Which ones of these sets are $\lambda$-definable? Justify your answer.*

- $\{\#M | M$ *$\lambda$-defines $f\}$ where $f$ is some function in* $\mathbb{N} \to \mathbb{N}$
- $\{\#M | M\ulcorner 5 \urcorner$ *evaluates to an even numeral*$\}$
- $\{\#M | M\ulcorner 0 \urcorner$ *has a normal form*$\}$
- $\{\#M | M\ulcorner 0 \urcorner$ *has not a normal form*$\}$
- $\{\#M | M$ *is solvable*$\}$
- $\{\#M | \#(MM)$ *is even*$\}$

- $\{\#M | M$ *has at most three* $\lambda$*'s inside itself*$\}$
- $\{\#M | M\ulcorner n\urcorner$ *has a normal form for a finite number of* $n\}$
- $\{\#M | M\ulcorner n\urcorner$ *has a normal form for a infinite number of* $n\}$
- $\{2 \cdot \#M + 1 | M\ulcorner 0\urcorner = \mathbf{I}\}$
- $\{f(\#M) | M\ulcorner 0\urcorner = \mathbf{I}\}$ *where* $f(n) = 3$ *if* $n$ *is even; otherwise* $f(n) = 2$
- $\{2 \cdot \#M + 1 | M\ulcorner M\urcorner = \mathbf{I}\}$

## 2.10 Other Facts

### 2.10.1 Step-by-step Interpreter

Here we build a more "traditional" interpreter, i.e. another version of **E**. This intrepreter evaluates the $\lambda$-term step-by-step, computing the result of repeatedly applying the $\beta$ rule (in a leftmost fashion). This allows us to specify a "timeout" parameter, if we want to. That is, we can ask the interpreter to run a program $M$ for $n$ steps, and tell us whether $M$ reached normal form within that time constraint.

**Exercise 109.** *Define* **Subst** *such that*

$$\mathbf{Subst}\ulcorner x\urcorner\ulcorner M\urcorner\ulcorner N\urcorner = \ulcorner N\{M/x\}\urcorner$$

*Watch out for the needed* $\alpha$*-conversions.*

**Exercise 110.** *Define* **Beta** *such that* $\mathbf{Beta}\ulcorner M\urcorner = \ulcorner M'\urcorner$ *where* $M'$ *is the result of applying* $\rightarrow_\beta$ *on* $M$ *in a leftmost fashion (recall Def. 47). When* $M$ *is in* $\beta$*-normal form, we just let* $M' = M$ *instead.*

**Exercise 111.** *Define* **Eta** *to apply* $\rightarrow_\eta$ *until* $\eta$*-normal form is reached.*

**Exercise 112.** *Define* **IsNF** *to check, given* $\ulcorner M\urcorner$*, whether* $M$ *is in* $\beta\eta$*-normal form.*

**Exercise 113.** *Define* **IsNumeral** *to check, given* $\ulcorner M\urcorner$*, whether* $M$ *is a numeral. That is, is* $M$ *is syntactically of the (normal) form* $\lambda sz.\, s(s(\cdots(sz)\cdots))$*, for some variables* $s, z$*. (Return* **T** *on all possible* $\alpha$*-conversions.)*

**Exercise 114.** *Define* **IsClosed** *to check, given* $\ulcorner M\urcorner$*, whether* $M$ *is in* $\Lambda^0$*.*

**Note.** All the above functions can be conveniently defined using the $\mathbf{\Theta}$ operator, which implements recursive calls. While $\mathbf{\Theta}$ allows *arbitrarily nested* recursive calls, for the functions above we can predict a bound for the depth of these calls. Roughly, the bound is strictly connected with the *size* of the $\lambda$-term. Here, by "size" we mean the maximum nesting of $\lambda$-abstractions or applications that occur in the syntax of the $\lambda$-term at hand. So, for instance, a **Subst** operation computing $N\{M/x\}$ will never require more recursive calls than the size of $N$, if we write **Subst** in the straightforward way — i.e. by induction on the structure of $N$.

**Definition 115.** *The size of $M$, written $|M|$ is defined as*

$$|x| = 1 \qquad |NO| = 1 + \max(|N|, |O|) \qquad |\lambda x.\, N| = 1 + |N|$$

**Exercise 116.** *Show that $\#M + 1 \geq |M|$, for all $M$.*

So, all the function seen above can be rewritten, roughly, replacing $\mathbf{\Theta}$ with a "lesser" version of the fixed point operator, which unfolds recursive calls only until depth $\#M + 1$. This operator could be, e.g.

$$\mathbf{LimFix} = \lambda fnz.nfz$$

For instance $\mathbf{LimFix}\, F\ulcorner 3\urcorner\Omega = F(F(F\Omega))$. By comparison, $\mathbf{\Theta}F$ would generate an unbounded number of $F$'s.

**Exercise 117.** *Write* **Subst** *using* **LimFix** *instead of* $\mathbf{\Theta}$*. Start from* $\mathbf{Subst} = \lambda xmn.\mathbf{LimFix}\, F(\mathbf{Succ}\, n)$ *and then find $F$. Do the same for the other functions seen above in this section.*

We shall return on this "bounded recursion" approach when we shall deal with *primitive recursion*.

**Exercise 118.** *Construct another version of* **E** *using the results above (see Lemma 98). Name this variant* **Eval**. *Define it so that, when $M$ has no normal form,* $\mathbf{Eval}\ulcorner M\urcorner$ *is unsolvable.*

**Exercise 119.** *It can be often useful to consider only the $\lambda$-terms that produce numerals. To this aim define a* **Term** *operator such that*

$$\begin{array}{ll} \mathbf{Term}\ulcorner M\urcorner & = \mathbf{I} \qquad \textit{if } M =_{\beta\eta} \ulcorner n\urcorner \textit{ for some } n \\ \mathbf{Term}\ulcorner M\urcorner & \textit{is unsolvable} \quad \textit{otherwise} \end{array}$$

*You might want to start from:*

$$\begin{array}{ll} \mathbf{TermIn}\ulcorner k\urcorner\ulcorner M\urcorner & = \mathbf{T} \quad \textit{if } M \xrightarrow{leftmost}{}^{*}_{\beta} N \to^{*}_{\eta} \ulcorner n\urcorner \textit{ for some } n \textit{ and } N \\ & \qquad \textit{using at most } k \textit{ } \beta\textit{-steps} \\ \mathbf{TermIn}\ulcorner k\urcorner\ulcorner M\urcorner & = \mathbf{F} \quad \textit{otherwise} \end{array}$$

*which is satisfied by*

$$\mathbf{TermIn} = \lambda km.\, \mathbf{IsNumeral}(\mathbf{Eta}(k\, \mathbf{Beta}\, m))$$

*Then, show that* $\forall M \in \Lambda^0$, $\mathbf{Term}\ulcorner M \urcorner M$ *either evaluates to a numeral or is unsolvable.*

# Chapter 3

# Logical Characterization of Computable Functions

## 3.1 Primitive Recursive Functions

**Lemma 120.** *The function $f(n) = 0$ is $\lambda$-definable.*

*Proof.* Take **K0**. □

**Lemma 121.** *The function $f(n) = n + 1$ is $\lambda$-definable.*

*Proof.* Take **Succ**. □

**Lemma 122.** *The projection functions $f_i(n_1, \ldots, n_k) = n_i$ with $1 \leq i \leq k$ are $\lambda$-definable.*

*Proof.* Take $\lambda n_1 \cdots n_k.\, n_i$. □

Note: the above includes the identity function $f(n) = n$.

**Lemma 123.** *The $\lambda$-definable (partial) functions are closed under composition.*

*Proof.* Let $f, g$ be $\lambda$-defined by $F, G$. Then, $f \circ g$ can be $\lambda$-defined by

$$M = \lambda x.J(F(Gx))$$

where $J$ is the jamming factor $Gx(KI)I$, as per Ex. 76. Let us check this:

- When $f(g(n))$ is defined, then $g(n)$ is defined as some $m \in \mathbb{N}$ and $f(m)$ is defined as well. Then, when $x = \ulcorner n \urcorner$, we have $J = \mathbf{I}$, $G\ulcorner n \urcorner = \ulcorner m \urcorner$, and $F\ulcorner m \urcorner = \ulcorner f(g(n)) \urcorner$. It is the trivial to check that $M\ulcorner n \urcorner = \ulcorner f(g(n)) \urcorner$.

- When $f(g(n))$ is undefined, then either $g(n)$ is undefined, or $g(n) = m \in \mathbb{N}$ but $f(m)$ is undefined.
  - If $g(n)$ is undefined, then $G\ulcorner n \urcorner$ is unsolvable, so $J$ is also unsolvable by Ex. 75, so $M\ulcorner n \urcorner$ is also unsolvable by the same Exercise.
  - If $g(n) = m \in \mathbb{N}$ but $f(m)$ is undefined, then $J = \mathbf{I}$, $G\ulcorner n \urcorner = \ulcorner m \urcorner$, and $F\ulcorner m \urcorner$ is unsolvable. So, $M\ulcorner n \urcorner = J(F\ulcorner m \urcorner) = F\ulcorner m \urcorner$ is unsolvable as well.

□

The above result can be generalized to $n$-ary functions:

**Lemma 124.** *The $\lambda$-definable (partial) functions are closed under general composition. That is, if $f \in (\mathbb{N}^k \rightsquigarrow \mathbb{N})$ and $g_1, \ldots, g_k \in (\mathbb{N}^j \rightsquigarrow \mathbb{N})$, then the function*

$$h(x_1, \ldots, x_j) = f(g_1(x_1, \ldots, x_j), \ldots, g_k(x_1, \ldots, x_j))$$

*is $\lambda$-definable.*

*Proof.* Easy adaptation of Lemma 123. □

**Lemma 125.** *The $\lambda$-definable functions are closed under* primitive recursion*. That is, if $g, h$ are $\lambda$-definable, so is $f(n, n_1, \ldots, n_k)$, inductively defined as:*

$$\begin{aligned} f(0, n_1, \ldots, n_k) &= g(n_1, \ldots, n_k) \\ f(n+1, n_1, \ldots, n_k) &= h(n, n_1, \ldots, n_k, f(n, n_1, \ldots, n_k)) \end{aligned}$$

*Proof.* Let $G, H$ be the $\lambda$-terms defining $g, h$. Then $f$ is $\lambda$-defined by

$$\begin{aligned} F =& \lambda n n_1 \cdots n_k. \\ & J\, n \Big( \lambda c.\, J'\, \mathbf{Cons}(\mathbf{Succ}(c\mathbf{T}))(H(c\mathbf{T})n_1 \cdots n_k(c\mathbf{F})) \Big) (\mathbf{Cons}\, \mathbf{0}\, (Gn_1 \cdots n_k))\mathbf{F} \end{aligned}$$

where $J$ and $J'$ are the usual jamming factors to force the evaluation of $h$ and $g$:

$$\begin{aligned} J =& Gn_1 \cdots n_k(\mathbf{KI})\mathbf{I} \\ J' =& H(c\mathbf{T})n_1 \cdots n_k(c\mathbf{F})(\mathbf{KI})\mathbf{I} \end{aligned}$$

The $F$ above works starting from the pair $\langle 0, g(n_1, \ldots, n_k) \rangle$. Then we apply $n$ times a function to this pair, incrementing the first component, and applying $h$ to the second. Finally, we take the resulting pair and extract the second component (the $\mathbf{F}$ at the end). □

**Definition 126.** *The set of the* primitive recursive *functions $\mathcal{PR}$ is defined as the smallest set of (total) functions in $\mathbb{N}^k \to \mathbb{N}$ which:*

- *includes the constant zero function, the successor function, and the projections ("the initial functions"); and*
- *is closed under general composition; and*
- *is closed under primitive recursion.*

Some facts about primitive recursive functions:

- If $f \in \mathcal{PR}$, then $f$ is a *total* function.
- $\mathcal{PR}$, being inductively defined, is a denumerable set.

**Exercise 127.** *Show that the following functions are in $\mathcal{PR}$.*

- *the "conditional" function ("if-then-else"):*

$$\mathsf{cond}(0, x, y) = x \qquad \mathsf{cond}(k+1, x, y) = y$$

- *the addition,subtraction (return e.g. 0 when negative), multiplication,division (return e.g. 0 when impossible)*
- *the factorial function*
- *the equality comparison: $\mathsf{eq}(x, x) = 0$, and 1 otherwise*
- *the less-than-or-equal comparison: $\mathsf{lt}(x, x+k) = 0$, and 1 otherwise*
- *the* encode *functions for pairs and disjoint union (easy), as well as their inverses (not so easy).*

**Exercise 128.** *Show that if $f$ is a binary function and $f \in \mathcal{PR}$, then the function $g$ given by $g(x, y) = f(y, x)$ is in $\mathcal{PR}$ as well.*

We can compare $\mathcal{PR}$ to the set of $\lambda$-definable functions. By the lemmata above, each $f \in \mathcal{PR}$ is $\lambda$-definable. Clearly, if we take a $\lambda$-definable non-total function, this is not in $\mathcal{PR}$, so the $\lambda$-definable functions form a larger set that $\mathcal{PR}$.

What if we restrict to *total* $\lambda$-definable functions, then? We can prove that the set of total $\lambda$-definable functions is still larger than $\mathcal{PR}$.

Basically, each $f \in \mathcal{PR}$ is either one of the basic functions or obtained from them through composition/primitive recursion in a *finite* number of

steps. This is not different from having a kind of programming language "$\mathcal{PR}$" having exactly the constructs mentioned in Def. 126. As we did for the $\lambda$-calculus we can enumerate this $\mathcal{PR}$ language using the $\mathsf{encode}$ functions. After that, we use a diagonalization argument, and construct a function $f(n)$ as follows: 1) take the $\mathcal{PR}$ program which has $n$ as its encoding, 2) run it using $n$ as input, 3) take the result $r$, and 4) let $f(n) = r + 1$. By diagonalization, we have $f \notin \mathcal{PR}$. Yet, $f$ can be $\lambda$-defined! We just need to write an interpreter for this $\mathcal{PR}$ language in the $\lambda$ calculus in order to define $f$. This can done as we did for **E**.

**Exercise 129.** *Define the "$\mathcal{PR}$ language" as we did for $\Lambda$, and an encoding $\mathcal{PR} \leftrightarrow \mathbb{N}$. Then, $\lambda$-define an interpreter for this $\mathcal{PR}$ language.*

Using this interpreter, we can clearly $\lambda$-define the total $f$ defined above, proving that $\lambda$-definable functions form a larger set than $\mathcal{PR}$ functions.

**Theorem 130.** *The set of $\lambda$-definable functions is strictly larger than $\mathcal{PR}$ functions.*

### 3.1.1 Ackermann's Function

This is another interesting total function that is $\lambda$-definable but not in $\mathcal{PR}$.

$$\begin{array}{ll} \mathsf{ack}(0, y) & = y + 1 \\ \mathsf{ack}(x + 1, 0) & = \mathsf{ack}(x, 1) \\ \mathsf{ack}(x + 1, y + 1) & = \mathsf{ack}(x, \mathsf{ack}(x + 1, y)) \end{array}$$

[also see Cutland page 46]

**Exercise 131.** *Show that* $\mathsf{ack}$ *is $\lambda$-definable.*

Note the "double recursion" in the last line. This is not a problem in the $\lambda$ calculus, but in $\mathcal{PR}$ we can only express "single" recursion. It is not obvious whether this form of double recursion can be somehow expressed using the single recursion of $\mathcal{PR}$.

It turns out that $\mathsf{ack}$ is *not* a primitive recursive function. So, this form of "double recursion" is (generally) not allowed in $\mathcal{PR}$. The actual proof for $\mathsf{ack} \notin \mathcal{PR}$ is rather long, so we omit it. We however provide some intuition below.

Roughly, the proof relies on $\mathsf{ack}$ to grow at a very, very high speed. Observe the following. We have $\mathsf{ack}(1, y) = y + 2$, as well as $\mathsf{ack}(2, y) = 3 + 2 \cdot y > 2 \cdot y$. Note the rôle of $y$ and 2 here: from $y + 2$ (addition) we went to $2 \cdot y$ (multiplication) by just incrementing the first parameter to $\mathsf{ack}$.

Moreover, $\mathsf{ack}(3, y) > 2^y$ (exponential), and $\mathsf{ack}(4, y) > 2^{2^{2^{\cdots}}}$ where there are $y$ exponents. And this goes on, generating very fast-growing functions.

Indeed, the $\mathsf{ack}$ beats each function in $\mathcal{PR}$:

$$\forall f \in \mathcal{PR}.\, \exists k \in \mathbb{N}.\, \forall y \in \mathbb{N}.\, \mathsf{ack}(k, y) > f(y)$$

The above can be proved by induction on the derivation of $f$ (we omit the actual proof). From here, one can prove that $\mathsf{ack} \notin \mathcal{PR}$ by contradiction: if $\mathsf{ack} \in \mathcal{PR}$, we also would have that $f(y) = \mathsf{ack}(y, y)$ is a primitive recursive function. By the statement above, we get some $k$ such that $\forall y \in \mathbb{N}.\, \mathsf{ack}(k, y) > \mathsf{ack}(y, y)$. If we now choose $y = k$, we get a contradiction.

**Exercise 132.** *Let us recap the main proof techniques:*

- *If we take $A = \mathcal{PR} \cup \{\mathsf{ack}\}$, do we get the whole set of total functions $\mathbb{N} \to \mathbb{N}$ ?*
- *Let $B$ be the closure of $A$ under general composition and primitive recursion. Is $B$ the whole set $\mathbb{N} \to \mathbb{N}$ ?*
- *Is $B$ the set of total $\lambda$-definable functions?*

## 3.2 General Recursive Functions

**Exercise 133.** *Let $f(x, y)$ be a total $\lambda$-definable function. Show that*

$$g(x, z) = \mu y < z.\, f(x, y) = 0$$

*is a total $\lambda$-definable function. By $\mu y < z.\, f(x, y) = 0$ we mean the least $y$ such that $y < z$ and $f(x, y) = 0$. If such a $y$ does not exist, we let the result to be $z$. This operation is called* bounded minimalisation.

**Exercise 134.** *Let $f(x, y)$ be in $\mathcal{PR}$. Show that*

$$g(x, z) = \mu y < z.\, f(x, y) = 0$$

*is in $\mathcal{PR}$. So primitive recursive functions are closed under bounded minimalisation.*

We now investigate what is missing from the definition of $\mathcal{PR}$ that makes it different from the whole $\lambda$-definable functions. Basically, the problem boils down to constructing an interpreter of the $\lambda$ calculus using the $\mathcal{PR}$ operators, that is:

"What is missing for (a variant of) **E** to be a function in $\mathcal{PR}$ ?"

Consider the construction of the step-by-step interpreter **Eval**, given in Ex. 118. All the basic constituents (**Beta**, **Eta**, **IsNumeral**, **IsNF**, **Subst**) can be defined using **LimFix**, which is basically the same thing of the primitive recursion operator: it iterates a function for a fixed number of times. So, these constituents can be indeed constructed inside $\mathcal{PR}$. For instance, $\exists\, \mathsf{subst} \in \mathcal{PR}$ such that

$$\mathsf{subst}(\#x, \#M, \#N) = \#(N\{M/x\})$$

and so on for the other basic functions. This means that the "single-step" function, implementing a single leftmost $\rightarrow_\beta$ step, is actually in $\mathcal{PR}$.

**Lemma 135.** *The functions*

$$\begin{aligned}
&\mathsf{subst} \in \mathbb{N}^3 \to \mathbb{N}\\
&\mathsf{beta} \in \mathbb{N} \to \mathbb{N}\\
&\mathsf{eta} \in \mathbb{N} \to \mathbb{N}\\
&\mathsf{isNumeral} \in \mathbb{N} \to \mathbb{N}\\
&\mathsf{isNF} \in \mathbb{N} \to \mathbb{N}\\
&\mathsf{app} \in \mathbb{N}^2 \to \mathbb{N}\\
&\mathsf{num} \in \mathbb{N} \to \mathbb{N}
\end{aligned}$$

*which are the arithmetic equivalents of the $\lambda$-terms* **Subst**,**Beta**,**Eta**,**IsNumeral**, **IsNF**,**App**,**Num**, *are in $\mathcal{PR}$.*

*Proof.* Left as a (long, and not so trivial) exercise. You might want to start from $\mathsf{subst}(x, n, m) = \mathsf{aux}(x, n, m, 2^m)$. □

**Exercise 136.** *Show that the function* $\mathsf{extract}(\#\ulcorner n \urcorner) = n$ *is in $\mathcal{PR}$. (Make it work on all possible $\alpha$-conversions of $\ulcorner n \urcorner$. Also, define* $\mathsf{extract}(x) = 0$ *for other inputs $x$.)*

So what is missing for a full interpreter? We do not know *how many* $\rightarrow_\beta$ *steps* are needed to reach normal form. For a full interpreter, we need unbounded iteration of the single-step function. So, we can augment $\mathcal{PR}$ with an *unbounded minimalisation* operator.

**Definition 137.** *The set of (partial)* general recursive *functions ($\mathcal{R}$) is defined as the smallest set of partial functions in $\mathbb{N}^k \rightsquigarrow \mathbb{N}$ which:*

- *includes the constant zero function, the successor function, and the projections ("the initial functions"); and*
- *is closed under general composition; and*
- *is closed under primitive recursion ; and*
- *is closed under unbounded minimalisation.*

*Unbounded minimalisation is defined as follows: given $f(x, y)$, we construct $g(x)$ as*

$$g(x) = (\mu y.\, f(x, y) = 0)$$

*(the least $y \in \mathbb{N}$ such that $f(x, y) = 0$.) Note that if no such $y$ exists, $g(x)$ is undefined, hence making $g$ a partial function. This definition is naturally extended to $n$-ary functions in $\mathbb{N}^k \rightsquigarrow \mathbb{N}$.*

**Exercise 138.** *Show that the following functions are in $\mathcal{R}$:*

- *$f = \emptyset$ (the always-undefined function)*
- *$f(2 \cdot n) = 1$ and $f(2 \cdot n + 1)$ undefined*
- *$\mathsf{ack}(x, y)$ (this is not so easy)*
  *Hint: one way to do it is by implementing a stack using $\mathsf{encode}_\times$.*

**Lemma 139.** *The $\lambda$-definable functions are closed under unbounded minimalisation.*

*Proof.* Let $f$ be $\lambda$-defined by $F$. Then, $g(x) = (\mu y.\, f(x, y) = 0)$ can be $\lambda$-defined by

$$G = \mathbf{\Theta}(\lambda g y x.\, \mathbf{Eq}\, \mathbf{0}\, (F x y)\, y\, (g(\mathbf{Succ}\, y) x))\mathbf{0}$$

□

**Lemma 140.** *The set of recursive functions $\mathcal{R}$ is included in the set of $\lambda$-definable functions.*

*Proof.* Immediate by all the lemmata above. □

**Theorem 141.** *The set of $\lambda$-definable functions is exactly the same as the set of recursive functions $\mathcal{R}$.*

*Proof.* We already proved that each $f \in \mathcal{R}$ is $\lambda$-definable. Now we prove that if $f$ is $\lambda$-definable (say by $F$), then $f \in \mathcal{R}$. By Exercise 127, $\mathsf{proj1}, \mathsf{proj2} \in \mathcal{PR}$, so by Lemma 135, and Exercise 136, we can define the following functions in $\mathcal{R}$:

$$\begin{aligned}
&\mathsf{steps}(x, 0) = x \\
&\mathsf{steps}(x, n+1) = \mathsf{beta}(\mathsf{steps}(x, n)) \\
&\mathsf{eval}(x) = \mathsf{extract}(\mathsf{proj1}(\mu n.\, \mathsf{and}(A, B) = 0) \\
&\qquad \text{where} \quad A = \mathsf{isNumeral}(C) \\
&\qquad\qquad\quad\;\; B = \mathsf{eq}(C, \mathsf{proj1}(n)) \\
&\qquad\qquad\quad\;\; C = \mathsf{eta}(\mathsf{steps}(x, \mathsf{proj2}(n))) \\
&\mathsf{f}(y) = \mathsf{eval}(\mathsf{app}(\#F, \mathsf{num}(y)))
\end{aligned}$$

We now claim that we indeed have $\forall y.\, \mathsf{f}(y) = f(y)$. First, we note that $\mathsf{app}(\#F, \mathsf{num}(y)) = \mathsf{app}(\#F, \#^{\ulcorner} y^{\urcorner}) = \#(F^{\ulcorner} y^{\urcorner})$.

- If $f(y)$ is undefined, then $F^{\ulcorner} y^{\urcorner}$ has no normal form. So, no matter what $\mathsf{proj2}(n)$ evaluates to, the function $\mathsf{steps}$ will perform that many $\beta$-steps on $x$, but will not reach the index of a $\beta$-normal form. So, $A$ will always evaluate to "false" (i.e. nonzero), since $\mathsf{isNumeral}$ syntactically checks against numerals, which are in normal form. Hence, the $\mathsf{and}(A, B)$ will always return "false", and the minimalisation operator $\mu n$ will keep on trying every $n \in \mathbb{N}$, in an infinite loop, and so making $\mathsf{f}(y)$ undefined.

- If $f(y)$ is defined, say $f(y) = z \in \mathbb{N}$, then $F^{\ulcorner} y^{\urcorner}$ has as its normal form the numeral $^{\ulcorner} z^{\urcorner}$. Define $k$ as the number of leftmost $\to_\beta$ steps needed to reach normal form. Therefore, $\mathsf{eta}(\mathsf{steps}(\#(F^{\ulcorner} y^{\urcorner}), k))$ will completely evaluate $F^{\ulcorner} y^{\urcorner}$ until $\beta\eta$ normal form, producing the index of a $\lambda$-term $M$, which is an $\alpha$-conversion[1] of $^{\ulcorner} z^{\urcorner}$. The minimalisation operator $\mu n$ will try each $n \in \mathbb{N}$, from 0 upwards.

  - When $0 \leq n < \mathsf{encode}_{\times}(\#M, k)$, we show that $\mathsf{and}(A, B)$ returns "false" (nonzero), so that the minimalisation will try the next $n$. By contradiction, assume that $\mathsf{and}(A, B)$ returns "true". This means that $A$ and $B$ are both "true". Since $A$ is "true",

[1] Recall Exercise 90. While we know that $M$ is of the form $\lambda ab.\, a(a(a(\cdots(a(ab)))))$, it still might be syntactically different from $^{\ulcorner} z^{\urcorner}$ by picking different variable names for $a$ and $b$. This mainly depends on the fact that we do not require our $\mathsf{beta}$ function to choose exactly the variables we use in the definition of $^{\ulcorner} z^{\urcorner}$.

$\mathsf{eta}(\mathsf{steps}(\#(F^{\ulcorner}y^{\urcorner}), \mathsf{proj2}(n)))$ is the index $i$ of a numeral, hence the index of a normal form of $F^{\ulcorner}y^{\urcorner}$. Since we need to do $k$ steps to reach normal form, we have[2] $\mathsf{proj2}(n) \geq k$. This implies that $i = \#M$. Since $B$ is "true", $\mathsf{proj1}(n) = i = \#M$. Hence $n = \mathsf{encode}_{\times}(\mathsf{proj1}(n), \mathsf{proj2}(n)) = \mathsf{encode}_{\times}(\#M, \mathsf{proj2}(n)) \geq \mathsf{encode}_{\times}(\#M, k)$, contradicting $n = \mathsf{encode}_{\times}(\mathsf{proj1}(n), \mathsf{proj2}(n)) < \mathsf{encode}_{\times}(\#M, k)$.

- So, eventually the $\mu n$ operator will try $n = \mathsf{encode}_{\times}(\#M, k)$. Here, it is trivial to check that $A$ and $B$ are both "true", so the loops halts. Indeed, we have that $\mathsf{eta}(\mathsf{steps}(\#(F^{\ulcorner}y^{\urcorner}), k))$ is a numeral (so $A$ is "true"[3]), and indeed $\mathsf{eta}(\mathsf{steps}(\#(F^{\ulcorner}y^{\urcorner}), k)) = \#M = \mathsf{proj1}(n)$ (so $B$ is "true").

So, the result of the whole $\mu n. \cdots$ expression is $\mathsf{encode}_{\times}(\#M, k)$. After we compute this, the definition of $\mathsf{eval}$ performs a $\mathsf{proj1}$, hence obtaining $\#M$. Finally, the $\mathsf{extract}$ function is applied, extracting $z$ from the index of $M =_{\alpha} {}^{\ulcorner}z^{\urcorner}$. We conclude that, when $f(y) = z$, we have $\mathsf{f}(y) = z$.

Since we proved both inclusions, we conclude that the set of $\lambda$-definable functions coincides with $\mathcal{R}$. □

**Exercise 142.** *Provide an alternative proof for Th. 141, following these hints.*

*First, define a function $g$ that given $i, x, k$ will run program number $i$ on input $x$ for $k$ steps, assume the result is a numeral (hence a normal form), and extract the result as a natural number. When the result is not a numeral, return anything you want (e.g. 0). Show that $g$ is recursive (actually, in $g \in \mathcal{PR}$).*

*Then, define a partial function $h$ that given $i, x$ returns the number of steps $k$ required for program number $i$ to halt on input $x$, reaching normal form. Function $h$ is undefined when no such $k$ exists. Use minimalisation for this.*

*Finally, build* $\mathsf{eval}$ *using $g$ and $h$.*

[2] The function $\mathsf{beta}$ has to be applied at least $k$ times to reach normal form. After normal form is reached, we required $\mathsf{beta}$ to act as the identity.

[3] Recall we require $\mathsf{isNumeral}$ to return "true" on all $\alpha$-conversions of numerals.

## 3.3 T,U-standard Form

This classical result states that every partial recursive function can be expressed by using the primitive recursion constructs and a *single* use of the unbounded minimalisation operator.

**Theorem 143.** *There exist* $\mathsf{T}, \mathsf{U} \in \mathcal{PR}$ *such that, each (partial) recursive function* $f \in \mathcal{R}$ *can be written as*

$$f(x) = \mathsf{U}(\mu n.\mathsf{T}(i, x, n))$$

*for some suitable natural* $i$*.*

*Proof.* We have already proved this when we proved 141. Indeed, the definition of $\mathsf{f}$ in that proof mentions a single $\mu n$ operator, using only primitive recursive functions inside of the $\mu n$, as well as outside of it. So $\mathsf{T}$ and $\mathsf{U}$ simply are defined in that way. The integer $i$ is instead the index $\#F$ for some $\lambda$-term $F$ that defines the function $f \in \mathcal{R}$. This $F$ indeed exists by Lemma 140. □

## 3.4 The FOR and WHILE Languages

Consider an imperative language having the following commands. Below we use `x` for variables (over $\mathbb{N}$), `e` for arithmetic expressions over variables, and `c` for commands.

- Assignment: `x:= e`
- Conditional: `if x = 0 then c1 else c2`
- Sequence: `c1 ; c2`
- For-loop: `for x := e1 to e2 do c`

Name this language "FOR".

The semantics of this language should be mostly obvious. We assume that `e1` and `e2` are evaluated *only once*, at the beginning of the for-loop. For instance, the command

```
y := 6 ;
for x := 1 to y do
    y := y + 1
```

will *terminate*, performing exactly six loop iterations. Further, we assume that the loop variable `x` is updated to the next value in the sequence from `e1` to `e2`, even if the loop body modifies the variable `x`. For instance,

```
sum := 0 ;
for x := 1 to 6 do
    sum := sum + x ;
    x := x - 1
```

will *terminate*, performing exactly six loop iterations. When the loop is exited, the variable `sum` has value $0+1+2+3+4+5+6 = 21$. Note that, under these assumptions, our for-loops will always terminate.

**Exercise 144.** *Define the formal semantics of the FOR language, as a function* $\mathbb{N} \to \mathbb{N}$*. Assume the input of FOR programs is just provided through a special* `input` *variable. Similarly, read the output of the program through a special* `output` *variable, to be read at the end of execution.*

**Definition 145.** *A function $f$ is FOR-definable if there is some FOR-program that has semantics $f$.*

**Theorem 146.** *The set of FOR-definable functions is exactly $\mathcal{PR}$.*

*Proof.* Left as a (rather long) exercise. You basically have to 1) simulate all the constructs of $\mathcal{PR}$ using the FOR-commands, and 2) simulate all FOR-commands using the $\mathcal{PR}$-constructs. This can be done by exploiting the $\mathsf{encode}_\times$ function to build arrays, so to store the whole execution state in a few variables. □

Now, we can extend the FOR language with the following construct:

- While-loop: `while x > 0 do c`

Name this language "WHILE". Note that, unlike FOR programs, WHILE programs might not terminate.

**Exercise 147.** *Define the semantics of WHILE programs.*

**Definition 148.** *A function $f$ is WHILE-definable if there is some WHILE-program that has semantics $f$.*

**Theorem 149.** *The set of WHILE-definable functions is exactly $\mathcal{R}$.*

*Proof.* (sketch)
($\subseteq$): a WHILE interpreter can be written in the $\lambda$-calculus (long exercise). So, each WHILE-definable function is in $\mathcal{R}$ by Th. 141.
($\supseteq$): Let $f \in \mathcal{R}$. We must find a WHILE program defining $f$. Take $\mathsf{T}$,$\mathsf{U}$ as in Th. 143. By Th. 146, $\mathsf{T}$ and $\mathsf{U}$ are FOR-definable, hence WHILE-definable. Following again Th. 143, all we have to do is to "add the missing $\mu n$" and compose $\mathsf{T}$ and $\mathsf{U}$ so to actually compute $f$. A single `while` construct is sufficient to try each $n \in \mathbb{N}$, thus emulating the $\mu n$ operator. □

**Theorem 150.** *Every WHILE-definable function can be WHILE-defined by a program having a single* `while` *loop.*

*Proof.* Direct consequence of Th. 143. □

## 3.5 Church's Thesis

Roughly, *all* programming languages can be proved equivalent w.r.t. the $\lambda$-calculus as we did for the WHILE language; that is, the set of the $\{\lambda$, WHILE, Java, $\dots\}$-definable functions does not depend on the choice of the programming language $L$. All you need to check is that

- all $\lambda$-definable functions are definable in the language L; e.g. you can write an interpreter for the $\lambda$-calculus in L
- all L-definable functions are definable in the $\lambda$-calculus; e.g. you can write an interpreter for L in the $\lambda$-calculus

The Church's Thesis is an informal statement, stating that

> The set of *intuitively* computable functions is exactly the set of functions definable in the $\lambda$-calculus (or Java, or Turing machines, or ⟨insert your favourite programming language here⟩).

Notable languages *not* equivalent to the $\lambda$-calculus:

- Plain HTML (with no Javascript). HTML just produces a hypertext, possibly formatted (e.g. by using CSS). However, you can not use HTML to "compute" anything. Indeed, it is not a programming language, but a hypertext description language.
- Plain SQL query language. It just searches the database for data, and return the results. It can not be used for general computing. Again, it is not a programming language, but only a query language. This

is actually *good*, because SQL queries can therefore be guaranteed to terminate.

Notable languages *equivalent* to the $\lambda$-calculus:

- PostScript and PDF. They should *only* describe a document. They allow for general recursion, so they could take a long time just to output one page. They can also loop, and fail to terminate, while requiring more and more memory. PostScript can even produce an infinite number of pages. By Rice, there is no effective way of predicting how many pages a PostScript file will print, since the number of pages is a semantic property.
- XSLT and XQuery. They should only perform some simple manipulation over XML. Due to some recursive constructs, they are actually able to achieve the power of the $\lambda$-calculus. So, it might happen that their execution does not terminate, allocating more memory, etc.
- Javascript. This is indeed a full-featured programming language. Running it inside a browser allows for arbitrary interaction with HTML, but exposes the browser to denial of service attacks, since the Javascript program can allocate more and more memory and fail to terminate. Naïve execution of Javascript can easily cause the browser to freeze. Firefox currently tries to mitigate the issue in this way. It runs the Javascript for a given amount of time (say 20 seconds). If it fails to halt, Firefox asks the user if he/she wants to abort the Javascript computation, or wait for other 20 seconds, after which the same question is asked to the user again.
- Turing Machines
- "conventional" programming languages

# Chapter 4

# Classical Results

In the previous sections, we studied $\lambda$-definability. While $\lambda$-definability is a powerful notion, it does not provide a semantics (a function $\mathbb{N}^k \rightsquigarrow \mathbb{N}$) for *all* $\lambda$-terms.

- For instance, $M = \lambda n.\, n\, \mathbf{K}\, \mathbf{\Theta}$ does not $\lambda$-define any partial function $f \in \mathbb{N}^k \rightsquigarrow \mathbb{N}$, for any $k > 0$. Indeed, if we let the first argument to be $k-1$, we get $M \ulcorner k-1 \urcorner \ulcorner x_2 \urcorner \cdots \ulcorner x_k \urcorner = (\lambda y_2 \ldots y_k.\, \mathbf{\Theta}) \ulcorner x_2 \urcorner \cdots \ulcorner x_k \urcorner = \mathbf{\Theta}$ which is solvable (by **KI**) but not a numeral.
- Another example is $N = \lambda n.\, n\, \mathbf{K}\, (\lambda y.\, yy)$. Assume this $\lambda$-defines $f \in \mathbb{N}^k \rightsquigarrow \mathbb{N}$. We get a contradiction from $N \ulcorner k-1 \urcorner \ulcorner x_2 \urcorner \cdots \ulcorner x_k \urcorner = \lambda y.\, yy$ which is solvable, but not a numeral.

However, we define an alternative semantics, relating each $\lambda$-term to a partial function. So, $\phi_i$ shall be the function related to the program $M$ having index $i$.

**Definition 151.** *$\phi_i(x) = y$ iff $M \ulcorner x \urcorner =_{\beta\eta} \ulcorner y \urcorner$ where $\#M = i$*

The above is trivially generalized to $k$-ary functions.

First, note that $\phi_i(x)$ is well-defined, since there can be at most one $y \in \mathbb{N}$ satisfying $M \ulcorner x \urcorner =_{\beta\eta} \ulcorner y \urcorner$. Second, note that $\phi_i$ is a partial function, which can be undefined when either $M \ulcorner x \urcorner$ has no normal form, or when $M \ulcorner x \urcorner$ *has* a normal form, but that normal form is not a numeral.

**Lemma 152.** *$f$ is $\lambda$-definable iff $f = \phi_i$ for some $i$*

*Proof.* ($\Rightarrow$) Let $f$ be $\lambda$-defined by $F$. Then, we take $i = \#F$, and check that $\phi_i = f$.

- If $f(x)$ is undefined, then we have that $F\ulcorner x\urcorner$ is unsolvable, so it has no normal form, so it is $\neq_{\beta\eta} \ulcorner y\urcorner$ for all $y$, and $\phi_i(x)$ is then undefined.
- If $f(x) = y \in \mathbb{N}$, then $F\ulcorner x\urcorner = \ulcorner f(x)\urcorner$, and $\phi_i(x) = f(x)$.

Since the above holds for any $x$, we get $\phi_i = f$.

($\Leftarrow$) Let $f = \phi_i$, and let $M$ such that $i = \#M$. Then, $f$ can be $\lambda$-defined by $F = \lambda n.\, J\, M\, n$, where $n \notin \mathsf{free}(M)$ and $J$ is the jamming factor

$$J = \mathbf{Term}(\mathbf{App}\ulcorner M\urcorner(\mathbf{Num}\, n))$$

and **Term** is from Ex. 119: $\mathbf{Term}\ulcorner O\urcorner$ evaluates to $\mathbf{I}$ when $O$ has a numeral as its normal form; otherwise it is unsolvable.

It is easy to check that $F$ indeed $\lambda$-defines $f$. When $f(x)$ is undefined, then $M\ulcorner x\urcorner$ has no numeral as normal form, and thus $J$ is unsolvable. Otherwise, when $f(x)$ is defined, $M\ulcorner x\urcorner$ has $\ulcorner f(x)\urcorner$ as its normal form, and thus $J = \mathbf{I}$. In this case, $F\ulcorner x\urcorner = M\ulcorner x\urcorner = \ulcorner f(x)\urcorner$. □

## 4.1 Padding Lemma

**Theorem 153** (Padding Lemma)**.**
*There is a function* $\mathsf{pad} \in \mathcal{R}$ *such that*

$$\phi_n = \phi_{\mathsf{pad}(n)} \wedge \mathsf{pad}(n) > n$$

*Proof.* Immediate from the Padding Lemma for the $\lambda$-calculus. □

Also see [Cutland]

## 4.2 Parameter Theorem (a.k.a. *s-m-n* Theorem)

**Theorem 154** (Parameter Theorem, *s-m-n* Theorem)**.**
*For all* $m > 0$ *and* $0 < n \leq m$*, there exists a total recursive function* $\mathsf{s}^n_m(i, x)$ *such that for all* $y_1, \ldots, y_m$

$$\phi_{\mathsf{s}^n_m(i,x)}(y_1, \ldots, y_{n-1}, y_{n+1}, \ldots, y_m) = \phi_i(y_1, \ldots, y_{n-1}, x, y_{n+1}, \ldots, y_m)$$

*Proof.* Easy adaptation of the Parameter Lemma for the $\lambda$-calculus. □

Also see [Cutland]

**Exercise 155.** *Show that* $\mathsf{pad}$ *and* $s^n_m$ *are primitive recursive functions.*

## 4.3 Universal Program

**Theorem 156** (Universal Program)**.**
*The partial function $f(x, y) = \phi_x(y)$ is recursive.*
*This can be generalized to n-ary partial functions.*

*Proof.* Easy adaptation of the Universal Program for the $\lambda$-calculus. We actually described such an $f$ in the proof of Th. 141. □

Also see [Cutland]

## 4.4 Fixed Point Theorem, a.k.a. Kleene's Second Recursion Theorem

**Theorem 157** (Kleene's Second Fixed Point Theorem (a.k.a. Second Recursion Theorem))**.**
*For each total computable function $f$, there is some $n \in \mathbb{N}$ such that*

$$\phi_n = \phi_{f(n)}$$

*Proof.* We adapt the proof of Th. 101. By Th. 156, the following is recursive:

$$g(x, y) = \phi_{f(s(x,x))}(y)$$

so $\phi_a = g$ for some $a$. Now take $n = s(a, a)$. We then have, for all $y$,

$$\phi_n(y) = \phi_{s(a,a)}(y) = \phi_a(a, y) = g(a, y) = \phi_{f(s(a,a))}(y) = \phi_{f(n)}(y)$$

□

Also see [Cutland]

## 4.5 Recursively enumerable Sets

**Definition 158.** *A set $A \subseteq \mathbb{N}$ is recursive iff the function $\chi_A$ is recursive. With some abuse of notation, we write $A \in \mathcal{R}$.*

So, a set $A$ is recursive if and only if there is a verifier program, returning "true" on $A$ and "false" on its complement $\bar{A}$.

One might wonder what happens if the verifier is not required to terminate for all inputs. For instance the verifier could simply terminate on $A$, and diverge on $\bar{A}$. We might call this a "partial verifier", or semi-verifier.

**Definition 159.** *A set $A \subseteq \mathbb{N}$ is recursively enumerable ($A \in \mathcal{RE}$) if and only if $A = \mathsf{dom}(f)$ for some $f \in \mathcal{R}$.*

Terminology: a recursive set is sometimes said to be decidable, computable, effective, $\lambda$-definable, WHILE-definable, ... These adjectives are equivalent. Recursively enumerable sets are said to be semi-decidable, semi-computable, ... instead.

**Exercise 160.** *Prove that the following properties of a set $A$ are equivalent.*

- $A \in \mathcal{RE}$
- *there is some $\lambda$-term $S_A$ such that $A = \mathsf{dom}(\phi_{\#S_A})$*
- *there is some $\lambda$-term $S_A$ such that $A = \{n \mid S_A{}^{\ulcorner}n^{\urcorner} \text{ has a normal form}\}$*
- *there is some $\lambda$-term $S_A$ such that $A$ is semi-$\lambda$-defined by $S_A$, that is $A = \{n \mid S_A{}^{\ulcorner}n^{\urcorner} \text{ has a normal form}\}$ and $\bar{A} = \{n \mid S_A{}^{\ulcorner}n^{\urcorner} \text{ is unsolvable}\}$*

*Hint: use the step-by-step interpreter and check the results using* **IsNumeral** *and related functions. Apply jamming factors as needed.*

By the exercise above, we have that the many different formalizations of "semi-verifier" are actually equivalent.

**Definition 161.** $\mathsf{K}' = \{n | \phi_n(n) \text{ is defined}\}$

**Lemma 162.** $\mathsf{K}' \notin \mathcal{R}$

*Proof.* Similar the the argument for $\mathsf{K}$. By contradiction, if $\mathsf{K}' \in \mathcal{R}$, then

$$f(n) = \begin{cases} \phi_n(n) + 1 & \text{if } n \in \mathsf{K}' \\ 0 & \text{otherwise} \end{cases}$$

would be a total recursive function. Hence, $f = \phi_a$ for some $a$. Since $f$ is total, $a \in \mathsf{K}'$, so we reach the contradiction $\phi_a(a) = f(a) = \phi_a(a) + 1$. □

**Lemma 163.** $\mathsf{K}' \in \mathcal{RE}$

*Proof.* By Th. 156, $f(n) = \phi_n(n)$ is in $\mathcal{R}$, and clearly $\mathsf{dom}(f) = \mathsf{K}'$. □

**Lemma 164.** $A \in \mathcal{R} \implies A \in \mathcal{RE}$

*Proof.* If $V_A$ $\lambda$-defines A, then $\lambda n.\, V_A\, n\, \mathbf{I}\, \Omega$ is a semi-verifier. □

**Lemma 165.** $A \in \mathcal{RE} \wedge \bar{A} \in \mathcal{RE} \implies A \in \mathcal{R}$

*Proof.* Given two semi-verifiers $S_A, S_{\bar{A}}$ for $A$ and $\bar{A}$, we execute them "in parallel" to construct a verifier for $A$. That is, suppose we are checking whether $n \in A$. An effective procedure could be:

- check whether $S_A{}^{\ulcorner}n^{\urcorner}$ halts in 1 step: if so, return "true"
- check whether $S_{\bar{A}}{}^{\ulcorner}n^{\urcorner}$ halts in 1 step: if so, return "false"
- check whether $S_A{}^{\ulcorner}n^{\urcorner}$ halts in 2 steps: if so, return "true"
- check whether $S_{\bar{A}}{}^{\ulcorner}n^{\urcorner}$ halts in 2 steps: if so, return "false"
- ...

This loop *will* eventually stop, since either $S_A{}^{\ulcorner}n^{\urcorner}$ or $S_{\bar{A}}{}^{\ulcorner}n^{\urcorner}$ must eventually halt. When one of them halts, we "abort" the parallel execution of the other and return the result. □

**Exercise 166.** *Construct the $\lambda$-term of the verifier used in the proof above.*

**Lemma 167.** $A \in \mathcal{RE} \wedge \bar{A} \in \mathcal{RE} \iff A \in \mathcal{R}$

*Proof.* Immediate by the lemmata above. □

**Lemma 168.** $\bar{\mathsf{K}}' \notin \mathcal{RE}$

*Proof.* Immediate by Lemma 167 and $\mathsf{K}' \in \mathcal{RE} \setminus \mathcal{R}$. □

**Lemma 169.** *All the following properties of a set $A \subseteq \mathbb{N}$ are equivalent*

1. $A \in \mathcal{RE}$
2. $A = \emptyset$ *or* $A$ *is the range of a total recursive function*
3. $A = \{n | \exists m.\, \mathsf{encode}_\times(n, m) \in B\}$ *for some* $B \in \mathcal{R}$
4. $A$ *is the range of a partial recursive function*

*Proof.* (1 $\implies$ 2) If $A$ is empty, it is straightforward. Otherwise, assume $x \in A$, and let $A = \mathsf{dom}(\phi_a)$. Then

$$f(n) = \begin{cases} \mathsf{proj1}(n) & \text{if running } \phi_a(\mathsf{proj1}(n)) \text{ halts in } \mathsf{proj2}(n) \text{ steps} \\ x & \text{otherwise} \end{cases}$$

Clearly $f$ is a total recursive function. Also, $\mathsf{ran}(f)$ is included in $A$ by construction. Moreover, if $y \in A$, then running $\phi_a(y)$ must halt, say in $k$

steps, implying $f(\mathsf{encode}_\times(y,k)) = y$, so $y \in \mathsf{ran}(f)$.

(2 $\implies$ 3) If $A = \emptyset$, take $B = \emptyset$. Otherwise, let $A = \mathsf{ran}(f)$, for a total recursive $f$. Define $B = \{\mathsf{encode}_\times(f(x), x) | x \in \mathbb{N}\}$. This $B$ is in $\mathcal{R}$: to check whether $n \in B$, we check that $f(\mathsf{proj2}(n)) = \mathsf{proj1}(n)$, which is doable because $f$ is total, so everything halts, and we can always effectively decide that equation. It is trivial to check that $A$ is indeed $\{n | \exists m.\, \mathsf{encode}_\times(n,m) \in B\}$.

(3 $\implies$ 4) Given $B \in \mathcal{R}$, consider the following partial function:

$$f(x) = \begin{cases} \mathsf{proj1}(x) & \text{if } x \in B \\ undefined & \text{otherwise} \end{cases}$$

Clearly, $f \in \mathcal{R}$. Also, $\mathsf{ran}(f) = A$.

(4 $\implies$ 1) By hypothesis, $A$ is the range of a partial recursive function $f$. Take $a$ such that $f = \phi_a$. Take $n$ as input, and "run" the following:

> For each $i \in \mathbb{N}$, starting from 0:
> Run $\phi_a(\mathsf{proj1}(i))$ for at most $\mathsf{proj2}(i)$ steps
> If that halts, and $\phi_a(\mathsf{proj1}(i)) = n$, stop (e.g. return 0)
> Otherwise, try the next $i$

This can be actually implemented in the $\lambda$-calculus using the step-by-step interpreter. Let $j$ be the index of that $\lambda$-term. It is easy to check that $\mathsf{dom}(\phi_j) = \mathsf{ran}(f) = A$, implying $A \in \mathcal{RE}$. Indeed, if $n \in \mathsf{ran}(f)$, then $n = f(x)$, and $\phi_a(x)$ can be computed in $y$ steps, for some $x$ and $y$. So, when $i = \mathsf{encode}_\times(x,y)$ the loop above stops, therefore $n \in \mathsf{dom}(\phi_j)$. For the other direction, if $n \in \mathsf{dom}(\phi_j)$, then the loop stops, so $f(\mathsf{proj1}(i)) = n$, for some $i$, and $n \in \mathsf{ran}(f)$.

*Summary.* The implications form a cycle $1 \implies 2 \implies 3 \implies 4 \implies 1$, so the properties $1, 2, 3, 4$ are equivalent. □

**Lemma 170.** $\mathsf{K} \in \mathcal{RE}$

*Proof.* We have $\mathsf{K} = \{n | \exists m.\, \mathsf{encode}_\times(n,m) \in B\}$ where

$$B = \{\mathsf{encode}_\times(n,m) | n = \#M \wedge M\ulcorner M\urcorner \text{ reaches normal form in } m \text{ steps}\}$$

$B$ is recursive, since it checks only for a given number of steps, so by Lemma 169, $\mathsf{K} \in \mathcal{RE}$. □

**Lemma 171.** $\bar{\mathsf{K}} \notin \mathcal{RE}$

*Proof.* Immediate by Lemma 167 and $\mathsf{K} \in \mathcal{RE} \setminus \mathcal{R}$. □

**Lemma 172.** $\bar{\mathsf{K}}' \notin \mathcal{RE}$

*Proof.* Analogous to the previous lemma. □

## 4.6 Reductions

### 4.6.1 Turing Reduction

Sometimes, it is interesting to pretend that in the $\lambda$-calculus some function or set is $\lambda$-definable, even if we do not know if they are, or even if we know they are not. More precisely, we consider a specific function/set and *extend* the $\lambda$-calculus with a *specific* construct to compute/decide that function/set. The overall result is a new language where that function/set is just *forced* to be computable. Clearly, this is a purely theoretical device, since we can not actually build a "computer" which is able to run this extended $\lambda$-calculus. To build that "computer" we would need a "magic" hardware component which enables us to compute the function/set. This component is usually named an "oracle". Even if this construction is a bit bizarre, it is useful to understand the relationships between undecidable sets.

To keep things simple, we just considers sets.

**Definition 173.** *When we extend the $\lambda$-calculus with an oracle for a set $A$, we speak about $(\lambda + A)$-calculus.*

*The syntax of the $(\lambda + A)$-calculus is*

$$M ::= x \mid MM \mid \lambda x.\, M \mid V_A$$

*where $V_A$ is a specific constant. The semantics is given by $=^A_{\beta\eta}$ defined as before, but extended with*

$$\begin{array}{ll} V_A \ulcorner n \urcorner \to^A_\beta \mathbf{T} & \text{when } n \in A \\ V_A \ulcorner n \urcorner \to^A_\beta \mathbf{F} & \text{otherwise} \end{array}$$

*The notion of $(\lambda + A)$-definability is then derived from the notion of $\lambda$-definability by using $=^A_{\beta\eta}$ instead of $=_{\beta\eta}$.*

Here's an important definition for comparing sets, by *reducing* one set to another. Informally, it states that $A$ is no more difficult to decide than $B$.

**Definition 174** (Turing-reduction)**.** *Given $A, B \subseteq \mathbb{N}$, we write $A \leq_T B$ when, the set $A$ can be $(\lambda + B)$-defined. We write $A \equiv_T B$ when $A \leq_T B$ and $B \leq_T A$.*

**Exercise 175.** *Prove the following statements:*

- *$\leq_T$ is a preorder (e.g. is reflexive and transitive)*
- *$A \leq_T B$ for any $A \in \mathcal{R}$, and* any *$B \subseteq \mathbb{N}$*
- *$A \equiv_T \bar{A}$ for all $A$, in particular $\mathsf{K} \equiv_T \bar{\mathsf{K}}$*
- *$\mathsf{K} \equiv_T \mathsf{K}'$*
- *If $A, B \leq_T C$, then $A \cup B \leq_T C$*
- *If $A, B \leq_T C$, then $\{\mathsf{encode}_\times(x, y) \mid x \in A \wedge y \in B\} \leq_T C$*
- *If $A, B \leq_T C$, then $\{\mathsf{inL}(x) \mid x \in A\} \cup \{\mathsf{inR}(x) \mid x \in B\} \leq_T C$*
- *If $A \in \mathcal{R}$ and $B \leq_T A$, then $B \in \mathcal{R}$*
- *From $A \in \mathcal{RE}$ and $B \leq_T A$, we can not conclude that $B \in \mathcal{RE}$ (in general)*

This notion of reduction is useful as it enables us to prove that a set $A$ is *not* $\lambda$-definable, by showing that $B \leq_T A$ for some $B$ that is known to be $\lambda$-*un*definable.

**Exercise 176.** *Consider the $(\lambda + \mathsf{K})$-calculus. Can every partial function $f \in \mathbb{N} \rightsquigarrow \mathbb{N}$ be $(\lambda + \mathsf{K})$-defined?*

### 4.6.2 Many-one Reduction

Another useful notion of reduction is the following:

**Definition 177** (many-one-reduction, a.k.a. m-reduction)**.**
*Given $A, B \subseteq \mathbb{N}$, we write $A \leq_m B$ when there is a* total *recursive function $f$ such that*

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

*We write $A \equiv_m B$ when $A \leq_m B$ and $B \leq_m A$.*

**Lemma 178.** $A \leq_m B \implies A \leq_T B$

*Proof.* Trivial: let $f$ be the total recursive m-reduction from $A$ to $B$. Let $f$ be $\lambda$-defined by $F$. Then $V_A = \lambda n.\, V_B(Fn)$ defines $A$ in the $(\lambda + B)$-calculus. □

**Lemma 179.** $A \leq_m B \iff \bar{A} \leq_m \bar{B}$

*Proof.* Directly from the definition, using the same $f$. □

**Lemma 180.** *If $B \in \mathcal{R}$ and $A \leq_m B$, then $A \in \mathcal{R}$.*
*If $B \in \mathcal{RE}$ and $A \leq_m B$, then $A \in \mathcal{RE}$.*

*Proof.* The first part is a direct consequence of $\leq_m$ implying $\leq_T$, and Ex.175. For the second part, let $B = \mathsf{dom}(\phi_b)$, and $f$ be the m-reduction from $A$ to $B$. Then $g(x) = \phi_b(f(x))$ is a partial recursive function defined only when $f(x) \in B$, i.e. $x \in A$. So, $\mathsf{dom}(g) = A$ and $A \in \mathcal{RE}$. □

**Lemma 181.** $A \leq_m \mathsf{K}' \implies A \in \mathcal{RE}$

*Proof.* Immediate from the lemma above. □

**Exercise 182.** *Prove the following:*

- $\leq_m$ *is a preorder (reflexive and transitive)*
- $A \leq_m B$ *iff* $\bar{A} \leq_m \bar{B}$
- $\mathsf{K} \not\leq_m \bar{\mathsf{K}}$ *and* $\bar{\mathsf{K}} \not\leq_m \mathsf{K}$

**Lemma 183.** $\mathsf{K}'$ *is $\mathcal{RE}$-complete (or m-complete), that is:* $\mathsf{K}' \in \mathcal{RE}$ *and for any $A \in \mathcal{RE}$, $A \leq_m \mathsf{K}'$.*

*Proof.* We have already proved that $\mathsf{K} \in \mathcal{RE}$. For the second part, let $S_A$ be the semi-verifier for $A$, i.e. $A = \mathsf{dom}(\phi_{\#S_A})$. Consider

$$f(n) = \#\left(\lambda x.\, S_A \ulcorner n \urcorner\right)$$

That is, $f(n)$ is returning an index of a program $M$ such that $M$ discards its input, and computes instead $\phi_{\#S_A}(n)$. This $f$ is a total recursive function[1]: let us check that it is an m-reduction from $A$ to $\mathsf{K}$.

$$n \in A \iff \phi_{\#S_A}(n) \text{ defined} \iff \phi_{f(n)}(f(n)) \text{ defined} \iff f(n) \in \mathsf{K}$$

□

[1]This is can be justified either by the appeal to inuition, i.e. by applying the Church's Thesis, or in a slightly more formal way as follows. Consider $g(n, x) = \phi_{\#S_A}(n)$. Clearly, $g \in \mathcal{R}$, so $g = \phi_j$ for some $j$. Taking $f(n) = s_2^1(j, n)$ then suffices.

**Lemma 184.** *$A \in \mathcal{RE}$ if and only if $A \leq_m \mathsf{K}'$*

*Proof.* Immediate from the lemmata above. □

**Exercise 185.** *Prove that $\mathsf{K}' \equiv_m \mathsf{K}$. From this, deduce that $A \in \mathcal{RE}$ if and only if $A \leq_m \mathsf{K}$.*

## 4.7 Rice-Shapiro Theorem

This is a fundamental result.

Recall that, when $f, g$ are partial functions, $g \subseteq f$ means that

$$g(n) = m \in \mathbb{N} \implies f(n) = m$$

That is, when $g(n)$ is defined, $f(n)$ is too, and has the same value $m$. Note that when $g(n)$ is not defined, $f(n)$ may be anything: either undefined, or defined to some $m$.

**Theorem 186** (Rice-Shapiro)**.**
*Let $\mathcal{F}$ be a set of partial recursive functions, i.e. $\mathcal{F} \subseteq \mathcal{R}$. Further, let $A = \{n | \phi_n \in \mathcal{F}\}$ be $\mathcal{RE}$. Then, for each partial recursive function $f$,*

$$f \in \mathcal{F} \iff \exists g \subseteq f.\, \mathsf{dom}(g) \text{ is finite} \wedge g \in \mathcal{F}$$

*Proof.* Also see [Cutland]
Since $f$ is recursive, $f = \phi_i$ for some $i \in \mathbb{N}$.

- ($\Rightarrow$) By contradiction, assume $f \in \mathcal{F}$ but for each finite $g$ s.t. $g \subseteq f$ we have $g \notin \mathcal{F}$. Now, consider

  $$h(n) = \#\left(\lambda x. \begin{cases} f(x) & \text{if } \phi_n(n) \text{ does not halt in } x \text{ steps} \\ undefined & \text{otherwise} \end{cases}\right)$$

  This means that $h(n)$ returns some program index $j$ such that

  $$\phi_j(x) = \begin{cases} f(x) & \text{if running } \phi_n(n) \text{ does not halt in } x \text{ steps} \\ undefined & \text{otherwise} \end{cases}$$

  Note that the function $h$ is well-defined: it is possible to decide whether $\phi_n(n)$ halts in $x$ steps, so $\lambda x. \ldots$ can be indeed formalized as an actual program, and so we can take $j$ as its index.

  The function $h$ is also a total recursive function: we can construct the index of such a $j$ by building the index of $\lambda x. \ldots$ given the parameter $n$.

  Now we note that:

- $n \notin \mathsf{K}' \implies \phi_{h(n)} = f \in \mathcal{F} \implies h(n) \in A$
- $n \in \mathsf{K}' \implies \phi_{h(n)}$ finite and $\phi_{h(n)} \subseteq f \implies \phi_{h(n)} \notin \mathcal{F} \implies h(n) \notin A$

This means that $\bar{\mathsf{K}}' \leq_m A \leq_m \mathsf{K}'$, which is a contradiction.

- ($\Leftarrow$) By contradiction, there is some finite $g \subseteq f$ with $g \in \mathcal{F}$, but $f \notin \mathcal{F}$.

  Now, consider

$$h(n) = \#\left(\lambda x. \begin{cases} f(x) & \text{if } x \in \mathsf{dom}(g) \text{ or } n \in \mathsf{K}' \\ undefined & \text{otherwise} \end{cases}\right)$$

  Again, such a program $\lambda x. \ldots$ indeed exists: the domain of $g$ is finite, so we can check for that. The other condition asks whether $n \in \mathsf{K}'$, and we can only semi-decide that. However, that is enough since otherwise we must not terminate.

  Again, $h(n)$ is a total recursive function: the index of $\lambda x. \ldots$ can be effectively computed given the number $n$.

  Finally, we have that:

  - $n \notin \mathsf{K}' \implies \phi_{h(n)} = g \in \mathcal{F} \implies h(n) \in A$
  - $n \in \mathsf{K}' \implies \phi_{h(n)} = f \notin \mathcal{F} \implies h(n) \notin A$

  This means that $\bar{\mathsf{K}}' \leq_m A \leq_m \mathsf{K}'$, which is a contradiction.

□

## 4.8 Rice's Theorem

**Theorem 187.** *Let $\mathcal{F} \subseteq (\mathbb{N} \rightsquigarrow \mathbb{N})$ be a set of partial recursive functions, and $A = \{n | \phi_n \in \mathcal{F}\}$. If $A \in \mathcal{R}$, then $A$ is trivial, i.e. either $A = \emptyset$ or $A = \mathbb{N}$.*

*Proof.* We could prove Rice's Theorem by adapting Th. 107, but we instead apply Rice-Shapiro (Th. 186). We have $A, \bar{A} \in \mathcal{R}$, so $A, \bar{A} \in \mathcal{RE}$. Let $\phi_i$ be the always-undefined function $g_\emptyset$, that has a finite domain. For all partial functions $f$, we have $g_\emptyset \subseteq f$. Clearly, $i$ is in one of the sets $A, \bar{A}$.

- If $i \in A$, then $g_\emptyset \in \mathcal{F}$. By Rice-Shapiro, we have $f \in \mathcal{F}$ for all $f$, and so $A = \mathbb{N}$.

- If $i \in \bar{A}$, then $g_\emptyset \notin \mathcal{F}$. By Rice-Shapiro, we have $f \notin \mathcal{F}$ for all $f$, and so $\bar{A} = \mathbb{N}$, implying $A = \emptyset$.

□

Also see [Cutland]

**Exercise 188.** *For any of these sets, state whether the set is $\mathcal{R}$, or $\mathcal{RE} \setminus \mathcal{R}$, or not in $\mathcal{RE}$.*

- $\mathsf{K} \cup \{5\}$
- $\{1, 2, 3, 4\}$
- $\{n | \phi_n(2) = 6\}$
- $\{n | \exists y \in \mathbb{N}. \phi_n(y) = 6\}$
- $\{n | \forall y \in \mathbb{N}. \phi_n(y) = 6\}$
- $\{n | \phi_n(n) = 6\}$
- $\{n | \mathsf{dom}(\phi_n)$ *is finite*$\}$
- $\{n | \mathsf{dom}(\phi_n)$ *is infinite*$\}$
- $\{n | \phi_n$ *is total*$\}$
- $\{n + 4 | \mathsf{dom}(\phi_n)$ *is finite*$\}$
- $\{ \lfloor 100/(n+1)^2 \rfloor \mid \mathsf{dom}(\phi_n)$ *is infinite*$\}$
- $A \cup B$, $A \cap B$, $A \setminus B$ *where* $A, B \in \mathcal{RE}$
- $A \cup B$, $A \cap B$, $A \setminus B$ *where* $A \in \mathcal{RE}$, $B \in \mathcal{R}$
- $\{\mathsf{inL}(n) \mid n \in A\} \cup \{\mathsf{inR}(n) \mid n \in B\}$ *where* $A, B \in \mathcal{RE}$
- $\{n \mid \forall m. \mathsf{encode}_\times(m, n) \in A\}$ *where* $A \in \mathcal{RE}$
- $\{\mathsf{encode}_\times(n, m) | \mathsf{encode}_\times(m, n) \in A\}$ *where* $A \in \mathcal{RE}$
- $\{f(n) | n \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$, $f$ *total*
- $\{f(n) | n \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$ *(may be non total)*
- $\{n | f(n) \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$, $f$ *total*

- $\{n | f(n) \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$ *(may be non total)*

**Exercise 189.** *(Hard) Show that* $f \in \mathcal{R}$ *where*

$$f(n) = \begin{cases} k & \text{if running } \phi_n(n) \text{ halts in } k \text{ steps} \\ undefined & \text{otherwise} \end{cases}$$

*Then show that there is no total recursive* $g$ *such that* $f \subseteq g$.
*Finally, show that* $\{n | \exists i.\, \phi_n \subseteq \phi_i \wedge \phi_i \text{ total}\} \notin \mathcal{RE}$.

**Exercise 190.** *Let* $A \in \mathcal{RE}$. *Define* $B = \{n | \exists m \in A.\, n < m\}$. *Can we deduce* $B \in \mathcal{RE}$ *? What about* $C = \{n | \forall m \in A.\, n < m\}$ *?*

**Exercise 191.** *Given a* $\lambda$*-term* $L$ *and a Java program* $J$*, let* $\phi_{\#L}$ *and* $\varphi_{\#J}$ *be the respective semantics, as functions* $\mathbb{N} \rightsquigarrow \mathbb{N}$ *(assume* $\#J$ *to be the index of the Java program* $J$*, defined using the usual encoding functions). Then, consider* $A = \{\mathsf{encode}_\times(\#L, \#J) \mid \phi_{\#L} = \varphi_{\#J}\}$. *Is* $A \in \mathcal{R}$*? Is it* $\mathcal{RE}$*?*

**Exercise 192.** *Let* $A \in \mathcal{R}$, *and* $B = \{n \mid \exists m.\, \mathsf{encode}_\times(n, m) \in A\}$. *We know that* $B \in \mathcal{RE}$ *by Lemma 169. Can we conclude that* $B \in \mathcal{RE} \setminus \mathcal{R}$ *?*

**Exercise 193.** *Consider the following formal language: (*$a, b$ *are two constants)*

$$X := a \mid b \mid (XX)$$

*and an equational semantics* $=_\gamma$ *given by*

$$\begin{array}{l} ((aX)Y) =_\gamma X \\ (((bX)Y)Z) =_\gamma ((XZ)(YZ)) \\ (XY) =_\gamma (X'Y') \quad \text{when } X =_\gamma X' \text{ and } Y =_\gamma Y' \\ =_\gamma \text{ is transitive, symmetric, reflexive} \end{array}$$

*Define* $\#X$ *as the index of* $X$ *using the usual encoding functions. Discuss whether you expect the sets below to be in* $\mathcal{R}$*,* $\mathcal{RE} \setminus \mathcal{R}$*, or not in* $\mathcal{RE}$*, justifying your assertions. (Note: I do not expect a real proof, but correct arguments.)*

- $\{\#X \mid X =_\gamma a\}$
- $\{\mathsf{encode}_\times(\#X, \#Y) \mid X \neq_\gamma Y\}$

# Appendix A

# Solutions

**Solution 194.** *We want a* $\mathbf{\Theta}$ *such that* $\mathbf{\Theta}F = F(\mathbf{\Theta}F)$. *We first write that as* $\mathbf{\Theta} = \lambda F.\, F(\mathbf{\Theta}F)$. *Then, we abstract the* $\mathbf{\Theta}$ *recursive call, obtaining*

$$M = \lambda w.\, \lambda F.\, F(wF)$$

*Then we duplicate the* $w$ *inside*

$$M = \lambda w.\, \lambda F.\, F(wwF)$$

*And finally, we let* $\mathbf{\Theta} = MM$, *that is*

$$\mathbf{\Theta} = (\lambda w.\, \lambda F.\, F(wwF))(\lambda w.\, \lambda F.\, F(wwF))$$

*We are done. Let us check* $\mathbf{\Theta}$ *is really a fixed point combinator.*

$$\begin{aligned}
\mathbf{\Theta}F &= (\lambda w.\, \lambda F.\, F(wwF))(\lambda w.\, \lambda F.\, F(wwF))F \\
&= \Big(\lambda F.\, F\big((\lambda w.\, \lambda F.\, F(wwF))(\lambda w.\, \lambda F.\, F(wwF))F\big)\Big)F \\
&= \big(\lambda F.\, F(\mathbf{\Theta}F)\big)F \\
&= F(\mathbf{\Theta}F)
\end{aligned}$$

*The* $\mathbf{\Theta}$ *above was given by Church. Turing discovered this other fixed point combinator*

$$\mathbf{Y} = \lambda F.\, MM \quad \textit{where} \quad M = \lambda w.F(ww)$$

*There other ones, of course. The* £ *below is a peculiar one given by Klop.*

$$\$ = £££££££££££££££££££££££££££££$$
$$£ = \lambda abcdefghijklmnopqstuvwxyzr.\, r(thisisafixedpointcombinator)$$

**Solution 195.** *By contradiction, assume* $\mathbf{T} =_{\beta\eta} \mathbf{F}$*. Clearly,* $\mathbf{T}$ *and* $\mathbf{F}$ *are* $\beta\eta$*-normal forms. By Th. 52, we have* $\mathbf{T} \twoheadrightarrow_{\beta\eta}^{*} \mathbf{F}$*. Since* $\mathbf{T}$ *is a normal form, this is not possible unless* $\mathbf{T} =_{\alpha} \mathbf{F}$*, which is clearly not the case* □

**Solution 196.** *Here are many useful combinators:*

$$\mathbf{And} = \lambda xy.\, xy\mathbf{F}$$
$$\mathbf{Or} = \lambda xy.\, x\mathbf{T}y$$
$$\mathbf{Not} = \lambda x.\, x\mathbf{FT}$$
$$\mathbf{Leq} = \lambda nm.\, \mathbf{IsZero}\,(m\,\mathbf{Pred}\,n)$$
$$\mathbf{Eq} = \lambda nm.\, \mathbf{And}(\mathbf{Leq}\,n\,m)(\mathbf{Leq}\,m\,n)$$
$$\mathbf{Lt} = \lambda nm.\, \mathbf{Leq}(\mathbf{Succ}\,n)m$$
$$\mathbf{Add} = \lambda nm.\, n\,\mathbf{Succ}\,m$$
$$\mathbf{Mul} = \lambda nm.\, n(\mathbf{Add}\,m)\mathbf{0}$$
$$\mathbf{Even} = \lambda n.\, n\,\mathbf{Not}\,\mathbf{T}$$

$\mathbf{LimMin}\,F\,Z\ulcorner n\urcorner$ *returns the smallest* $m \in \{0..n\}$ *such that* $F\ulcorner m\urcorner = \mathbf{T}$*. If no such* $m$ *exists, it returns* $Z$*. Note that* $F$ *must return only* $\mathbf{T}$ *or* $\mathbf{F}$ *for this to work.*

$$\mathbf{LimMin} = \lambda fzn.\, \mathbf{Succ}\,n\,(\lambda gx.fxx(g(\mathbf{Succ}\,x)))(\mathbf{K}z)\,\mathbf{0}$$
$$\mathbf{Any} = \lambda fn.\, \mathbf{Leq}(\mathbf{LimMin}\,f\,(\mathbf{Succ}\,n)\,n)\,n$$
$$\mathbf{All} = \lambda fn.\, \mathbf{Not}(\mathbf{Any}(\circ\,\mathbf{Not}\,f)n)$$

*The following is integer division:* $\mathbf{Div}\ulcorner n\urcorner\ulcorner m\urcorner = \ulcorner \lfloor n/m \rfloor \urcorner$

$$\mathbf{Div} = \lambda nm.\, \mathbf{LimMin}\,(\lambda x.\, \mathbf{Lt}\,n(\mathbf{Mul}(\mathbf{Succ}\,x)m))\Omega n$$

*The following is the* $\lambda$*-term defining the* $\mathsf{encode}_{\times}$ *function. The definition is straightforward from the formula for* $\mathsf{encode}_{\times}$*.*

$$\mathbf{Pair} = \lambda nm.\, \mathbf{Add}(\mathbf{Div}\,(\mathbf{Mul}(\mathbf{Add}\,n\,m)(\mathbf{Succ}\,(\mathbf{Add}\,n\,m)))\ulcorner 2\urcorner)n$$

*We compute the inverse of* $c = \mathsf{encode}_{\times}(n, m)$ *by "brute force". We merely try all the possible values of* $n, m$*, encode them, and stop when we find the unique* $n, m$ *pair which has* $c$ *as its encoding. By Lemma 22, we only need to search for* $n, m \in \{0..c\}$*, so we limit our search to that square.*

$$\mathbf{Proj1} = \lambda c.\, \mathbf{LimMin}\,(\lambda n.\, \mathbf{Any}(\lambda m.\, \mathbf{Eq}\,c\,(\mathbf{Pair}\,n\,m))c)\Omega\,c$$
$$\mathbf{Proj2} = \lambda c.\, \mathbf{LimMin}\,(\lambda m.\, \mathbf{Any}(\lambda n.\, \mathbf{Eq}\,c\,(\mathbf{Pair}\,n\,m))c)\Omega\,c$$

$$\mathbf{InL} = \lambda n.\, \mathbf{Mul}\, \ulcorner 2 \urcorner\, n$$
$$\mathbf{InR} = \lambda n.\, \mathbf{Succ}(\mathbf{Mul}\, \ulcorner 2 \urcorner\, n)$$
$$\mathbf{Case} = \lambda nlr.\, \mathbf{Even}\, n\, (l(\mathbf{Div}\, n\, \ulcorner 2 \urcorner))\, (r(\mathbf{Div}\, n\, \ulcorner 2 \urcorner))$$

*Above, when $n$ is odd, we compute $(n-1)/2$ by just using* $\mathbf{Div}\ulcorner n \urcorner \ulcorner 2 \urcorner = \ulcorner \lfloor n/2 \rfloor \urcorner = \ulcorner (n-1)/2 \urcorner$. *We could also apply* Pred *to $n$, leading to the same result.*

**Solution 197.**

$$\mathbf{Length} = \mathbf{\Theta}(\lambda gl.\, \mathbf{Eq}\, \mathbf{0}\, (\mathbf{Fst}\, l) \mathbf{0}\, (\mathbf{Succ}(g(\mathbf{Snd}\, l))))$$
$$\mathbf{Merge} = \mathbf{\Theta}(\lambda gab.\, \mathbf{Eq}\, \mathbf{0}\, (\mathbf{Fst}\, a)\, b\, (\mathbf{Eq}\, \mathbf{0}\, (\mathbf{Fst}\, b)\, a\, A))$$
$$A = \mathbf{Leq}\, (\mathbf{Fst}\, a)\, (\mathbf{Fst}\, b)\, B\, C$$
$$B = \mathbf{Cons}\, (\mathbf{Fst}\, a)\, (g\, (\mathbf{Snd}\, a)\, b)$$
$$C = \mathbf{Cons}\, (\mathbf{Fst}\, b)\, (g\, a\, (\mathbf{Snd}\, b))$$
$$\mathbf{Split} = \mathbf{\Theta}(\lambda ga.\, \mathbf{Eq}\, \mathbf{0}\, A_1\, (\mathbf{Cons}\, a\, a)\, (\mathbf{Cons}\, (\mathbf{Cons}\, A_1\, B_2)\, B_1))$$
$$B_1 = \mathbf{Fst}\, (g\, A_r)$$
$$B_2 = \mathbf{Snd}\, (g\, A_r)$$
$$A_1 = \mathbf{Fst}\, a$$
$$A_r = \mathbf{Snd}\, a$$
$$\mathbf{MergeSort} = \mathbf{\Theta}(\lambda ga.\, \mathbf{Eq}\, \mathbf{0}\, A_1\, a\, (\mathbf{Eq}\, \mathbf{0}\, A_2\, a\, M))$$
$$M = \mathbf{Merge}\, (g\, (\mathbf{Fst}\, (\mathbf{Split}\, a)))\, (g\, (\mathbf{Snd}\, (\mathbf{Split}\, a)))$$
$$A_1 = \mathbf{Fst}\, a$$
$$A_2 = \mathbf{Fst}\, (\mathbf{Snd}\, a)$$

**Solution 198.**

- $G \ulcorner M \urcorner = \ulcorner MM \urcorner$

$$G = \lambda m.\, \mathbf{App}\, m\, m$$

- $G \ulcorner MN \urcorner = \ulcorner NM \urcorner$

$$G = \lambda x.\, \mathbf{Case}\, x\, \Omega\, (\lambda y.\, \mathbf{Case}\, y\, (\lambda z.\, \mathbf{InR}\, (\mathbf{InL}\, (\mathbf{Pair}\, (\mathbf{Proj2}\, z)\, (\mathbf{Proj1}\, z))))\, \Omega)$$

- $G \ulcorner \lambda x.\, M \urcorner = \ulcorner M \urcorner$

$$G = \lambda x.\, \mathbf{Case}\, x\, \Omega\, (\lambda y.\, \mathbf{Case}\, y\, \Omega\, (\lambda z.\, \mathbf{Proj2}\, z))$$

- $G\ulcorner\lambda x.\,\lambda y.\,M\urcorner = \ulcorner\lambda y.\,\lambda x.\,M\urcorner$
  *This is rather complex:*

  $$G = \lambda x.\,\mathbf{Case}\,x\,\Omega\,(\lambda y.\,\mathbf{Case}\,y\,\Omega\,(\lambda z.\,A\,(\mathbf{Proj2}\,z)))$$
  $$A = \lambda x'.\,\mathbf{Case}\,x'\,\Omega\,(\lambda y'.\,\mathbf{Case}\,y'\,\Omega\,(\lambda z'.\,B))$$
  $$B = \mathbf{InR}\,(\mathbf{InR}\,(\mathbf{Pair}\,(\mathbf{Proj1}\,z')\,(\mathbf{InR}\,(\mathbf{InR}\,(\mathbf{Pair}\,(\mathbf{Proj1}\,z)\,(\mathbf{Proj2}\,z'))))))$$

- $G\ulcorner\mathbf{I}M\urcorner = \ulcorner M\urcorner$ *and* $G\ulcorner\mathbf{K}M\urcorner = \ulcorner\mathbf{I}\urcorner$

  $$G = \lambda x.\,\mathbf{Case}\,x\,\Omega\,(\lambda y.\,\mathbf{Case}\,y\,(\lambda z.\,\mathbf{Eq}(\mathbf{Proj1}\,z)\ulcorner\mathbf{I}\urcorner(\mathbf{Proj2}\,z)\ulcorner\mathbf{I}\urcorner)\,\Omega)$$

- $G\ulcorner\lambda x_i.\,M\urcorner = \ulcorner\lambda x_{i+1}.\,M\urcorner$

  $$G = \lambda x.\,\mathbf{Case}\,x\,\Omega\,(\lambda y.\,\mathbf{Case}\,y\,\Omega\,(\lambda z.\,\mathbf{InR}\,(\mathbf{InR}\,(\mathbf{Pair}\,(\mathbf{Succ}(\mathbf{Proj1}z))\,(\mathbf{Proj2}z)))))$$

- $G\ulcorner M\urcorner = \ulcorner N\urcorner$ *where* $N$ *is obtained from* $M$ *replacing every (bound or free) variable* $x_i$ *with* $x_{i+1}$

  $$G = \mathbf{\Theta}(\lambda gx.\mathbf{Case}\,x\,(\lambda i.\,\mathbf{InL}(\mathbf{Succ}\,i))(\lambda y.\,\mathbf{Case}\,y\,A\,B))$$
  $$A = \lambda z.\,\mathbf{InR}\,(\mathbf{InL}\,(\mathbf{Pair}\,(g(\mathbf{Proj1}\,z))\,(g(\mathbf{Proj2}\,z))))$$
  $$B = \lambda z.\,\mathbf{InR}\,(\mathbf{InR}\,(\mathbf{Pair}\,(\mathbf{Succ}(\mathbf{Proj1}\,z))\,(g(\mathbf{Proj2}\,z))))$$

- $G\ulcorner M\urcorner = \ulcorner M\{\mathbf{I}/x_0\}\urcorner$ *(this does not require* $\alpha$*-conversion)*

  $$G = \mathbf{\Theta}(\lambda gx.\mathbf{Case}\,x\,A(\mathbf{Case}\,y\,B\,C))$$
  $$A = \lambda i.\,\mathbf{Eq}\,i\,\mathbf{0}\ulcorner\mathbf{I}\urcorner\,(\mathbf{InL}\,i)$$
  $$B = \lambda z.\,\mathbf{InR}\,(\mathbf{InL}\,(\mathbf{Pair}\,(g(\mathbf{Proj1}\,z))\,(g(\mathbf{Proj2}\,z))))$$
  $$C = \lambda z.\,\mathbf{Eq}\,(\mathbf{Proj1}\,z)\,\mathbf{0}\,z\,(\mathbf{InR}\,(\mathbf{InR}\,(\mathbf{Pair}\,(\mathbf{Proj1}\,z)\,(g(\mathbf{Proj2}\,z)))))$$

**Solution 199.** *By contradiction, suppose* $\mathsf{K}_0$ *is* $\lambda$*-defined by* $F$*. Then, we consider*

$$G = \lambda x.\,F(\mathbf{App}\ulcorner\mathbf{K}\urcorner\,(\mathbf{App}\,x(\mathbf{Num}\,x)))$$

*We have that* $G\ulcorner M\urcorner = F\ulcorner\mathbf{K}(M\ulcorner M\urcorner)\urcorner$*. The latter evaluates to* $\mathbf{T}$ *of* $\mathbf{F}$ *depending on whether* $\mathbf{K}(M\ulcorner M\urcorner)\mathbf{0} = M\ulcorner M\urcorner$ *has a normal form. So* $G$ *actually* $\lambda$*-defines* $\mathsf{K}$*, which is a contradiction.* □

**Solution 200.** *Take* $\mathsf{Pad} = \lambda n.\,\mathsf{App}\ulcorner\mathbf{I}\urcorner n$*. Then,* $\mathsf{Pad}\ulcorner M\urcorner = \ulcorner\mathbf{I}M\urcorner$*, and we have*

$$\#(\mathbf{I}M) = 1 + 2\cdot(2\cdot(\tfrac{(\#\mathbf{I}+\#M)(\#\mathbf{I}+\#M+1)}{2} + \#\mathbf{I})) \geq$$
$$\geq 1 + 4\cdot\tfrac{\#M}{2} > \#M$$