Short Lecture Notes — Computability (2008–2010)

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

Preliminary Version — 15 Dec 2010

General Information

These notes are meant to be a short summary of the topics covered in my *Computability* course kept in 2008 and 2010 in 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.
- You might still want to refer to the books for some parts. I will try to provide suitable references in the notes.
- While I tried to include all the relevant technical definitions and results in these notes, at the moment there is almost no discussion about *what* is computability and *why* we want to study it.
- Reporting errors in these notes will be awarded.

In the margins of these notes, you will find some markers for those definitions, statements and proofs which will be asked during the oral exam. For example:

- **Statement** This is a statement you need to know for the exam. You will *not* be asked to prove it, but you may be asked to apply it to some concrete case, or otherwise to prove you understand it.
- **Proof** This is a statement you need to know for the exam. You can be asked to provide a *proof* for it (such a proof is included in these notes).

Also, please remember the following, taken from ESSE3:

> *Prerequisites*: strong basic notions of set theory; strong formula-handling skills; good programming skills.

When relevant to the discussion during the oral test, you may be asked about the above topics even when not explicitly marked in the notes.

Roberto Zunino

Contents

Chapter 1

Basics

In this chapter we shall recall some preliminary facts which we shall use in the rest of the course. Most proofs here are left as an exercise to the reader: you should be able to do this with a moderate effort. Moreover, you should test your formula-understanding skills by performing most exercises in this section.

1.1 Logic Notation

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

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

$p \lor \neg p$	*excluded middle*
$\neg(p \lor q) \iff (\neg p \land \neg q)$	*De Morgan*
$\neg(p \land q) \iff (\neg p \lor \neg q)$	*De Morgan*
$(p \implies q) \iff (\neg p \lor q)$	*classical implication*
$(p \land q \implies r) \iff (p \implies (q \implies r))$	*export/import*
$(p \implies q) \iff (\neg q \implies \neg p)$	*contraposition*
$(p \iff q) \iff (\neg p \iff \neg q)$	*contraposition*
$(p \land q) \lor r \iff (p \lor r) \land (q \lor r)$	*distribution*
$(p \lor q) \land r \iff (p \land r) \lor (q \land r)$	*distribution*
$(\neg \forall x.\, p(x)) \iff (\exists x.\, \neg p(x))$	*De Morgan*
$(\neg \exists x.\, p(x)) \iff (\forall x.\, \neg p(x))$	*De Morgan*

$(p \wedge (\forall x.\, q(x))) \iff (\forall x.\, p \wedge q(x))$	*scope extrusion (x not in p)*
$(p \vee (\forall x.\, q(x))) \iff (\forall x.\, p \vee q(x))$	*scope extrusion (x not in p)*
$(p \wedge (\exists x.\, q(x))) \iff (\exists x.\, p \wedge q(x))$	*scope extrusion (x not in p)*
$(p \vee (\exists x.\, q(x))) \iff (\exists x.\, p \vee q(x))$	*scope extrusion (x not in p)*
$(p \implies (\forall x.\, q(x))) \iff (\forall x.\, (p \implies q(x)))$	*scope extrusion (x not in p)*
$(p \implies (\exists x.\, q(x))) \iff (\exists x.\, (p \implies q(x)))$	*scope extrusion (x not in p)*
$((\forall x.\, p(x)) \implies q) \iff (\exists x.\, (p(x) \implies q))$	*scope extrusion (x not in q)*
$((\exists x.\, p(x)) \implies q) \iff (\forall x.\, (p(x) \implies q))$	*scope extrusion (x not in q)*
$\exists y.\forall x.\, p(x,y) \implies \forall x.\exists y.\, p(x,y)$	
$\forall x.\exists y.\, p(x,y) \not\Longrightarrow \exists y.\forall x.\, p(x,y)$	
$\exists! x.\, p(x) \iff \exists c.\, (\forall x.\, (p(x) \iff x = c))$	*uniqueness*
$\exists! x.\, p(x) \iff (\exists x.\, p(x)) \wedge (\forall x, y.\, (p(x) \wedge p(y) \implies x = y))$	

Exercise 2. *Convince yourself that the formulas above indeed hold.*

1.2 Set Theory

Let $A, B, \ldots, X, Y, Z$ be sets. Below, we provide standard definitions and examples. I recommend you read them and check they match with your intuition.

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

$$\begin{aligned} A \setminus B &= \{x | x \in A \land x \notin B\} \\ X \subseteq Y &\iff \forall x \in X.\, x \in Y \\ \mathcal{P}(A) &= \{B | B \subseteq A\} \end{aligned}$$

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

$$\begin{aligned} \langle x, y\rangle = \langle x', y'\rangle &\iff (x = x' \land y = y') \\ X \times Y &= \{\langle x, y\rangle | x \in X \land y \in Y\} \end{aligned}$$

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

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

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 (or one-to-one) 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*

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

Definition

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 *(or "onto") when $\mathsf{ran}(f) = B$. An injective and surjective function is said to be* bijective *(or a* bijection*, or a one-to-one correspondence).* □

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*.

Definition 10. *The composition of two partial functions f, g is defined as*

$$(f \circ g)(x) = f(g(x))$$

Note that, whenever $g(x)$ is undefined, so is $f(g(x))$.

Exercise 11. *Let A, B, C be sets, and let $g \in (A \to B)$ and $f \in (B \to C)$. Prove that*

- *If f and g are injective, then $f \circ g$ is injective.*
- *If f and g are surjective, then $f \circ g$ is surjective.*
- *If f and g are bijections, then $f \circ g$ is a bijection.*

1.2.1 Further Notation

Definition

For this course, we shall use

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

The (total) function χ_A is called the *characteristic function* of the set A. Similarly, the partial function $\tilde{\chi}_A$ is called the *semi-characteristic function* of A.

1.3 Induction

Many concepts in computer science (and mathematics) are defined through some sort of inductive definition. Similarly, many useful properties are often proved by exploiting some induction principle.

In this section, we survey some different, yet equivalent, ways to present an inductive definition. Students which have no or little background on these topics may find some of these hard to understand at the beginning. Also note that a deep understanding of these is not strictly necessary for the rest of the course[1]. As a guideline, as long as you are able to solve Ex. 15 below, you should be able to understand every other use of induction in these notes.

Below, we provide an inductive definition for the set of natural numbers $\mathbb{N}$. This is done in several different ways, so that the reader can get used to all of these. Some informal argument supporting the fact that these definition indeed match our intuitive notion of $\mathbb{N}$ is provided.

Definition 12. *The set of natural numbers $\mathbb{N}$ can be equivalently defined as follows:*

- *(Informal definition)* $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$

[1] In spite of this, it is my opinion that each graduating Computer Science student should be rather knowledgeable with induction techniques, as these play such a huge rôle in our discipline.

- *(Through inductive inference rules) We let $\mathbb{N}$ be the set of those elements that can be generated by the following inference rules: (below, s is a symbol for the successor function "+1")*

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

  <u>*Intuition*</u>*: the rules above can generate only natural numbers since we can only use 0 and the successor function s; vice versa, any natural number n can be constructed by starting with the first rule and then applying the second one n times.*

- *(Through the so-called "least prefixed-point" property) Let $\hat{R}$ be the following function:*

$$\hat{R}(X) = \{0\} \cup \{s(n) | n \in X\}$$

  That, we let $\mathbb{N}$ be the least of the prefixed points of $\hat{R}$, i.e.

$$\mathbb{N} = \bigcap \{X | \hat{R}(X) \subseteq X\} \qquad \wedge \qquad \hat{R}(\mathbb{N}) \subseteq \mathbb{N}$$

  <u>*Intuition*</u>*: the function $\hat{R}(X)$ applies the inference rules above* once *to the elements of X. Hence, $\mathbb{N}$ is the least set that is closed under application of $\hat{R}$.*

- *(Through the so-called "least fixed-point" property) Let $\hat{R}$ as above. Then, $\mathbb{N}$ is the least of the fixed points of $\hat{R}$, i.e.*

$$\mathbb{N} = \bigcap \{X | \hat{R}(X) = X\} \qquad \wedge \qquad \hat{R}(\mathbb{N}) = \mathbb{N}$$

  <u>*Intuition*</u>*: $\mathbb{N}$ is the least set that is unaffected by the application of $\hat{R}$.*

- *(As a limit of an increasing chain) Let $\hat{R}$ as above, and write $\hat{R}^n(X)$ for the result of applying n times the function $\hat{R}$ to X. That is, $\hat{R}^0(X) = X$, $\hat{R}^1(X) = \hat{R}(X)$, $\hat{R}^2(X) = \hat{R}(\hat{R}(X))$, and so on. Then,*

$$\mathbb{N} = \bigcup_{n \geq 0} \hat{R}^n(\emptyset)$$

  <u>*Intuition*</u>*: we have $\hat{R}^0(\emptyset) = \emptyset$, $\hat{R}^1(\emptyset) = \{0\}$, $\hat{R}^2(\emptyset) = \{0, 1\}$,… $\hat{R}^n(\emptyset) = \{0, 1, 2, \ldots, n-1\}$. The union of all these sets is clearly $\mathbb{N}$.*

- *(As a recursive set-theoretic equation) Let* $\mathbf{1}$ *below denote a singleton set, e.g.* $\mathbf{1} = \{0\}$*). We let* $\mathbb{N}$ *to be the least solution of the equation*

  $$X \simeq \mathbf{1} \uplus X$$

  Intuition: we have $\mathbb{N} \simeq \mathbf{1} \uplus (\mathbf{1} \uplus (\mathbf{1} \uplus \cdots$*, so this equation is roughly "generating" a sequence of distinct terms, which represent the natural numbers.*

The (non-trivial) equivalence of the definitions above is a consequence of the *Knaster-Tarski* theorem, which is one of the most important foundational theorems in computer science. It is usually discussed when studying the formal semantics of programming languages.

We can rephrase the "prefixed-point" definition of $\mathbb{N}$ as follows:

$$\mathbb{N} = \bigcap \{X | 0 \in X \land \forall m.\, m \in X \implies s(m) \in X\}$$

This allows us to state the usual induction principle on $\mathbb{N}$:

Theorem 13 (Induction Principle)**.** *Given a predicate* p *on* $\mathbb{N}$*, we have* $\forall n \in \mathbb{N}.\, p(n)$ *iff both of these hold*

$$\begin{aligned} & p(0) \\ & \forall m \in \mathbb{N}. \quad p(m) \implies p(m+1) \end{aligned}$$

Proof. The ($\Rightarrow$) direction is trivial.

For the ($\Leftarrow$) direction, we take $Y = \{n \in \mathbb{N} | p(n)\}$ and show $Y = \mathbb{N}$, proving the thesis $\forall n \in \mathbb{N}.\ p(n)$. By definition of Y, $Y \subseteq \mathbb{N}$ is immediate. By hypothesis, we have

$$\begin{aligned} & 0 \in Y \\ & \forall m \in \mathbb{N}. \quad m \in Y \implies m+1 \in Y \end{aligned}$$

the above implies

$$Y \in \{X | 0 \in X \land \forall m.\, m \in X \implies s(m) \in X\}$$

which together with

$$\mathbb{N} = \bigcap \{X | 0 \in X \land \forall m.\, m \in X \implies s(m) \in X\}$$

implies $Y \subseteq \mathbb{N}$. ☐

Exercise 14. *Prove* $\forall n \in \mathbb{N}. 0 + 1 + 2 + \cdots + n = \frac{n\cdot(n+1)}{2}$.

Note how the induction principle (Th.13) closely matches the inductive inference rules,

Consider the above equation

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

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

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

Indeed, grammars are a kind of inductive definitions.

Exercise 15. *Starting from the grammar of binary trees (of naturals)*

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

rewrite the above definition using inference rules. Then, further rewrite it as a recursive set-theoretic equation. You can use $\mathbb{N}, \times, \uplus$ *for the latter.*

Exercise 16. *Express the set* T *of Ex. 15 using* $\bigcap$*:*

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

Exercise 17. *(For logically minded people)*
Write an induction principle for $\mathbb{T}$*.*

Exercise 18. *Define* A^**, the set of finite sequences (i.e. strings) of elements of the set* A *using an inductive definition.*

Exercise 19. *Consider the set of natural numbers* A *defined by the inductive rules below.*

$$\frac{}{6} \qquad \frac{n \quad m}{n+m} \qquad \frac{n \quad m}{n \cdot m} \qquad \frac{n \quad m}{n^m}$$

State an induction principle for this set, in the spirit of Th. 13. Then use it to prove that every number in A *is an even natural number.*

An important set of inductive rules is the following one, which is used in defining equivalence relations.

Definition 20. *The* equivalence relation *inductive rules for a relation* R *are the following:*

$$\frac{}{x \; R \; x} \qquad \frac{x \; R \; y}{y \; R \; x} \qquad \frac{x \; R \; y \quad y \; R \; z}{x \; R \; z}$$

1.4 Cardinality

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

Disjoint Union We now construct a bijection between $\mathbb{N} \uplus \mathbb{N}$ and $\mathbb{N}$. The set $\mathbb{N} \uplus \mathbb{N}$ is intuitively composed of two parts: the "left $\mathbb{N}$" and the "right $\mathbb{N}$". We define two functions, named inL ("in-left") and inR ("in-right") which map the left/right parts into the set of even and odd naturals, respectively. Then we construct the wanted bijection $\mathsf{encode}_{\uplus}$ exploiting these auxiliary functions.

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

Definition

Exercise 21. *Prove that this is a bijection. (Check that it is injective and surjective)*

Exercise 22. *Write the inverse function $\mathbb{N} \to \mathbb{N} \uplus \mathbb{N}$. See Sol. 223.*

Cartesian Product We now provide a bijection $(\mathbb{N} \times \mathbb{N}) \leftrightarrow \mathbb{N}$: this is the so-called "dovetail" function.

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

Definition

	m=0	1	2	3	4	5	6	7
n=0	0	1	3	6	10	15	21	28
1	2	4	7	11	16	22	29	
2	5	8	12	17	23	30		
3	9	13	18	24	...			
4	14	19	25	...				
5	20	26	...					
6	27	...						
7	...							

Exercise 23. *Describe the inverse function $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$. This is usually seen as* two *projection functions* $\mathsf{proj1}$ *and* $\mathsf{proj2}$.

Exercise 24. *Construct a bijection* $(\mathbb{N} \uplus (\mathbb{N} \times \mathbb{N})) \leftrightarrow ((\mathbb{N} \uplus \mathbb{N}) \times \mathbb{N})$. *Do not re-invent everything from scratch, but exploit previous results instead.*

Theorem 25. *There a bijection between* $\mathbb{N}$ *and* $\mathbb{N}^+$ *(the set of finite non-empty sequences of naturals).*

Proof. Left as an exercise. First, provide an inductive definition for $\mathbb{N}^+$. Then, define the bijection inductively. □

Exercise 26. *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 27. *Define a bijection between* $\mathbb{N}$ *and* $\mathbb{Q}$.

Exercise 28. *Prove that*

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

Exercise 29. *Prove that* pair *is monotonic on both arguments, that is:*

$$\forall x, x', y, y'.\ x \leq x' \land y \leq y' \implies \mathsf{pair}(\langle x, y\rangle) \leq \mathsf{pair}(\langle x', y'\rangle)$$

Lemma 30.

$$\mathsf{pair}(\langle n, m\rangle) \geq n$$
$$\mathsf{pair}(\langle n, m\rangle) \geq m$$

Statement

Proof. The first part is trivial:

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

For the second part

$$\begin{aligned}\mathsf{pair}(\langle n, m\rangle) &= \frac{(n+m)(n+m+1)}{2} + n \geq \frac{(n+m)(n+m+1)}{2} = \\ &= \frac{n^2+m^2+2nm+n+m}{2} \geq \frac{m^2+m}{2} \geq \frac{m+m}{2} = m\end{aligned}$$

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

1.5 Paradoxes and Related Techniques

This section presents one of the first computability results.

First, we will consider computer programs, as entities defining an effective (or automatic, mechanizable) procedure to process an *input* so to construct, upon termination, an *output*. We will then restrict to the simple case where inputs and output are just natural numbers. In this case, we can say that a given program compute a partial function $\mathbb{N} \rightsquigarrow \mathbb{N}$.

Then, we will show the existence of a specific total function $f \in \mathbb{N} \to \mathbb{N}$ which *no program can compute*. In other words, if we consider the set $\mathbb{R}$ of the partial functions g such that there is at least one program that can compute g, our function f does not belong to $\mathbb{R}$. Again, in other words $\mathbb{R}$ is a *strict* subset of $\mathbb{N} \rightsquigarrow \mathbb{N}$. We will see that, intuitively, f is just "too complex" to be computed by a program. While a computer is a magnificent device which can solve a large amount of different tasks, still its power has some limits: tasks so complex that no computer can possibly solve do exist.

In order to construct this "impossible-to-compute" function f we need to borrow a clever proof technique from logic: the *diagonalisation* technique.

1.5.1 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:

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

But if this were true for *all* men m, we could 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.

1.5.2 Diagonalisation

Theorem 31 (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:

$$\begin{array}{c|ccccl} & x & y & z & \dots & \text{(all the elements of } A\text{)} \\ \hline f(x) & \underline{yes} & no & no & \dots & \\ f(y) & no & \underline{no} & no & \dots & \\ f(z) & no & yes & \underline{yes} & \dots & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \end{array}$$

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	$\ldots$	(all the elements of A)
X	no	yes	no	$\ldots$	

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

Exercise 32. *Construct a bijection from* $\mathbb{R}$ *to the interval* $[0, 1)$.
(Hint: start from $\arctan(x)$*)*

Theorem 33. *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 34. *There is no bijection* f *between* $\mathbb{N}$ *and* $(\mathbb{N} \to \mathbb{N})$. **Proof**

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

Proof. Left as an exercise. Hint: prove the following

$$\begin{aligned} &\emptyset \neq B \subseteq B' \land f \in (A \to B').\ f \text{ surjective} \\ &\implies \exists g \in (A \to B).\ g \text{ surjective} \end{aligned}$$

□

1.6 Cardinality Argument for Incomputability

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)

Proof

Theorem 36. *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. 26). 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 .35 it can not be surjective. □

Note that the proof above actually hints to one of these incomputable 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 P_n to denote the n-th program. So, we can write $\varphi_x(y)$ for the output of the x-th program (P_x) 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 φ_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.

Theorem 37. *The total function f defined above is not computable.* **Proof**

Proof. First, note that $f(i)$ is defined for all i, so f is indeed a total function.

By contradiction, assume that f is computable by some program P. Since programs can be enumerated, we have $P = P_x$ for some natural index x. The fact that P_x computes f can be written as $\forall i.\ \varphi_x(i) = f(i)$. Since this holds for all i, we can pick $i = x$ and have $f(x) = \varphi_x(x)$. Since f is total $\varphi_x(x)$ must be defined. From this last statement, by expanding the definition of f we get $\varphi_x(x) = f(x) = \varphi_x(x)+1$. This is a contradiction. $\square$

Exercise 38. *What happens if we change the* 0 *in the definition of f to some other natural? Does the incomputability argument still hold? What if we change it to "undefined", thus defining f to be a partial function?*

1.7 Summary

The most important facts in this section:

- naïve set theory ; logical formulas
- encoding and decoding functions for $\mathbb{N}^2, \mathbb{N} \uplus \mathbb{N}$
- diagonalization method for constructing a non-computable function

Chapter 2

The λ Calculus

Why the λ-calculus in a computability course?

The usual way to introduce students to computability theory is to work in a rather abstract setting, and reason about what can (and can not) computed by programs by making as less assumptions as possible about what programs *are*, in which *programming language* they are written (if any at all), and how they are *executed*. Being, in a sense, "language-agnostic" is one of the main strengths of computability theory, since it allows one to achieve very general results.

On the other hand, coping with this high level of abstraction might be difficult for students, at least at the beginning. More precisely, it can be hard to keep track of the connections between the abstract theory (functions, indexes, enumerations) and the more concrete world of computer science (programming languages, interpreters, semantics). In order to bridge the gap, it is possible to first present computability results on a *specific* programming language, and then abstract from that choice later on, when (hopefully) a strong intuition about the meaning of such results has been developed.

Another point in favour of starting our investigation using a specific programming language is the following. Some results in computability are "positive", in the sense that they state that some function can indeed be computed by a program. To prove this in an abstract setting, where no convenient programming language can be used, can be a daunting task. Often, a full proof would be rather long, full of technicalities, tedious, and not very useful to students as there is no deep insight to be gained from such a proof. Indeed, it is common practice to omit these proofs, and refer to some informal principle such as Church's Thesis to support the statement.

Instead, when a programming language is used, these proofs amount to solving specific programming exercises, which is a task worth doing in a Computer Science course.

So, why using the (untyped) λ calculus and not another programming language (say, Java)? The λ calculus has some specific features which, at least in my opinion, make it a very good choice for studying computability.

- The syntax of the λ calculus is extremely small. This greatly helps when defining procedures which manipulate program code, since we have a very small number of cases to consider, only. By comparison, the full Java syntax is huge.
- The full semantics of the λ calculus fits half of a page, even when including all the auxiliary definitions. This helps in constructing interpreters (or compilers). Building a full Java interpreter is much more complex.
- The λ calculus is reasonably expressive. Despite being minimalistic, all the common building blocks of programs can be defined. This includes data types (e.g. booleans, naturals), usual operations (e.g. multiplication, testing for $\leq$), data structures (e.g. lists, trees), control structures (if-then-else, loops, recursion).
- Some classic computability results have a remarkably simple and elegant proof when using the λ calculus.

To be fair, there are some drawbacks as well. For instance, we will omit the proofs of some fundamental facts such as the Normalization theorem and the Church-Rosser theorem, since these are not short enough to be included in a computability course without sacrificing too much time. Further, as we will see, some infelicities arise from subtle differences among "not having a numeral normal form", "not having a normal form", and "being unsolvable".

In this chapter, we will provide a short introduction to the untyped λ calculus. For the full gory details, see the introduction of [Barendregt].

2.1 Syntax

Definition

Definition 39 (λ-terms)**.** *Let* $\mathsf{Var} = \{x_0, x_1, \ldots\}$ *be a denumerable set of variables. The syntax of the* λ*-terms is*

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

The set of all λ-terms is written as Λ.

Intuitively, a λ-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 λ calculus there is *no other syntax* other than that shown in Def. 39. Later, we shall see how we can express things like 5 and x^2 in the calculus.

Exercise 40. *Rewrite the definition of Λ, 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.

A often-used set of inductive rules are the structural rules. They are used to allow a relation R between λ-terms to be applied to each subterm.

Definition 41. *The λ-structural inductive rules for a relation* R *between λ-terms are the following:*

$$\frac{M \mathrel{\mathsf{R}} N}{(MO) \mathrel{\mathsf{R}} (NO)} \qquad \frac{M \mathrel{\mathsf{R}} N}{(OM) \mathrel{\mathsf{R}} (ON)} \qquad \frac{M \mathrel{\mathsf{R}} N}{(\lambda x.\, M) \mathrel{\mathsf{R}} (\lambda x.\, N)}$$

2.2 Curry's Isomorphism

How to express functions with more than one parameter in the λ-calculus? The answer is suggested by the following result.

Lemma 42. *Let A, B, C be sets. Then, there exists a bijection*

$$\Big[(A \times B) \to C\Big] \leftrightarrow \Big[A \to (B \to C)\Big]$$

Proof. Left as an exercise. □

To represent binary functions using only unary functions, we proceed as follows. 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 α-conversion, Free Variables, and Substitution

In computer programs, the name of variables is immaterial. Variables can be arbitrarily renamed without affecting the run-time behaviour of the program. It is important, though, that *all* the occurrences of the same variable are renamed consistently. This includes both variable *declaration* and *use*, as we can see below.

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

Above the "λx" *declares*, or *binds*, the variable x which is then *used* in the expression $x^2 + 5$. If we want to rename x to y, we can intuitively do that without affecting the meaning of the expression, as long as we rename all the occurences of x.

The renaming of program variables is known as *α-conversion*, and is written as $=_\alpha$.

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

In order to precisely define the rules for α-conversion, we start with identifying those variables which can *not* be renamed. For instance, we can not rename those variables for which there is no declaration, i.e. no enclosing λ in the λ-term at hand. For example, consider the following:

$$\lambda x.\, x + z$$

Here, we can rename x, but we can not rename z, since there is no λz around. Such a variable is said to be *free.*

Definition 43. *The* free variables $\mathsf{free}(M)$ *of a λ-term are those not under a λ-binder. Formally, they are inductively defined as follows:*

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

Definition

Exercise 44. *Prove that for all λ-terms M, the set $\mathsf{free}(M)$ is finite.*

Let us consider the λ-term $\lambda x.\ M$. Roughly, in order to α-convert a variable x into y we have to perform two steps: 1) change λx into λy; 2) substitute every x in M into y. Formally, the substitution in step 2 is denoted with $M\{y/x\}$.

Note that formally defining the result of the substitution is not as trivial as it might seem. For instance, consider the following:

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

Renaming the x in the body of the λx is done by

$$(\lambda y.\, x + y)\{y/x\}$$

A **wrong** result for this would be $\lambda y.\, y+y$. This is wrong because otherwise we would have the following α-conversion:

$$\lambda x.\, \lambda y.\, x + y =_\alpha \lambda y.\, \lambda y.\, y + y \qquad \textbf{wrong}$$

In the right hand side there is no information about which declaration (λy) is related to each use of y in $y+y$. The meaning of the original expression is lost. Causing this kind of confusion must therefore be forbidden. If we really want to rename x to y, we also need to rename the "other" y to something different beforehand, e.g. as follows:

$$\lambda x.\, \lambda y.\, x + y =_\alpha \lambda y.\, \lambda z.\, y + z$$

In order to do that, we should define substitution such that e.g.

$$(\lambda y.\, x + y)\{y/x\} = (\lambda z.\, y + z)$$

This is done as follows. Below we generalize the variable-variable substitution $M\{y/x\}$ to the more general variable-term substitution $M\{N/x\}$, allowing x to be replaced with an arbitrary term N, rather than just a variable y.

Definition 45. *The result of applying a substitution $M\{N/x\}$ is defined as follows.*

$$\begin{array}{ll}
x_i\{N/x_i\} = N & \\
x_i\{N/x_j\} = x_i & \textit{when } i \neq j \\
(MO)\{N/x_i\} = (M\{N/x_i\})(O\{N/x_i\}) & \\
(\lambda x_i.\, M)\{N/x_i\} = (\lambda x_i.M) & \\
(\lambda x_j.\, M)\{N/x_i\} = \lambda x_k.\,(M\{x_k/x_j\}\{N/x_i\}) & \textit{when } i \neq j \\
\quad \textit{where } k = \min\{k \mid x_k \notin \mathsf{free}(N) \cup \mathsf{free}(\lambda x_j.\, M)\} &
\end{array}$$

Definition

In the last line we avoid variable clashes. First, we rename x_j to x_k, a "fresh" variable, picked[1] 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.

Note: as a consequence of having $(\lambda x_i.\, M)\{N/x_i\} = (\lambda x_i.M)$ we get

$$\lambda x.\, \lambda x.\, x + x =_\alpha \lambda y.\, \lambda x.\, x + x$$

This means that, whenever the same variable x appears in two nested declarations, the inner one "shadows" the outer one. That is, the x occurring in $x+x$ is the one declared by the *inner* λ-binder. This follows the same *static scoping* conventions found in programming languages: each occurrence of a variable is bound by the innermost definition.

We can finally formally define our $=_\alpha$ relation.

Definition 46 (α-conversion)**.** *The (equivalence) relation $=_\alpha$ between λ-terms is inductively defined by the following inductive rules:*

- *equivalence relation rules for $=_\alpha$ (see Def. 20)*
- *λ-structural rules for $=_\alpha$ (see Def. 41)*
- *rule α*
  $\lambda x.\, M =_\alpha \lambda y.\, M\{y/x\}$ *when* $y \notin \mathsf{free}(M)$

Definition

For more details, see [Barendregt 2.1.11].

Following [Barendregt], unless otherwise stated, we will often consider λ-terms up to $=_\alpha$; i.e. we will consider α-congruent terms as identical. To stress this fact we will include the following inference rule in our inductive definitions.

[1]We pick the variable x_k having minimum index k. This peculiar choice is actually irrelevant. Picking any other "fresh" variable would lead to exactly the same α-conversion relation.

Definition 47 (Up-to-α rule)**.** *The "up-to-α" inference rule for a relation* R *between λ-terms is the following.*

$$\frac{M =_\alpha M' \quad M' \mathrel{\mathsf{R}} N' \quad N' =_\alpha N}{M \mathrel{\mathsf{R}} N}$$

2.4 β and η Rules

Definition 48 (β rule)**.** *Here's the β rule, used to compute the result of function application.*

$$(\lambda x.\, M)N \rightarrow^t_\beta M\{N/x\}$$

(Note: the t stands for "at the top-level") **Definition**

Example:

$$(\lambda x.x^2 + x + 1)5 \rightarrow^t_\beta 5^2 + 5 + 1$$

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 49 (η rule)**.** *Here's the η rule, used to remove redundant λ's.* **Definition**

$$(\lambda x.\, Mx) \rightarrow^t_\eta M \qquad\qquad \text{if } x \notin \mathsf{free}(M)$$

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 50. *Can you state the η rule in Java (or another procedural language), at least in some loose form?*

Relations $\rightarrow^t_\beta$ and $\rightarrow^t_\eta$ can be extended so that β and η rules can be applied to subterms as well, i.e. not only at the top level.

Definition 51. *Given a reduction relation $\rightarrow^t_R$ (e.g. with $R = \beta$ or $R = \eta$), we define the relation $\rightarrow_R$ on λ-terms as per the inductive rules below.* **Definition**

- *λ-structural rules for $\rightarrow_R$ (see Def. 41)*
- *up-to-α rule for $\rightarrow_R$ (see Def. 47)*
- *top-level rule R*
  $M \rightarrow_R N$ *when* $M \rightarrow^t_R N$

Example 52. *Here's an example which shows that in some cases it is* mandatory *to α-convert λ-bound variables.*

$$\begin{aligned}
&(\lambda x.\ ((\lambda y.(\lambda x.\ y\ x))\ (x\ x)))\\
&\rightarrow_\beta (\lambda x.\ (\lambda x.\ y\ x)\{(x\ x)/y\})\\
&= (\lambda x.\ (\lambda \hat{x}.\ x\ x\ \hat{x}))
\end{aligned}$$

In the last line, the inner λx must be renamed, since $x \in \mathsf{free}(x\ x)$. Forgetting to rename x leads to the **wrong** *result $(\lambda x.\ (\lambda x.\ x\ x\ x))$, in which all the x's are bound by the inner λx, i.e. the wrong result can be α-converted to $(\lambda y.\ (\lambda x.\ x\ x\ x))$, which is completely different from the correct result.*

Suggestion. *Since the definition of $\to_\beta$ includes the "up-to-α" rule, we are allowed to rename variables before applying β. A simple thumb rule to avoid mistakes such as the above one is*

always keep λ-bound variables distinct:
immediately rename multiple occurrences of λx for the same x

In the example above, the rule suggest to immediately perform this renaming:

$$(\lambda x.\ ((\lambda y.(\lambda x.\ y\ x))\,(x\ x))) =_\alpha (\lambda x.\ ((\lambda y.(\lambda \hat{x}.\ y\ \hat{x}))\,(x\ x)))$$

We can now apply β in a safe way without caring about needed α-conversions: since we renamed everything earlier, no further α-conversion is needed. This thumb rule can cause you to perform more α-conversions than strictly needed, but will never lead you to a wrong result.

Unlike $\to^t_R$, the above relations are non-deterministic, i.e. they can lead to different residual λ-terms.

Exercise 53. *Prove that the relations $\to_R, R \in \{\beta, \eta\}$ above are non-deterministic, i.e.*

$$M \to_R M_1 \wedge M \to_R M_2 \wedge M_1 \neq M_2$$

for some M, M_1, M_2.

Sometimes a single $\to^t_{\beta\eta}$ or $\to_{\beta\eta}$ relation is used to denote either the β or η reduction relation.

Definition

Definition 54. *We let*

$$\begin{array}{ll} M \to^t_{\beta\eta} N & \textit{iff } M \to^t_\beta N \textit{ or } M \to^t_\eta N \\ M \to_{\beta\eta} N & \textit{iff } M \to_\beta N \textit{ or } M \to_\eta N \end{array}$$

A λ-term that can not be further reduced is said to be in *normal form*.

Definition 55 (Normal form)**.** *Given a reduction relation $\to_R$ (e.g. with $R = \beta$, $R = \eta$, or $R = \beta\eta$), we say that a term M is in R-normal form iff $M \not\to_R$.*

Definition

2.4.1 β Normal Forms

We now consider the repeated application of $\to_\beta$ starting from a given λ-term M. This constructs a sequence such as the following one:

Definition 56. *A β-reduction*[2] *for M is a finite or infinite sequence of terms M_i such that:*

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

Intuitively, this corresponds to "executing" program M: at each step the expression at hand is rewritten in an equivalent form (according to β). Exactly one of the following must hold:

- The β-reduction stops: that is, we reach some M_k which is a β-normal form. Intuitively, this is the result of running M. We say that the β-reduction above *halts.*

- The β-reduction never stops: that is, it is infinite. So, the β-reduction is non-halting.

When a normal form is reached, we regard that λ-term as the result (the "output") of the β-reduction. If instead it does not exist, we regard the β-reduction as a non-terminating one (is "divergent").

Recall that the relation $\to_\beta$ is non-deterministic. So, a term might have *multiple different β-reductions.*

Exercise 57. *Construct different β-reductions for*

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

As far as we know, a term M could have different β-reductions leading to different β-normal forms.

Definition 58. *We say that N is* a β-normal form of M *if and only if M has some β-reduction ending with N, and N is a β normal form.*

Here's an example of a term having *no* β-normal form.

Exercise 59. *Show that $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ has no halting β-reduction, hence no β normal form.*

Definition

[2]We follow the terminology of [Barendregt] here. Reductions as the above are also called *runs*, or *traces* for M.

Here's an example of a term having *no* β-normal form and having a β-reduction made of distinct terms.

Exercise 60. *Check the above on* $\Omega_3 = (\lambda x.\, xxx)(\lambda x.\, xxx)$.

Here's an example of a term having *one* β-normal form.

Exercise 61. *Show that* $\lambda x.\ x$ *has exactly one* β *normal form. (Yes, it is trivial.)*

Here's an example of a term having exactly *one* β-normal form, despite having infinitely many halting reductions, and infinitely many non-halting reductions.

Exercise 62. *Prove the above using* $(\lambda x.5)(\Omega_3\Omega_3)$.

Now, a question arises. Can a λ-term have more than one β-normal form? The following result states that, while there might be multiple different β-reductions, any term M has at most one β-normal form (up to α-conversion[3]). Alas, we omit the proof.

Definition 63. *A relation* $\rightarrow_R$ *is a Church-Rosser relation iff* $\forall M, N_1, N_2$

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

Theorem 64 (Church-Rosser)**.** *The relation* $\rightarrow_\beta$ *is a Church-Rosser relation. As a consequence, each* λ*-term has at most one* β*-normal form (up-to* α*-conversion).*

[Barendregt 3.2.8 — no proof].

Now we know that a given M has either zero or one β-normal forms. So, we are now entitled to say "<u>the</u> β-normal form" instead of "<u>a</u> β-normal form".

So, how can we compute the β-normal form of a term M (assuming there is one)? Fortunately, we do not need to search among all possible β-reductions of M (which may be infinite): by the following result, it is enough to check just one specific β-reduction.

Definition 65 (Leftmost-outermost reduction relation)**.** *The leftmost-outermost* β*-reduction relation is* $\rightarrow_\beta$ *constrained as follows: it must be applied as to the left as possible, i.e. to the first occurrence of an applied* λ *binder, reading*

[3]That is, if N_1 and N_2 are two β-normal forms for M, then $N_1 =_\alpha N_2$.

the λ-term left-to-right. Below we show a procedure to compute the leftmost-outermost residual.

```
procedure $\mathsf{L}(M)$
Input: a $\lambda$-term $M$
Output: either a leftmost-outermost residual of $M$,
        or the special constant NormalForm if no residual exists
if $M = x_i$ then return NormalForm
else if $M = \lambda x_i.N$ then
        if $\mathsf{L}(N) \neq$ NormalForm then return $\lambda x_i.\ \mathsf{L}(N)$
        else return NormalForm
else if $M = NO$ then
        if $N = \lambda x_i.P$ then return $P\{O/x_i\}$
        else if $\mathsf{L}(N) \neq$ NormalForm then return $(\mathsf{L}(N))O$
        else if $\mathsf{L}(O) \neq$ NormalForm then return $N(\mathsf{L}(O))$
        else return NormalForm
```

Exercise 66. *Prove that the above procedure indeed applies β in a leftmost-outermost way. Proceed by induction on the structure of M.*

By the following theorem, to find a normal form we just need to apply L repeatedly. Alas, we omit the proof.

Theorem 67 (Normalization)**.** *The leftmost-outermost strategy (i.e. repeatedly applying procedure* L *above) is normalizing, i.e. it finds the β-normal form as long as it exists.* **Statement**

Nota Bene*: when no β-normal form exists, this strategy constructs to an* infinite *β-reduction, so it never halts.*

[Barendregt 13.2.2 — no proof]

The fact that the normalizing procedure above may fail to halt (as it does when M has no normal forms) is no coincidence. Indeed, we will use results from computability theory to explain that there is actually *no way* we can improve the above procedure by very much. More concretely, we will later on prove that each algorithm to find the β-normal form[4] of a term M must fail to halt for some M. In other words, "fixing" the normalization procedure to print the message "there is no normal form" when that is the case is simply impossible.

[4]When it exists.

2.4.2 η Normal Forms

While finding β-normal forms can be a hard task, η-normal forms are almost trivial. This is because η-normal forms always exist, unlike for β.

A η-reduction is defined as for β-reduction, mutatis mutandis. Similarly for the notion of "N is a η-normal form of M", etc.

Exercise 68. *Define a function* $\mathsf{size}(M)$ *which counts the number of syntactic elements (abstractions, applications, variables) in* M*.*

Then, prove that if $M \to_\eta N$ *then* $\mathsf{size}(M) > \mathsf{size}(N)$*.*

Finally use the above result to prove that no infinite η*-reduction.*

Exercise 69. *Prove that* $\to_\eta$ *is Church-Rosser.*

Theorem 70 (Existence and uniqueness of η)**.** *Each given* M *admits exactly one* η*-normal form.*

Proof. The above exercises imply the statement. □

Exercise 71. *Let* M *be a* β*-normal form, and* $M \to_\eta N$*. Prove that* N *is still a* β*-normal form.*

Exercise 72 (Commuting η and β)**.** *Prove the following. If* $M \to_\eta N \to_\beta O$*, then* $M \to_\beta N' \to_\eta^* O$ *for some* N'*.*

Theorem 73. M *has a* β*-normal form if and only if* M *has a* $\beta\eta$*-normal form.*

Proof. ($\Rightarrow$) Immediate from Ex.71. (exercise)

($\Leftarrow$) Assume $M \to_{\beta\eta}^* N$ with N $\beta\eta$-normal form. This means that there is a reduction

$$M \to_{\gamma_1} \cdots \to_{\gamma_n} N$$

with $\gamma_i \in \{\beta, \eta\}$. By repeated application of Ex. 72 we get that there is also a reduction

$$M \to_\beta^* N' \to_\eta N$$

for some N' in β-normal form. This concludes. □

The previous results allow us to state the following.

Theorem 74 (Normalization for $\to_{\beta\eta}$)**.** *To find the* $\beta\eta$*-normal form for* M *(when existing), it is enough to apply the normalizing leftmost-outermost strategy, take its output (a* β*-normal form of* M*), and apply* $\to_\eta$ *as far as possible.*

Proof. Direct from the lemmata above. □

2.4.3 Equational Theory

The relation $\to_{\beta\eta}$ describes how to "compute" with the λ-calculus. We now exploit this relation to define an equivalence between λ-terms.

Definition 75 (Axiomatic semantics for the untyped λ-calculus)**.** *The equivalence relation $=_{\beta\eta}$ between λ-terms is inductively defined below.* Definition

- *equivalence relation rules for $=_{\beta\eta}$ (see Def. 20)*
- *rule $\beta\eta$*
  $M =_{\beta\eta} N$ *when* $M \to_{\beta\eta} N$

We also write $=_\beta$ (respectively, $=_\eta$) for the equivalence relations defined by using $\to_\beta$ (resp. $\to_\eta$) instead of $\to_{\beta\eta}$.

Convention: when unambiguous we shall often write $M = N$ instead of $M =_{\beta\eta} N$.

Note that using the structural rules one can apply the β and η rules even to *subterms* of the λ-term at hand, e.g.

$$\lambda x.\,((\lambda y.\,y)a) =_{\beta\eta} \lambda x.\,a$$

Indeed, the following holds.

Exercise 76. *Prove that $=_{\beta\eta}$ is closed under the λ-structural rules of Def. 41.*

Exercise 77. *Use the η rule to prove the* **ext** *rule.*

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

Exercise 78. *Show that the η rule is actually equivalent to the* **ext** *rule above.*

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

Theorem 79. *If $M =_{\beta\eta} N$ and N is a $\beta\eta$-normal form, then $M \to^*_{\beta\eta} N$.* Statement

Proof. Left as an exercise. Suggestion: prove the following stronger statement, instead.

- If $M =_{\beta\eta} O$ both the following properties hold:
  - if $O \to^*_{\beta\eta} N$ and N is a $\beta\eta$-normal form, then $M \to^*_{\beta\eta} N$

– if $M \to^*_{\beta\eta} N$ and N is a $\beta\eta$-normal form, then $O \to^*_{\beta\eta} N$

Proceed by induction on $=_{\beta\eta}$. You might want to exploit the Church-Rosser property in some case.

See also [Barendregt 3.2.9] for a proof. □

Figure 2.1 provides a summary of the syntax and semantics of the λ-calculus.

2.5 Useful Combinators

Definition

Below, we list several common λ-terms.

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

The λ-term **I** represents the identity function. The λ-term **K** is used to build constant functions: e.g. $\mathbf{K}\,5$ is a function which always returns 5, since $\mathbf{K}\ 5\ x = 5$ for all x.

The λ-terms **T** and **F** are used to represent the boolean values "true" and "false". We will provide a justification for this choice below.

Example 80. *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*

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

free variables $(\mathsf{free}(M), \Lambda^0)$

$\mathsf{free}(x_i) = \{x_i\}$

$\mathsf{free}(NO) = \mathsf{free}(N) \cup \mathsf{free}(O)$

$\mathsf{free}(\lambda x_i.N) = \mathsf{free}(N) \setminus \{x_i\}$

$\Lambda^0 = \{M \mid \mathsf{free}(M) = \emptyset\}$

λ-terms (M, Λ)

$M \;::=\; x_i \mid (M\ M) \mid (\lambda x_i.\ M)$

$\Lambda = \{\ M \mid M$ is a λ-term $\}$

equivalence relation rules for R

$$\frac{}{x\ \mathsf{R}\ x} \qquad \frac{x\ \mathsf{R}\ y}{y\ \mathsf{R}\ x} \qquad \frac{x\ \mathsf{R}\ y \quad y\ \mathsf{R}\ z}{z\ \mathsf{R}\ z}$$

λ-structural rules for R

$$\frac{M\ \mathsf{R}\ N}{(MO)\ \mathsf{R}\ (NO)} \qquad \frac{M\ \mathsf{R}\ N}{(OM)\ \mathsf{R}\ (ON)} \qquad \frac{M\ \mathsf{R}\ N}{(\lambda x.\,M)\ \mathsf{R}\ (\lambda x.\,N)}$$

substitution $(M\{N/x\})$

$x_i\{N/x_i\} = N$

$x_i\{N/x_j\} = x_i$ if $i \neq j$

$(MO)\{N/x_i\} = (M\{N/x_i\})(O\{N/x_i\})$

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

$(\lambda x_j.\,M)\{N/x_i\} = \lambda x_k.\,(M\{x_k/x_j\}\{N/x_i\})$ if $i \neq j$

where $k = \min\{k \mid x_k \notin \mathsf{free}(N) \cup \mathsf{free}(\lambda x_j.\,M)\}$

α conversion $(=_\alpha)$

- equivalence relation rules for $=_\alpha$
- λ-structural rules for $=_\alpha$
- $\lambda x.\ M =_\alpha \lambda y.\ (M\{y/x\})$ if $y \notin \mathsf{free}(M)$

up-to-α inference rule for R

$$\frac{M =_\alpha M' \quad M'\,\mathsf{R}\,N' \quad N' =_\alpha N}{M\,\mathsf{R}\,N}$$

β reduction relation $(\to_\beta^t, \to_\beta)$

- $(\lambda x.\,M)N \to_\beta^t M\{N/x\}$

- λ-structural rules for $\to_\beta$
- up-to-α rule for $\to_\beta$
- top-level rule $\dfrac{M \to_\beta^t N}{M \to_\beta N}$

η reduction relation $(\to_\eta^t, \to_\eta)$

- $(\lambda x.\,M\,x) \to_\eta^t M$ if $x \notin \mathsf{free}(M)$

- λ-structural rules for $\to_\eta$
- up-to-α rule for $\to_\eta$
- top-level rule $\dfrac{M \to_\eta^t N}{M \to_\eta N}$

$\beta\eta$ reduction relation $(\to_{\beta\eta}^t, \to_{\beta\eta})$

$M \to_{\beta\eta}^t N$ iff $M \to_\beta^t N$ or $M \to_\eta^t N$

$M \to_{\beta\eta} N$ iff $M \to_\beta N$ or $M \to_\eta N$

$\beta\eta$ equivalence $(=_{\beta\eta})$

- equivalence relation rules for $=_{\beta\eta}$
- $\dfrac{M \to_{\beta\eta} N}{M =_{\beta\eta} N}$

Figure 2.1: The syntax and semantics of the untyped λ-calculus

Exercise 81. *Prove that we do not have* $\mathbf{T} =_{\beta\eta} \mathbf{F}$*. See Sol. 225.*

Exercise 82. *Define*

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

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

$$\begin{aligned} \text{if } \mathbf{T} \text{ then } N \text{ else } O &=_{\beta\eta} N \\ \text{if } \mathbf{F} \text{ then } N \text{ else } O &=_{\beta\eta} O \end{aligned}$$

This justifies the names for $\mathbf{T}$ *and* $\mathbf{F}$*.*

Exercise 83. *Define the usual logical operators:* **And***,* **Or***,* **Not***. (See Sol. 226)*

Lemma 84. *Application is not associative, that is*

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

Proof. By contradiction,

$$\begin{aligned} (\mathbf{K}(\mathbf{IT}))\mathbf{F} &= \mathbf{IT} = \mathbf{T} \\ ((\mathbf{KI})\mathbf{T})\mathbf{F} &= \mathbf{IF} = \mathbf{F} \end{aligned}$$

□

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

$$\begin{aligned} &MN = NM \\ &M(NO) = O(MN) \\ &M(MO) = MO \\ &MO = MOO \\ &MM = M \\ &MN = \lambda x.\, M(Nx) \end{aligned}$$

Exercise 86. *Check whether these terms have a β-normal form*

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

2.5.1 Pairs

Pairs can be encoded as follows: **Definition**

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

Exercise 87. *Prove the usual pair laws:*

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

Exercise 88. *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, consider 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 λ-term f that satisfies the equation above? Of course, the equation itself has f on both sides so it does not define a λ-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 λ-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*[5] 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 λ-term f.

Exercise 89. *Use the above definition of f to compute the factorial of* 3.

Exercise 90. *Write a λ-term for computing $\sum_{i=0}^{n} i^2$.*

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 β 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 λ-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 91. *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. 224.

Definition

[5] Oh, the irony...

Exercise 92. *Check whether these terms have a β-normal form*

$$\begin{array}{l}\mathbf{\Theta}\\ \mathbf{KI\Theta}\\ \mathbf{K\Theta I}\\ \mathbf{\Theta I}\\ \mathbf{\Theta K}\\ \mathbf{\Theta}(\mathbf{KI})\end{array}$$

2.7 Church's Numerals

The λ calculus does not have any numbers in its syntax. In spite of this, it is possible to *encode* naturals into λ-terms, and compute with them. That is, we shall pick an infinite sequence of (closed) λ-terms, and use them to denote naturals in the λ calculus. We shall name these λ-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 λ 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 93. *The sequence of Church numerals is inductively defined as follows. Let s and z be variables*[6]. **Definition**

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

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

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

[6]E.g. let us pick $s = x_0$ and $z = x_1$.

Then, we can define a “zero” and “successor” λ-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:

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

Exercise 94. *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 95. *Check that* **Pred** *is correct.*

Exercise 96. *Define the usual arithmetic operators and comparisons. Also see the solution in the appendix (Sol. 226).*

Exercise 97. *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* **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. 227.

Exercise 98. *Find an encoding for lists of arbitrary (opaque) data, and adapt the functions seen above. What about binary trees?*

2.8 λ-definable Functions

In this section, we define when a λ-term M can be regarded as the implementation of some (partial) function $f \in (\mathbb{N} \rightsquigarrow \mathbb{N})$. Clearly, we must require at least the following:

$$\text{when } f(n) \text{ is defined, then } M\ulcorner n \urcorner =_{\beta\eta} \ulcorner f(n) \urcorner$$

But what to require when $f(n)$ is not defined? That is, when $n \notin \mathsf{dom}(f)$, what should we require for $M\ulcorner n \urcorner$?

$$\text{when } f(n) \text{ is not defined, then } M\ulcorner n \urcorner \ldots \text{(what to put here?)} \ldots$$

Intuitively we want to state "$M\ulcorner n \urcorner$ does not provide a result". Below, we list several available options to state this:

Definition 99. *Options for representing undefinedness:*

1. *when $f(n)$ is not defined, then $M\ulcorner n \urcorner$ has no* numeral *$\beta\eta$-normal form.*
2. *when $f(n)$ is not defined, then $M\ulcorner n \urcorner$ has no $\beta\eta$-normal form.*
3. *when $f(n)$ is not defined, then $M\ulcorner n \urcorner N_1 \cdots N_k$ has no $\beta\eta$-normal form, for all $N_1, \ldots, N_k$.*

Option 1 above is the most simple one: we regard anything which is not a numeral (according to $\beta\eta$) as "undefined". According to this definition, each λ-term M has an associated function f such that M λ-defines f. Unfortunately, some technical difficulties arise with this option. For instance, consider the following programs:

$$G = \lambda x.\, x\,\mathbf{K}\,\Omega \qquad H = \lambda xyz.\, y\,\ulcorner 0 \urcorner\,\Omega$$

According to option 1 above, both these programs implement the always-undefined function. Indeed $G\ulcorner n \urcorner = \ulcorner n \urcorner \mathbf{K}\Omega = \lambda x_1 \ldots x_n.\, \Omega$ which is not a numeral (it does not even have a $\beta\eta$-normal form). Also, $H\ulcorner n \urcorner = \lambda yz.\, y\,\ulcorner 0 \urcorner\,\Omega$ which is not a numeral (as before).

So, what is wrong with the programs G, H above? Let us try to compose them. Intuitively, composing the always-undefined function with itself, will yield again the always-undefined function. However, this is not the case with the G, H programs above:

$$\lambda n.\, G(Hn) =_{\beta\eta} \lambda n.\, H\; n\; \mathbf{K}\; \Omega =_{\beta\eta} \lambda n.\, \mathbf{K}\; \ulcorner 0 \urcorner\; \Omega =_{\beta\eta} \lambda n.\, \ulcorner 0 \urcorner$$

So, according to option 1 above, we can have two always-non-terminating programs which, once composed, implement the always-defined constant function 0. This is highly counter-intuitive, and we want to avoid this.

A possible solution would be to use a more sophisticated way of composing functions. That is instead of using $\lambda n.\, G(Hn)$ we would use something more complex. Even when doing this, it is still unclear how to watch out for "garbage" such as the one found above.

For the time being, it is easier if we just rule out this garbage, and require that when $f(n)$ is not defined, $M\ulcorner n \urcorner$ must not only differ from numerals, but also differ from the garbage above. So, we discard option 1 for something stronger. Note that option 2 above is stronger, yet not strong enough to disallow H. Instead, we shall take option 3: H is now ruled out since

$$H \ulcorner n \urcorner\ (\lambda ab.\, \mathbf{I})\ \Omega =_{\beta\eta} \mathbf{I}$$

G is instead not ruled out: $G\ulcorner n \urcorner N_1 \ldots N_k = \mathbf{K}(\ldots(\mathbf{K}\Omega))N_1 \ldots N_k$ has no normal form, no matter how we choose the N_i. Hence $G\ulcorner n \urcorner$ complies with option 3.

Option 3 is best described in terms of *solvability*.

Definition

Definition 100 (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 101. *Show that if M is unsolvable, then MN is also unsolvable, for any N.*

Exercise 102. *For each term in the following list, state whether it is solvable or not.*

$$\Omega,\ (\lambda x.\, \Omega),\ (\lambda x.\, x\, \Omega),\ (\lambda x.\, \Omega\, x),\ \mathbf{KKI},\ \mathbf{\Theta},\ \mathbf{SII}$$

Exercise 103. *Show that Church's numerals can be* uniformly solved *by finding M, N such that $\forall n \in \mathbb{N}. \ulcorner n \urcorner MN = \mathbf{I}$.*

Statement

Theorem 104. *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 now define λ-definability for (partial) functions:

Definition 105 (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 $\mathbb{N}^k \rightsquigarrow \mathbb{N}$.

Definition

Note that according to the above definition, the "garbage" λ-term H seen before does not λ-define any function. Indeed, it returns something which is not a numeral, yet solvable, and this is forbidden by the definition above.

Exercise 106. *Show that if f, g are partial λ-definable functions, then their composition $f \circ g$ is such.*
Hint: exploit Ex. 103, 101.

Definition 107. *A set $A \subseteq \mathbb{N}$ is λ-defined by G iff*

$$\begin{array}{l} n \in A \implies G\ulcorner n\urcorner = \mathbf{T} \\ n \notin A \implies G\ulcorner n\urcorner = \mathbf{F} \end{array}$$

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

Definition

Exercise 108. *Change $\mathbf{T}$ with 1 and $\mathbf{F}$ with 0 in the definition above, and prove this alternative notion of λ-definability for sets to be equivalent.*

Exercise 109. *Show that finite subsets of $\mathbb{N}$ are λ-definable.*

Lemma 110. *λ-definable sets are closed under*

- *union* (∪)
- *complement* (∖)
- *intersection* (∩)

Statement

Proof. Left as an exercise. □

Lemma 111. *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 $\mathbf{T}$. If $n \notin A$, then $f(n) \notin B$ since f is injective, and $M_B\ulcorner f(n)\urcorner$ evaluates to $\mathbf{F}$. □

2.9 Classical Computability Results in the λ calculus

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

First, we need to enumerate the λ-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 Λ and $\mathbb{N}$; we write the natural corresponding to M as $\#M$.

Definition

Definition 112. *We define the bijection # as follows.*

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

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

Definition

Definition 113. *The function $\ulcorner M \urcorner$ is defined as follows.*

$$
\begin{aligned}
\ulcorner - \urcorner &\in (\Lambda \to \Lambda^0)\\
\ulcorner M \urcorner &= \ulcorner\!\ulcorner \#M \urcorner\!\urcorner
\end{aligned}
$$

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 λ-defined by F. Then, define

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

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

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

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

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

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 114. *The function f defined above is not λ-definable.* **Proof**

Exercise 115. *Compare this result with Th. 37. 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 116. *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 117. *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 118. $\mathsf{K}_\lambda = \{\#M | M\ulcorner M\urcorner$ *has a β-normal form*$\}$ **Definition**

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

Lemma 119. K_λ *is not λ-definable* **Proof**

Proof. By contradiction, if K_λ 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 $\chi_{\mathsf{K}_\lambda}$, the characteristic function of the set K_λ. □

We want to show that many syntactic transformations of λ-terms, e.g. transforming $\#(MN)$ into $\#(NM)$, can be done in the λ-calculus. To manipulate an index $\#O$ we basically need to compute the $\mathsf{pair}, \mathsf{encode}_{\uplus}$ functions and their inverses.

Exercise 120. *Show that* pair *and* inL, inR *can be* λ*-defined, as well as their inverses. In order to do that, construct the following functions:*

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

Definition

Also see Solution 226 in the appendix.

Exercise 121. *Construct a "shallow decoder" for our bijection #. A shallow decoder is a function which decodes just the top-level structure of the encoded* λ*-term. More in detail, it satisfies the following.*

$$\begin{aligned}
&\mathbf{Sd}\ulcorner x_i\urcorner\, V\ A\ L &&= V\ulcorner\!\ulcorner i\urcorner\!\urcorner\\
&\mathbf{Sd}\ulcorner \lambda x_i.M\urcorner\, V\ A\ L &&= L\ulcorner\!\ulcorner i\urcorner\!\urcorner\ulcorner M\urcorner\\
&\mathbf{Sd}\ulcorner MN\urcorner\, V\ A\ L &&= A\ulcorner M\urcorner\ulcorner N\urcorner
\end{aligned}$$

Definition

See also Solution 229.

2.9.1 Parameter Lemma

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

Lemma 122 (Parameter lemma, s-m-n lemma — simple version)**.** *There exists* $\mathbf{App} \in \Lambda^0$ *such that,* $\forall M, N$

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

Proof

Exercise 123. *Prove it. See Solution 230.*

Exercise 124. *Define a* $G \in \Lambda^0$ *such that: (standalone exercises follow)*

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

- $G\ulcorner M \urcorner = \ulcorner MMM \urcorner$
- $G\ulcorner M \urcorner = \ulcorner M(MM) \urcorner$
- $G\ulcorner MN \urcorner = \ulcorner NM \urcorner$
- $G\ulcorner \lambda x.\, M \urcorner = \ulcorner M \urcorner$
- $G\ulcorner \lambda x.\, \lambda y.\, M \urcorner = \ulcorner \lambda y.\, \lambda x.\, M \urcorner$
- $G\ulcorner \mathbf{I} M \urcorner = \ulcorner M \urcorner$ *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* α*-conversion)*

See Solution 228 in the Appendix.

2.9.2 Padding Lemma

Intuitively, many different programs actually have the same semantics. Indeed, recall Ex. 116. We can actually automatically generate an infinite number of equivalent programs.

Lemma 125 (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* λ*-term* **Pad** *such that*

$$\mathbf{Pad}\ulcorner M \urcorner =_{\beta\eta} \ulcorner N \urcorner$$

Proof. Left as an exercise. See Solution 233. □ **Proof**

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 λ-term **E** ("evaluate") that, given the code $\ulcorner M \urcorner$, can run it and behave as M. This **E** is said to be a universal program, since it can be used to compute anything that can be computed in λ-calculus. It is, in a sense, "the most general program".

Note that we only allow *closed* M here[7].

Lemma 126 (Self-interpreter)**.** *There exists* $\mathbf{E} \in \Lambda^0$ *such that*

$$\mathbf{E}\ulcorner M \urcorner =_{\beta\eta} M$$

Statement

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 $\rho \ulcorner\ulcorner i \urcorner\urcorner$. Here, the rôle of the parameter ρ is to define the meaning of the free variables in M, defining the value of x_i as $\rho \ulcorner\ulcorner i \urcorner\urcorner$. This ρ is called the *environment* function.

- $\mathbf{Upd}\,\rho \ulcorner\ulcorner i \urcorner\urcorner\, a = \rho'$ where ρ' is the "updated" environment, obtained from ρ by replacing the value of x_i with the new value a. Formally,

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

  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:

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

These equations are satisfied by:

$$\begin{aligned}&\mathbf{E}' = \boldsymbol{\Theta}\Big(\lambda f m \rho.\, \mathbf{Sd}\, m\, \rho\, A\, L\Big)\\ &A = \lambda n o.\, f\, n\, \rho\, (f\, o\, \rho)\\ &L = \lambda i n.\, \Big(\lambda a.\, f\, n\, (\mathbf{Upd}\,\rho\, i\, a)\Big)\end{aligned}$$

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

Exercise 127. *Check the correctness of* $\mathbf{E}$ *in some concrete (small) cases. For instance check that* $\mathbf{E}\ulcorner \mathbf{I} \urcorner = \mathbf{I}$ *and* $\mathbf{E}\ulcorner \mathbf{K} \urcorner = \mathbf{K}$.

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

2.9.4 Kleene's Fixed Point Theorem

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

Lemma 128. *There exists* $\mathbf{Num} \in \Lambda^0$ *such that for all* $n \in \mathbb{N}$ Statement

$$\mathbf{Num}\ulcorner\!\!\ulcorner n \urcorner\!\!\urcorner =_{\beta\eta} \ulcorner\ulcorner\!\!\ulcorner n \urcorner\!\!\urcorner\urcorner$$

Proof.

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

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 $y = \ulcorner\!\!\ulcorner n \urcorner\!\!\urcorner$. Then, $A = \ulcorner \lambda x_1.\, x_0(x_0(\cdots(x_0\,x_1)))\urcorner$, and so $\mathbf{Num}\ulcorner\!\!\ulcorner n \urcorner\!\!\urcorner = \ulcorner \lambda x_0.\, \lambda x_1.\, x_0(x_0(\cdots(x_0\,x_1)))\urcorner = \ulcorner\ulcorner\!\!\ulcorner n \urcorner\!\!\urcorner\urcorner$. □

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

Theorem 129 (Kleene's fixed point)**.** *For all* $F \in \Lambda$, *there is* $X \in \Lambda$ *such that* Proof

$$F\ulcorner X \urcorner =_{\beta\eta} X$$

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,

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

□

Note the difference between Th. 129 and Lemma 126. 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 130. *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
- $P\ulcorner P \urcorner = \ulcorner\ulcorner 1 \urcorner\urcorner$ *and* $P\ulcorner\ulcorner n \urcorner\urcorner = \mathbf{0}$ *otherwise*
- $P\mathbf{0} = \ulcorner P \urcorner$ *and* $P\ulcorner\ulcorner n \urcorner\urcorner = \ulcorner \mathbf{E} \urcorner$ *otherwise*
- $P\ulcorner\ulcorner n \urcorner\urcorner = \ulcorner\ulcorner n+2 \urcorner\urcorner$
- $P\ulcorner\ulcorner n \urcorner\urcorner = P\ulcorner\ulcorner n+1 \urcorner\urcorner$
- $P\ulcorner\ulcorner n \urcorner\urcorner = P\ulcorner\ulcorner n+\#P \urcorner\urcorner$
- $P\ulcorner\ulcorner n \urcorner\urcorner = \mathbf{Succ}(P\ulcorner\ulcorner n \urcorner\urcorner)$
- $P\ulcorner\ulcorner n \urcorner\urcorner = \ulcorner P(P\ulcorner\ulcorner n \urcorner\urcorner) \urcorner$
- $\#P = \#P + 1$
- $\#P = \#(P\ulcorner P \urcorner)$
- $\#P = \#\mathbf{K}$

Exercise 131. *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 K_λ^0 is related to the set K_λ. As for K_λ, this set is not λ-definable.

Definition 132. $\mathsf{K}_\lambda^0 = \{\#M|\ M\mathbf{0}$ *has a β-normal form*$\}$

Exercise 133. *Prove that K_λ^0 is not λ-definable. (See sol. 232)*

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-λ-definable.

Definition 134. *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 135 (Rice's theorem)**.** *Let $\mathcal{L} \subseteq \Lambda$ be closed under $\beta\eta$, and let $\{\#M\ |M \in \mathcal{L}\}$ be λ-defined by F. Then, $\mathcal{L}$ is trivial, i.e. either empty or equal to Λ.*

Proof

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 136. *Which ones of these sets are λ-definable? Justify your answer.*

- $\{\#M | M$ *λ-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 λ'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 λ-term step-by-step, computing the result of repeatedly applying the β 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 137. *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 α-conversions.

Exercise 138. *Define* **Beta** *such that* **Beta**$\ulcorner M\urcorner = \ulcorner M'\urcorner$ *where M' is the result of applying $\rightarrow_\beta$ on M in a leftmost fashion (recall Def. 65). When M is in β-normal form, we just let $M' = M$ instead.*

See also Sol. 231.

Exercise 139. *Define* **Eta** *to apply $\rightarrow_\eta$ until η-normal form is reached.*

Exercise 140. *Define* **IsNF** *to check, given $\ulcorner M\urcorner$, whether M is in $\beta\eta$-normal form.*

Exercise 141. *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 α-conversions.)*

Exercise 142. *Define* **IsClosed** *to check, given $\ulcorner M\urcorner$, whether M is in Λ^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 λ-term. Here, by "size" we mean the maximum nesting of λ-abstractions or applications that occur in the syntax of the λ-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 143. *The size of M, written $|M|$ is defined as*

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

Exercise 144. *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 145. *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 146. *Construct another version of* **E** *using the results above (see Lemma 126). Name this variant* **Eval***. Define it so that, when M has no normal form,* $\mathbf{Eval} \ulcorner M \urcorner$ *is unsolvable.* Definition

Exercise 147. *It can be often useful to consider only the λ-terms that produce numerals. To this aim define a* **Term** *operator such that*

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

You might want to start from:

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

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.*

2.11 Summary

The most important facts in this section:

- syntax of the untyped λ-calculus
- how to program in the untyped λ-calculus:
  - encoding numbers, data structures
  - control flow: conditionals, loops, recursion
- well-known combinators (including fixed-point)
- λ-definability
  - constructing a non-λ-definable function
  - non-λ-definable sets, K_λ
  - classical results: parameter lemma, padding lemma, universal program, Kleene's fixed point theorem, Rice's theorem
- intuition underlying the construction of a step-by-step interpreter

Chapter 3

Logical Characterization of Computable Functions

3.1 Primitive Recursive Functions

Lemma 148. *The function $f(n) = 0$ is λ-definable.* Proof

Proof. Take **K0**. □

Lemma 149. *The function $f(n) = n + 1$ is λ-definable.* Proof

Proof. Take **Succ**. □

Lemma 150. *The projection functions $f_i(n_1, \dots, n_k) = n_i$ with $1 \le i \le k$ are λ-definable.* Proof

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

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

Lemma 151. *The λ-definable (partial) functions are closed under composition.* Proof

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

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

where J is the jamming factor $Gx(KI)I$, as per Ex. 103. 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 then 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 $G\ulcorner\!\ulcorner n\urcorner\!\urcorner$ is unsolvable, so J is also unsolvable by Ex. 101, so $M\ulcorner\!\ulcorner n\urcorner\!\urcorner$ is also unsolvable by the same Exercise.
  - If $g(n) = m \in \mathbb{N}$ but $f(m)$ is undefined, then $J = \mathbf{I}$, $G\ulcorner\!\ulcorner n\urcorner\!\urcorner = \ulcorner\!\ulcorner m\urcorner\!\urcorner$, and $F\ulcorner\!\ulcorner m\urcorner\!\urcorner$ is unsolvable. So, $M\ulcorner\!\ulcorner n\urcorner\!\urcorner = J(F\ulcorner\!\ulcorner m\urcorner\!\urcorner) = F\ulcorner\!\ulcorner m\urcorner\!\urcorner$ is unsolvable as well.

$\square$

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

Lemma 152. *The λ-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 λ-definable.

Proof. Easy adaptation of Lemma 151. $\square$

Statement

Lemma 153. *The λ-definable functions are closed under* primitive recursion. *That is, if g, h are λ-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 λ-terms defining g, h. Then f is λ-defined by

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

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

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

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). $\square$

Definition 154. *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:* Definition

- *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 155. *Show that the following functions are in $\mathcal{PR}$.*

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

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

- *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* pair *and* $\mathsf{encode}_{\uplus}$ *functions for pairs and disjoint union (easy), as well as their inverses (not so easy).*

Exercise 156. *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 λ-definable functions. By the lemmata above, each $f \in \mathcal{PR}$ is λ-definable. Clearly, if we take a λ-definable non-total function, this is not in $\mathcal{PR}$, so the λ-definable functions form a larger set that $\mathcal{PR}$.

What if we restrict to *total* λ-definable functions, then? We can prove that the set of total λ-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. 154. As we did for the λ-calculus we can enumerate this $\mathcal{PR}$ language using the 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 λ-defined! We just need to write an interpreter for this $\mathcal{PR}$ language in the λ calculus in order to define f. This can done as we did for $\mathbf{E}$.

Exercise 157. *Define the "$\mathcal{PR}$ language" as we did for Λ, and an encoding $\mathcal{PR} \leftrightarrow \mathbb{N}$. Then, λ-define an interpreter for this $\mathcal{PR}$ language.*

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

Statement

Theorem 158. *The set of λ-definable functions is strictly larger than $\mathcal{PR}$ functions.*

3.1.1 Ackermann's Function

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

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

[also see Cutland page 46]

Exercise 159. *Show that* ack *is λ-definable.*

Note the "double recursion" in the last line. This is not a problem in the λ 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 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 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 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 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 160. *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 λ-definable functions?*

3.2 General Recursive Functions

Exercise 161. *Let $f(x, y)$ be a total λ-definable function. Show that*

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

is a total λ-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 162. *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 λ-definable functions. Basically, the problem boils down to constructing an interpreter of the λ 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. 146. 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 $\to_\beta$ step, is actually in $\mathcal{PR}$.

Lemma 163. *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 λ-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 164. *Show that the function* $\mathsf{extract}(\#\ulcorner n \urcorner) = n$ *is in $\mathcal{PR}$. (Make it work on all possible α-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 $\to_\beta$ 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

Definition 165. *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

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

where the above, intuitively, means "the least $y \in \mathbb{N}$ such that $f(x, y) = 0$, provided f is defined for smaller values of y". More formally,

$$g(x) = \min\{y \mid f(x, y) = 0 \wedge \forall z < y.f(x, z) \text{ defined}\}$$

Note that $g(x)$ is undefined whenever the set above is empty: this may happen because e.g. $f(x, y) > 0$ for all y, or even because $f(x, y) > 0$ for $y \in \{0 \dots 4\}$ but $f(x, 5)$ is undefined. In that case, g is a strictly partial (i.e. non-total) function. This definition is naturally extended to n-ary functions in $\mathbb{N}^k \rightsquigarrow \mathbb{N}$.

Exercise 166. *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 pair*.*

Lemma 167. *The λ-definable functions are closed under unbounded minimalisation.*

Statement

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

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

□

Lemma 168. *The set of recursive functions $\mathcal{R}$ is included in the set of λ-definable functions.*

Proof. Immediate by all the lemmata above. □

Theorem 169. *The set of λ-definable functions is exactly the same as the set of recursive functions $\mathcal{R}$.* Statement

Proof. We already proved that each $f \in \mathcal{R}$ is λ-definable. Now we prove that if f is λ-definable (say by F), then $f \in \mathcal{R}$. By Exercise 155, $\mathsf{proj1}, \mathsf{proj2} \in \mathcal{PR}$, so by Lemma 163, and Exercise 164, we can define the following functions in $\mathcal{R}$:

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

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

- If $f(y)$ is undefined, then $F\ulcorner\ulcorner y\urcorner\urcorner$ has no normal form. So, no matter what $\mathsf{proj2}(n)$ evaluates to, the function steps will perform that many β-steps on x, but will not reach the index of a β-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 μn 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 $F\ulcorner\ulcorner y\urcorner\urcorner$ has as its normal form the numeral $\ulcorner\ulcorner z\urcorner\urcorner$. Define k as the number of leftmost $\rightarrow_\beta$ steps needed to reach normal form. Therefore, $\mathsf{eta}(\mathsf{steps}(\#(F\ulcorner\ulcorner y\urcorner\urcorner), k))$ will completely evaluate $F\ulcorner\ulcorner y\urcorner\urcorner$ until $\beta\eta$ normal form, producing the index of a λ-term M, which is an α-conversion[1] of $\ulcorner\ulcorner z\urcorner\urcorner$. The minimalisation operator μn will try each $n \in \mathbb{N}$, from 0 upwards.

  – When $0 \le n < \mathsf{pair}(\#M, k)$, we show that $\mathsf{and}(A, B)$ returns "false" (nonzero), so that the minimalisation will try the next

[1] Recall Exercise 117. 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 beta function to choose exactly the variables we use in the definition of $\ulcorner z\urcorner$.

n. By contradiction, assume that $\mathsf{and}(A,B)$ returns "true". This means that A and B are both "true". Since A is "true", $\mathsf{eta}(\mathsf{steps}(\#(F\ulcorner\ulcorner y\urcorner\urcorner), \mathsf{proj2}(n)))$ is the index i of a numeral, hence the index of a normal form of $F\ulcorner\ulcorner y\urcorner\urcorner$. 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 $n = \mathsf{pair}(\mathsf{proj1}(n), \mathsf{proj2}(n)) = \mathsf{pair}(\#M, \mathsf{proj2}(n)) \geq \mathsf{pair}(\#M, k)$, contradicting $n = \mathsf{pair}(\mathsf{proj1}(n), \mathsf{proj2}(n)) < \mathsf{pair}(\#M, k)$.

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

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

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

Exercise 170. *Provide an alternative proof for Th. 169, 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 eval *using g and h.*

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

[3] Recall we require $\mathsf{isNumeral}$ to return "true" on all α-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 171. *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 169. Indeed, the definition of f in that proof mentions a single μn operator, using only primitive recursive functions inside of the μn, as well as outside of it. So T and U simply are defined in that way. The integer i is instead the index $\#F$ for some λ-term F that defines the function $f \in \mathcal{R}$. This F indeed exists by Lemma 168. $\square$

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 172. *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 173. *A function f is FOR-definable if there is some FOR-program that has semantics f.*

Theorem 174. *The set of FOR-definable functions is exactly $\mathcal{PR}$.* **Statement**

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 pair 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 175. *Define the semantics of WHILE programs.*

Definition 176. *A function f is WHILE-definable if there is some WHILE-program that has semantics f.*

Theorem 177. *The set of WHILE-definable functions is exactly $\mathcal{R}$.* **Statement**

Proof. (sketch)
($\subseteq$): a WHILE interpreter can be written in the λ-calculus (long exercise). So, each WHILE-definable function is in $\mathcal{R}$ by Th. 169.
($\supseteq$): Let $f \in \mathcal{R}$. We must find a WHILE program defining f. Take T,U as in Th. 171. By Th. 174, T and U are FOR-definable, hence WHILE-definable. Following again Th. 171, all we have to do is to "add the missing μn" and compose T and U so to actually compute f. A single `while` construct is sufficient to try each $n \in \mathbb{N}$, thus emulating the μn operator. $\square$

Theorem 178. *Every WHILE-definable function can be WHILE-defined by a program having a single* `while` *loop.*

Proof. Direct consequence of Th. 171. $\square$

3.5 Church's Thesis

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

- all λ-definable functions are definable in the language L; e.g. you can write an interpreter for the λ-calculus in L
- all L-definable functions are definable in the λ-calculus; e.g. you can write an interpreter for L in the λ-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 λ-calculus (or Java, or Turing machines, or $\langle$insert your favourite programming language here$\rangle$).

Notable languages *not* equivalent to the λ-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 λ-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 λ-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

3.6 Summary

The most important facts in this section:

- primitive recursive functions (subset of total functions)

- general recursive functions (subset of partial functions)

- λ-definable functions coincide with general recursive functions

- proof of $\supseteq$
- intuition about the proof of $\subseteq$

Chapter 4

Classical Results

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

- For instance, $M = \lambda n.\, n\,\mathbf{K}\,\mathbf{\Theta}$ does not λ-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 $\mathbf{KI}$) but not a numeral.

- Another example is $N = \lambda n.\, n\,\mathbf{K}\,(\lambda y.\, yy)$. Assume this λ-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 λ-term to a partial function. So, ϕ_i shall be the function related to the program M having index i.

Definition 179. *$\phi_i(x) = y$ iff $M\ulcorner x\urcorner =_{\beta\eta} \ulcorner y\urcorner$ where $\#M = i$* **Definition**

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 ϕ_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. Essentially, the definition above is using Option 1 from Def. 99 to model undefinedness.

Lemma 180. *f is λ-definable iff $f = \phi_i$ for some i*

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

- If $f(x)$ is undefined, then we have that $F\ulcorner\!\ulcorner x\urcorner\!\urcorner$ is unsolvable, so it has no normal form, so it is $\neq_{\beta\eta} \ulcorner\!\ulcorner y\urcorner\!\urcorner$ for all y, and $\phi_i(x)$ is then undefined.
- If $f(x) = y \in \mathbb{N}$, then $F\ulcorner\!\ulcorner x\urcorner\!\urcorner = \ulcorner\!\ulcorner f(x)\urcorner\!\urcorner$, 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 λ-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. 147: $\mathbf{Term}\ulcorner O\urcorner$ 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 λ-defines f. When $f(x)$ is undefined, then $M\ulcorner\!\ulcorner x\urcorner\!\urcorner$ has no numeral as normal form, and thus J is unsolvable. Otherwise, when $f(x)$ is defined, $M\ulcorner\!\ulcorner x\urcorner\!\urcorner$ has $\ulcorner\!\ulcorner f(x)\urcorner\!\urcorner$ as its normal form, and thus $J = \mathbf{I}$. In this case, $F\ulcorner\!\ulcorner x\urcorner\!\urcorner = M\ulcorner\!\ulcorner x\urcorner\!\urcorner = \ulcorner\!\ulcorner f(x)\urcorner\!\urcorner$. □

4.1 Padding Lemma

Statement

Theorem 181 (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 λ-calculus. □

Also see [Cutland]

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

Statement

Theorem 182 (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 λ-calculus. □

Also see [Cutland]

Exercise 183. *Show that* pad *and* s^n_m *are primitive recursive functions.*

A typical application of the parameter theorem is to construct functions "returning indexes of programs", e.g. a *total* f such that

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

where g is a (partial) *recursive* function. Note that f is required to be total, but g is not required to be such, so g can be non-total.

The above equation can be satisfied by e.g.:

$$f(x) = \#(\lambda y.\, G)$$

where G λ-defines g. The above f can be constructed by applying the usual syntax manipulation functions $\mathbf{App}, \mathbf{Lam}, \ldots$, so it is recursive.

More generally, without exploiting features of the λ-calculus, one can instead build f as follows. First, fix an index of g, i.e. pick some j such that $\phi_j = g$. Then, let $f(x) = s(j, x)$. Hence, by using the parameter lemma:

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

Convention. The above technique is frequently used when dealing with m-reductions (Sect. 4.6.2). Instead of repeating all the steps shown above, we shall simply write $f(x) = \#(\lambda y.\, G)$ without mentioning the implicit use of the parameter lemma.

4.3 Universal Program

Theorem 184 (Universal Program)**.**
The partial function $f(x, y) = \phi_x(y)$ is recursive.
This can be generalized to n-ary partial functions. **Statement**

Proof. Easy adaptation of the Universal Program for the λ-calculus. We actually described such an f in the proof of Th. 169. □

Also see [Cutland]

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

Theorem 185 (Kleene's Fixed Point Theorem (a.k.a. Second Recursion Theorem))**.**
For each total computable function f, there is some $n \in \mathbb{N}$ such that **Proof**

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

Proof. We adapt the proof of Th. 129. By Th. 184, 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

Definition 186. *A set $A \subseteq \mathbb{N}$ is recursive iff the function χ_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

Definition 187. *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, λ-definable, WHILE-definable, ... These adjectives are equivalent. Recursively enumerable sets are said to be semi-decidable, semi-computable, ... instead.

Exercise 188. *Prove that the following properties of a set A are equivalent.*

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

inition **Definition 189.** $\mathsf{K} = \{n | \phi_n(n) \textit{ is defined}\}$

Lemma 190. $\mathsf{K} \notin \mathcal{R}$ Proof

Proof. Similar the the argument for K_λ. 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 191. $\mathsf{K} \in \mathcal{RE}$ Proof

Proof. By Th. 184, $f(n) = \phi_n(n)$ is in $\mathcal{R}$, and clearly $\mathsf{dom}(f) = \mathsf{K}$. □

Lemma 192. $A \in \mathcal{R} \implies A \in \mathcal{RE}$ Proof

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

Lemma 193. $A \in \mathcal{RE} \wedge \bar{A} \in \mathcal{RE} \implies A \in \mathcal{R}$ Proof

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 194. *Construct the λ-term of the verifier used in the proof above.*

Lemma 195. $A \in \mathcal{RE} \wedge \bar{A} \in \mathcal{RE} \iff A \in \mathcal{R}$ Proof

Proof. Immediate by the lemmata above. □

Lemma 196. $\bar{\mathsf{K}} \notin \mathcal{RE}$ **Proof**

Proof. Immediate by Lemma 195 and $\mathsf{K} \in \mathcal{RE} \setminus \mathcal{R}$. □

Proof **Lemma 197.** *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. $A = \{n | \exists m. \mathsf{pair}(n, m) \in B\}$ *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 $f(\mathsf{pair}(y, k)) = y$, 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 $B = \{\mathsf{pair}(f(x), x) | x \in \mathbb{N}\}$. 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 $\{n | \exists m. \mathsf{pair}(n, m) \in B\}$.

($3 \implies 4$) 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 λ-calculus using the step-by-step interpreter. Let j be the index of that λ-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 $i = \mathsf{pair}(x, y)$ 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 198. $\mathsf{K}_\lambda \in \mathcal{RE}$

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

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

B is recursive, since it checks only for a given number of steps, so by Lemma 197, $\mathsf{K}_\lambda \in \mathcal{RE}$. □

Lemma 199. $\bar{\mathsf{K}}_\lambda \notin \mathcal{RE}$

Proof. Immediate by Lemma 195 and $\mathsf{K}_\lambda \in \mathcal{RE} \setminus \mathcal{R}$. □

Lemma 200. $\bar{\mathsf{K}} \notin \mathcal{RE}$ **Proof**

Proof. Immediate by Lemma 195 and $\mathsf{K} \in \mathcal{RE} \setminus \mathcal{R}$. □

4.6 Reductions

4.6.1 Turing Reduction

Sometimes, it is interesting to pretend that in the λ-calculus some function or set is λ-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 λ-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 λ-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 201. *When we extend the λ-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

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

The notion of $(\lambda + A)$-definability is then derived from the notion of λ-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 202 (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 203. *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 $\mathsf{K}_\lambda \equiv_T \bar{\mathsf{K}}_\lambda$*
- *$\mathsf{K}_\lambda \equiv_T \mathsf{K}$*
- *If $A, B \leq_T C$, then $A \cup B \leq_T C$*
- *If $A, B \leq_T C$, then $\{\mathsf{pair}(x, y) \mid x \in A \wedge y \in B\} \leq_T C$*
- *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* λ-definable, by showing that $B \leq_T A$ for some B that is known to be λ-*un*definable.

Exercise 204. *Consider the $(\lambda + \mathsf{K}_\lambda)$-calculus. Can every partial function $f \in \mathbb{N} \rightsquigarrow \mathbb{N}$ be $(\lambda + \mathsf{K}_\lambda)$-defined?*

4.6.2 Many-one Reduction

Another useful notion of reduction is the following:

Definition 205 (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 ("a m-reduction") such that*

Definition

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

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

Exercise 206. *Prove that the above requirement on f is equivalent to require both*

Statement

- *for all n, $n \in A \implies f(n) \in B$*
- *for all n, $n \notin A \implies f(n) \notin B$*

Lemma 207. $A \leq_m B \implies A \leq_T B$

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

Lemma 208. $A \leq_m B \iff \bar{A} \leq_m \bar{B}$

Proof

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

Lemma 209. *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

Proof. The first part is a direct consequence of $\leq_m$ implying $\leq_T$, and Ex.203. 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 210. $A \leq_m \mathsf{K} \implies A \in \mathcal{RE}$

Proof

Proof. Immediate from the lemma above. □

Exercise 211. *Prove the following:*

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

Statement

Lemma 212. *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

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

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

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: let us check that it is an m-reduction from A to K.

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

□

Proof

Lemma 213. *$A \in \mathcal{RE}$ if and only if $A \leq_m \mathsf{K}$*

Proof. Immediate from the lemmata above. □

Exercise 214. *Prove that $\mathsf{K} \equiv_m \mathsf{K}_\lambda$. From this, deduce that $A \in \mathcal{RE}$ if and only if $A \leq_m \mathsf{K}_\lambda$.*

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 215 (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,

Proof

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

Proof. Since f is recursive, f is λ-defined by some F. Since $A \in \mathcal{RE}$, we can *not* have $\bar{\mathsf{K}} \leq_m A$: this will be used below.

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

  We now obtain a contradiction by proving $\bar{\mathsf{K}} \leq_m A$. The reduction h is the following:

  $$h(n) \quad = \# \left(\lambda x. \begin{cases} \mathit{undefined} & \text{if } \phi_n(n) \text{ halts in (at most) } x \text{ steps} \\ f(x) & \text{otherwise} \end{cases} \right)$$

  Note that the above h is well-defined, since the condition "…halts in x steps" is decidable, and $f \in \mathcal{R}$. So, $h \in \mathcal{R}$ is total recursive.

  Let us check h is indeed a reduction:

  – If $n \notin \mathsf{K}$, then $\phi_{h(n)} = f$ since "$\phi_n(n)$ halts in x steps" is always false. So, $\phi_{h(n)} \in \mathcal{F}$, hence $h(n) \in A$

  – If $n \in \mathsf{K}$, we have that $\phi_n(n)$ halts in, say, j steps. So, for $x < j$ we have $\phi_{h(n)}(x) = f(x)$, while for $x \geq j$ we have that $\phi_{h(n)}(x)$ is undefined. This implies that $\phi_{h(n)}$ is a finite restriction of f: $\phi_{h(n)}$ finite and $\phi_{h(n)} \subseteq f$. By assumption, no such finite restriction of f belongs to $\mathcal{F}$. Hence, $h(n) \notin A$.

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

  We now obtain a contradiction by proving $\bar{\mathsf{K}} \leq_m A$. The reduction h is the following:

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

  Such an h is well-defined because $\mathsf{dom}(g)$ is finite (hence decidable) and K is $\mathcal{RE}$, so we have a semi-verifier for the property "$x \in \mathsf{dom}(g)$ or $n \in \mathsf{K}$" which is what we need above. Indeed, $h(n)$ is a total recursive function.

  Let us check h is indeed a reduction:

  – If $n \notin \mathsf{K}$, then $\phi_{h(n)}(x) = f(x)$ when $x \in \mathsf{dom}(g)$, and undefined otherwise. So, $\phi_{h(n)}$ is f restricted to $\mathsf{dom}(g)$, which implies $\phi_{h(n)} = g$ since $g \subseteq f$. Therefore, $\phi_{h(n)} \in \mathcal{F}$. We conclude $h(n) \in A$.

  – If $n \in \mathsf{K}$, then $\phi_{h(n)}(x) = f(x)$ for all x. This implies $\phi_{h(n)} = f \notin \mathcal{F}$, hence $h(n) \notin A$

□

Common Use. Usually, Rice-Shapiro is used to prove that some set is **not** $\mathcal{RE}$. This can be done in two ways, depending whether we use the ($\Rightarrow$) or ($\Leftarrow$) direction of the theorem. We summarize these typical arguments below. Let $A = \{n|\phi_n \in \mathcal{F}\}$ with $\mathcal{F} \subseteq \mathcal{R}$.

- Rice-Shapiro ($\Rightarrow$): to prove $A \notin \mathcal{RE}$ it suffices to
  - pick some $f \in \mathcal{F}$, and
  - show that *all* the finite restrictions g of f do not belong to $\mathcal{F}$.
- Rice-Shapiro ($\Leftarrow$): to prove $A \notin \mathcal{RE}$ it suffices to
  - pick some $f \notin \mathcal{F}$, and
  - pick some finite restriction g of f which belongs to $\mathcal{F}$.

4.8 Rice's Theorem

Statement

Theorem 216. *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. 135, but we instead apply Rice-Shapiro (Th. 215). We have $A, \bar{A} \in \mathcal{R}$, so $A, \bar{A} \in \mathcal{RE}$. Let ϕ_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 217. *For any of these sets, state whether the set is $\mathcal{R}$, or $\mathcal{RE} \setminus \mathcal{R}$, or not in $\mathcal{RE}$.*

- $\mathsf{K}_\lambda \cup \{5\}$
- $\{1, 2, 3, 4\}$

- $\{n | \phi_n(2) = 6\}$
- $\{n | \exists y \in \mathbb{N}.\, \phi_n(y) = 6\}$
- $\{n | \forall y \in \mathbb{N}.\, \phi_n(y) = 6\}$
- $\{n | \phi_n(n) = 6\}$
- $\{n | \mathsf{dom}(\phi_n) \textit{ is finite}\}$
- $\{n | \mathsf{dom}(\phi_n) \textit{ is infinite}\}$
- $\{n | \phi_n \textit{ is total}\}$
- $\{n + 4 | \mathsf{dom}(\phi_n) \textit{ is finite}\}$
- $\{ \lfloor 100/(n+1)^2 \rfloor \mid \mathsf{dom}(\phi_n) \textit{ is infinite}\}$
- $A \cup B$, $A \cap B$, $A \setminus B$ *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* $A, B \in \mathcal{RE}$
- $\{n \mid \forall m.\, \mathsf{pair}(m, n) \in A\}$ *where* $A \in \mathcal{RE}$
- $\{\mathsf{pair}(n, m) | \mathsf{pair}(m, n) \in A\}$ *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 218. *(Hard) Show that* $f \in \mathcal{R}$ *where*

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

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

Exercise 219. *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 220. *Given a λ-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 $A = \{\mathsf{pair}(\#L, \#J) \mid \phi_{\#L} = \varphi_{\#J}\}$. Is $A \in \mathcal{R}$? Is it $\mathcal{RE}$?*

Exercise 221. *Let $A \in \mathcal{R}$, and $B = \{n \mid \exists m. \mathsf{pair}(n, m) \in A\}$. We know that $B \in \mathcal{RE}$ by Lemma 197. Can we conclude that $B \in \mathcal{RE} \setminus \mathcal{R}$?*

Exercise 222. *Consider the following formal language: (a, b are two constants)*

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

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

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

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.)

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

Appendix A

Solutions

Solution 223.
$$\mathsf{encode}_{\uplus}^{-1}(n) = \langle n \ \mathsf{mod}\ 2,\ \left\lfloor \frac{n}{2} \right\rfloor \rangle$$

Solution 224. *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 Turing. Church 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 225. *By contradiction, assume* $\mathbf{T} =_{\beta\eta} \mathbf{F}$. *Clearly,* $\mathbf{T}$ *and* $\mathbf{F}$ *are* $\beta\eta$*-normal forms. By Th. 79, we have* $\mathbf{T} \to^*_{\beta\eta} \mathbf{F}$. *Since* $\mathbf{T}$ *is a normal form, this is not possible unless* $\mathbf{T} =_\alpha \mathbf{F}$, *which is clearly not the case* $\square$

Solution 226. *Here are many useful combinators:*

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

$\mathbf{LimMin}\,F\,Z\ulcorner n\urcorner$ *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.*

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

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 λ*-term defining the* pair *function. The definition is straightforward from the formula for* pair.

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

We compute the inverse of $c = \mathsf{pair}(n, m)$ 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 30, we only need to search for $n, m \in \{0..c\}$, so we limit our search to that square.

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

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

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

Solution 227.

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

Solution 228.

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

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

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

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

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

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

- $G\ulcorner \lambda x.\,\lambda y.\,M\urcorner = \ulcorner \lambda y.\,\lambda x.\,M\urcorner$
  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* α*-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 229.

$\mathbf{Sd} = \lambda n\ v\ a\ l.\ \mathbf{Case}\ n\ v\ O$
$O = \lambda m.\ \mathbf{Case}\ m\ A\ L$
$A = \lambda x.\ a\ (\mathbf{Proj1}\ x)\ (\mathbf{Proj2}\ x)$
$L = \lambda x.\ l\ (\mathbf{Proj1}\ x)\ (\mathbf{Proj2}\ x)$

Solution 230.

$$\mathbf{App} = \lambda m\, n.\, \mathbf{InR}(\mathbf{InL}(\mathbf{Pair}\ m\ n))$$

Solution 231. *The following program follows the algorithm for the leftmost-outermost strategy of Def. 65. The "shallow decoder"* **Sd** *of Ex. 121 is also exploited.*

$$\begin{aligned}
&\mathbf{Lam} = \lambda i\ m.\ \mathbf{InR}\ (\mathbf{InR}\ (\mathbf{Pair}\ i\ m))\\
&\mathbf{Beta} = \lambda n.\ \mathbf{Be}\ n\ \mathbf{T}\\
&\mathbf{IsBetaNF} = \lambda n.\ \mathbf{Be}\ n\ \mathbf{F}\\
&\mathbf{Be} = \mathbf{\Theta}(\lambda b\ n.\ \mathbf{Sd}\ n\ V\ A\ L)\\
&V = \lambda i.\ \mathbf{Cons}\ n\ \mathbf{T}\\
&L = \lambda i\ m.\ b\ m\ \mathbf{F}\ (\mathbf{Cons}\ n\ \mathbf{T})\ (\mathbf{Cons}\ (\mathbf{Lam}\ i\ (b\ m\ \mathbf{T}))\ \mathbf{F})\\
&A = \lambda m\ o.\ \mathbf{Sd}\ m\ (\mathbf{K}\ M)\ (\mathbf{K}\ (\mathbf{K}\ M))\ L'\\
&L' = \lambda i\ m'.\ \mathbf{Cons}(\mathbf{Subst}\ (\mathbf{InL}\ i)\ o\ m')\ \mathbf{F}\\
&M = b\ m\ \mathbf{F}\ \Big(b\ o\ \mathbf{F}\ (\mathbf{Cons}\ n\ \mathbf{T})\ (\mathbf{Cons}\ (\mathbf{App}\ m\ (b\ o\ \mathbf{T}))\ \mathbf{F})\Big)\\
&\qquad\quad \Big(\mathbf{Cons}\ (\mathbf{App}\ (b\ m\ \mathbf{T})\ o)\ \mathbf{F}\Big)
\end{aligned}$$

Solution 232. *By contradiction, suppose K^0_λ is λ-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 λ-defines K_λ, which is a contradiction. ☐

Solution 233. *Take* $\mathbf{Pad} = \lambda n.\, \mathbf{App}\,\ulcorner\mathbf{I}\urcorner\,n$. *Then,* $\mathbf{Pad}\ulcorner M\urcorner = \ulcorner\mathbf{I}M\urcorner$, *and we have*

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

A.1 More Proofs

Here we establish Church-Rosser for $\to_\beta$.

Definition 234. *We define $\to_p$ as a "parallel" variant of $\to_\beta$. Its inductive definition comprises the "up-to-α" rule and the following ones.*

$$\frac{}{M \to_p M} \tag{A.1}$$

$$\frac{M \to_p M' \quad N \to_p N'}{MN \to_p M'N'} \tag{A.2}$$

$$\frac{M \to_p M'}{\lambda x.\, M \to_p \lambda x.\, M'} \tag{A.3}$$

$$\frac{M \to_p M' \quad N \to_p N'}{(\lambda x.\, M)N \to_p M'\{N'/x\}} \tag{A.4}$$

Lemma 235. *While $\to_\beta$ and $\to_p$ are different relations, they have the same transitive reflexive closure, i.e. $\to_\beta^* = \to_p^*$.*

Proof. By simple induction. □

Lemma 236. *The (one-step, parallel) relation $\to_p$ is Church-Rosser:*

$$\forall M\, M_1\, M_2.\, M \to_p M_1 \land M \to_p M_2 \implies \exists N.\, M_1 \to_p N \land M_2 \to_p N$$

Proof. (Sketch) By induction and case analysis. Checking all the pairs of rules (A.1),...,(A.4) suffices. □

Lemma 237. *The (many-steps, parallel) relation $\to_p^*$ is Church-Rosser:*

$$\forall M\, M_1\, M_2.\, M \to_p^* M_1 \land M \to_p^* M_2 \implies \exists N.\, M_1 \to_p^* N \land M_2 \to_p^* N$$

Proof. By Lemma 236 and induction on the number of steps. □