Short Lecture Notes — Computability (2008–2013)

Roberto Zunino
Dipartimento di Matematica
Università degli Studi di Trento
roberto.zunino@unitn.it

Last update — 29 Apr 2016

General Information

These notes are meant to be a short summary of the topics covered in my *Computability* course kept in 2008, 2010, 2011, 2012, and 2013 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 only little 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 markers for those definitions, statements and proofs which will be asked during the 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 and its direct consequences.

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).

Definition

- Definitions to remember are marked as this.

Also, please remember the following, taken from ESSE3:

> *Prerequisites*: understanding of mathematical proofs; basic notions of set theory; programming skills.

Exam questions often implicitly test you on the above skills as well.

Roberto Zunino

Contents

Chapter 1

Introduction

What is Computability?

Broadly speaking, Computability is a discipline which studies the theoretical limits of computer programming. A main purpose of it is to understand which tasks can be performed by a program (hence "computable"), and which instead are so hard that no program is able to perform them. As such, its results can be roughly divided in *positive* (proving that something can be computed by a program) and *negative* (proving that no such program exists).

Positive results are the easiest to establish. Indeed, it is sufficient to exhibit a program and justifying that it really solves a given task ("this procedure correctly sorts the input array"). Negative results are much harder to prove, instead, since they require to rule out the possibility that such a program might exist. Establishing a negative result can not obviously be done by examining every single program, since we have infinitely many of them. Since a non trivial approach is needed for negative results, these results in Computability are by far the most important ones.

Negative results are also made strong by the fact that Computability theory puts no constraints on the amount of resources which a program can demand. A program is allowed to require any amount of memory, including those which are impossible to obtain in practice (e.g. a terabyte for each atom in the known universe). Similarly, a program is allowed to run for a huge amount of time before it completes (e.g. the amount of time between the Big Bang and now). The main point in accepting these vastly inefficient programs is to make strong our negative results. Indeed, if we can prove that a task can not be solved by any program, even in presence of unlimited resources, then we can surely infer that no real-world computer can hope to

solve that task. In other words, when we prove a task to be non computable, we know it will never be solved by a computer, no matter how powerful the computing hardware can become in the future.

These notes are organized as follows.

In Chapter 1 we provide some mathematical preliminaries. There we also discuss the diagonalization proof technique, and use it to construct the first example of a non-computable task.

In Chapter 2 we focus on one specific, albeit unusual, programming language: the untyped λ-calculus. We first use it to establish some positive results. Some of these are expected (e.g., multiplication is computable), while others are less so (self-interpreter, Kleene's fixed point theorem). Finally, a strong general tool to prove negative results is provided: Rice's theorem.

In Chapter 3 we start abstracting away from the choice of a programming language. We shall characterize there the set of those functions which are computable, i.e. can be implemented by some program. We shall provide a definition of computable function which does not rely on the λ-calculus, yet prove it equivalent to the one given for that language. We conclude by stating that a similar result also holds in all the common programming languages.

In Chapter 4 we extend the theory of computable functions. We re-state Kleene's fixed point theorem and Rice's theorem in a more general setting. We then proceed to investigate recursive sets, recursively enumerable sets, and m-reductions. We conclude by stating the Rice-Shapiro theorem, a powerful general tool to prove that some sets are not recursively enumerable.

1.1 Logic Notation

We shall now 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 some exercises in this section.

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

$p \vee \neg p$	*excluded middle*
$\neg(p \vee q) \iff (\neg p \wedge \neg q)$	*De Morgan*
$\neg(p \wedge q) \iff (\neg p \vee \neg q)$	*De Morgan*
$(p \implies q) \iff (\neg p \vee q)$	*classical implication*
$(p \wedge q \implies r) \iff (p \implies (q \implies r))$	*export/import*
$(p \implies q) \iff (\neg q \implies \neg p)$	*contraposition*
$(p \iff q) \iff (\neg p \iff \neg q)$	*contraposition*
$(p \wedge q) \vee r \iff (p \vee r) \wedge (q \vee r)$	*distribution*
$(p \vee q) \wedge r \iff (p \wedge r) \vee (q \wedge r)$	*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)*
$\exists y. \forall x.\, p(x, y) \implies \forall x. \exists y.\, p(x, y)$	
$\forall x. \exists y.\, p(x, y) \not\implies \exists y. \forall x.\, p(x, y)$	
$\exists! x.\, p(x) \iff \exists c.\, (\forall x.\, (p(x) \iff x = c))$	*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.

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

Cartesian product We shall use ordered pairs $\langle x, y \rangle$, as well as ordered tuples. The cartesian product is then defined in the usual way.

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

Note that cartesian product can be used to model some kinds of data in programming. For instance, imagine that a sensor can be queried so to obtain the current temperature and pressure. In that case we can represent the set of possible answers as $\mathsf{Temp} \times \mathsf{Press}$. If both are modelled as real numbers, then $\mathbb{R} \times \mathbb{R}$ represents the result.

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

Disjoint union Suppose that an area is filled with two kinds of sensors. One kind is able to measure the temperature, while the other one measures the pressure. Querying any one of them results in an answer such as "`temp: 300.4`" or "`press: 1334.12`". We could model the whole set of possible answers using $\mathsf{Temp} \cup \mathsf{Press}$, but if both are real numbers this would result in $\mathbb{R} \cup \mathbb{R} = \mathbb{R}$ which does not really capture the information present in the answers. Indeed, beyond the numeric values, the answers reveal more

information, i.e. whether that value is a temperature or a pressure. A better modelling could then use two "tags" to distinguish the two copies of $\mathbb{R}$, as follows:

$$\{\langle \mathsf{temp}, x\rangle | x \in \mathbb{R}\} \cup \{\langle \mathsf{press}, x\rangle | x \in \mathbb{R}\}$$

where $\mathsf{temp}, \mathsf{press}$ are distinct constants, whose value is immaterial, and whose only purpose is to force the above sets apart. We can indeed pick these tags to be 0 and 1. The disjoint union above will be denoted with $\mathsf{Temp} \uplus \mathsf{Press}$, following the general definition below.

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

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

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

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

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

Definition 5. *A function f is injective (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*

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

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

For this course, we shall use

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

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")*

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

  Intuition: 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:*

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

  Then, we let $\mathbb{N}$ *be the least prefixed point of* $\hat{R}$*, i.e.*

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

  Intuition: 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.*

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

  Intuition: $\mathbb{N}$ *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,*

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

  Intuition: 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* $\mathbf{1}$ *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:

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

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 the following properties hold:*

$$p(0)$$

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

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, so we now prove $\mathbb{N} \subseteq Y$. By hypothesis, we have

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

the above implies

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

which together with

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

implies $\mathbb{N} \subseteq Y$. □

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.*

$$\frac{}{6} \qquad \frac{n \quad m}{n+m} \qquad \frac{n \quad m}{n \cdot m} \qquad \frac{n \quad m}{n^{m+1}}$$

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

In programming, we are used to exploit different data types such as strings, trees, lists,... to represent information. Some of these data types admit only a finite number of values (e.g. booleans only admit two: true and false), while others do not (strings can be of any length, for instance). We now focus on those data types which admit an infinite number of values: these are typically defined inductively exploiting disjoint union and cartesian product. For instance:

$\mathbb{L} \simeq \mathbf{1} \uplus (\mathbb{N} \times \mathbb{L})$	lists
$\mathbb{T} \simeq \mathbb{N} \uplus (\mathbb{T} \times \mathbb{T})$	binary trees
$\mathbb{T} \simeq \mathbb{N} \uplus \Big((\mathbb{T} \times \mathbb{T}) \uplus (\mathbb{T} \times \mathbb{T} \times \mathbb{T}) \Big)$	trees with branching 2 or 3

While having several different data types is convenient in programming, it is useful to be able to represent all of them in a common format. Indeed, it is common for a program to *serialize* a value of any such data type into a file ("save"), or to *deserialize* a file back to the original value ("load"). Files can then be transmitted and duplicated without caring about the actual value inside it, or even its data type. In a sense, they are universal information carriers.

In computability theory, a similar technique is used to represent information using a common format. However, instead to convert data into a file, i.e. a byte string, it is common to convert data into a natural number. The difference is actually minimal, since a natural number can be arbitrarily large, therefore it can have an arbitrarily large number of decimal digits (for instance), so it can represent as much information as needed. Both files and natural numbers can be used as universal information carriers. In computability theory, we tend to prefer using natural numbers since we do want to use numbers for general programming anyway, hence using them also as data carriers is the most economic choice: in our theory we shall never need to consider other data types but natural numbers.

Below, we show that any data type defined in terms of disjoint union or cartesian product (and having infinite values) can be encoded into natural numbers in a bijective fashion. Informally, given any "common" data type, there is a function which is able to *serialize* any value of that type into a natural number, in such a way that it is possible to *deserialize* it back to the original value. Further, the encoding is surjective, so that actually any number can be deserialized into some value. Surjectivity is not strictly needed, but it simplifies the theory, since we shall never need to consider those cases in which the deserialization fails.

1.4.1 Bijections from $\mathbb{N} \uplus \mathbb{N}, \mathbb{N} \times \mathbb{N}, \mathbb{N}^*, \ldots$ to $\mathbb{N}$

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.

Definition

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

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. 318.*

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

Definition

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

This can be visualized as follows:

	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	...							

Indeed, by inspecting the table above it is easy to check that

$$\mathsf{pair}(\langle 0, x \rangle) = \sum_{i=0}^{x} i = \frac{x \cdot (x+1)}{2}$$

Also, by inspection we get

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

Hence, in the general case

$$\mathsf{pair}(\langle n, m\rangle) = \mathsf{pair}(\langle n-1, m+1\rangle) + 1 = \mathsf{pair}(\langle n-2, m+2\rangle) + 2 = \cdots$$
$$= \mathsf{pair}(\langle 0, m+n\rangle) + n = \tfrac{(n+m)(n+m+1)}{2} + n$$

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* $\mathbb{Q}$. *(Hint: represent* $\mathbb{Q}$ *as the set of fractions* p/q *with* p, q *coprime.)*

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:*

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

Lemma 30.

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

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, since any structured data can be encoded into those. Hence, we describe the mapping between inputs and outputs of a given program using a partial function $\mathbb{N} \rightsquigarrow \mathbb{N}$. The partiality of this function is due to the fact that a program might loop forever, without producing any result at all.

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

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

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

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

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

which is a contradiction. □

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

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

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

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

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

Exercise 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. 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 $(\mathbb{N} \rightsquigarrow \mathbb{N})$.*

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

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

□

1.6 A Cardinality Argument for Non Computability

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

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

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

Theorem 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 source file 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.

Proof

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

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. □

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?*

We can rephrase the results above in a more concrete way. Consider the following problem, which is named the *halting problem*:

> Given a program P_x and an input n for it, does the execution of P_x on n halt after a finite amount of time (hence returning some output value), or does it never halt?

We could wonder if there is an automatic way to answer such a question, i.e. a program which, given x, n, is able to decide whether P_x on n halts. If there is such a program, we would say that the problem above is *decidable*.

In more formal terms, we want to know if there is an implementation for the following function:

$$h(x,n) = \begin{cases} 1 & \text{if } \phi_x(n) \text{ is defined} \\ 0 & \text{if } \phi_x(n) \text{ is undefined} \end{cases}$$

The following result states that no such implementation exists.

Proof

Theorem 39. *The halting problem is undecidable.*

Proof. By contradiction, assume we have an implementation H for function h above. Then, we could build a program as follows:

```
function F(n) :
    if H(n, n) = 1
        then return φ_n(n) + 1;
        else return 0;
```

The above program relies on the fact that $\phi_n(n)+1$ is computable. Indeed, that is the case: intuitively, if we wrote the above code in Java, we could compute $\phi_n(n)+1$ by e.g. "saving" the number n in a file (generating the source code of P_n), invoking the Java compiler on that file, run the resulting executable with input n, taking the result of that and add 1 to it. The whole procedure is guaranteed to halt in finite time since $H(n,n) = 1$: this for instance rules out the possibility that program P_n (on n) gets stuck into an infinite loop and never return any output.

However, the above program F would then compute the function f defined in Th. 37, which we proved to be non computable. Hence, we obtain a contradiction. □

On terminology. In Computability literature, it is common to read expressions such as "computable function", "effective function", "recursive function". All of these actually mean the same thing, namely that there exists an implementation for a given function. Similarly, "computable problem", "decidable problem", "recursive problem" indicate the existence of a program which is able to return (in a finite time) whether a given property $p(n)$ is true or false when given n as input. Similarly for "decidable predicate/property", etc. Finally, "decidable set", "recursive set", etc. indicate the decidability of the membership predicate "$n \in A$".

1.7 Summary

The most important facts in this section:

- naïve set theory ; logical formulas
- encoding and decoding functions for $\mathbb{N}^2, \mathbb{N} \uplus \mathbb{N}$, as well as the usual data types
- diagonalization method for constructing a non-computable function
- the halting problem is undecidable

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. Proving 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 a single 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 technical 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 40 (λ-terms)**.** *Let* $\mathsf{Var} = \{x_0, x_1, \ldots\}$ *be a denumerable set of variables. The syntax of the* λ*-terms is*

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

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

Intuition. Roughly speaking, the λ-abstraction $\lambda x.\, N$ represents the function which takes input x and returns N. The subterm N can depend on x. For instance, instead of writing

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

we shall write

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

Application of a λ-abstraction then behaves as follows:

$$(M\ 3) = ((\lambda x.\, x^2 + 5)\ 3) = 3^2 + 5 = 14$$

This will be made precise in the following sections, when we shall define the α, β, η semantic rules.

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. 40. Later, we shall see how we can express things like 5 and x^2 in the calculus.

Exercise 41. *Rewrite the definition of Λ, providing a recursive equation of the form $\Lambda \simeq \cdots$. Use only the following constructs:* $\mathsf{Var}, \times, \uplus$.

Notation: chains of left applications. As a notational convention, we write chains of applications *associated to the left* such as

$$(((x\ y)\ z)\ w)$$

in the more compact form

$$x\, y\, z\, w$$

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.

λ-structural rules. An 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 any subterm.

Definition 42. *The λ-structural inductive rules for a relation R between λ-terms are the following:*

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

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 43. *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. We build such a bijection h by mapping function $f \in \Big[(A \times B) \to C\Big]$ to the function $h(f) \in \Big[A \to (B \to C)\Big]$ defined as:

$$\begin{array}{rl} h(f) & = g_f \in \Big[A \to (B \to C)\Big] \quad \text{where} \\ g_f(a) & = g_{f,a} \in (B \to C) \\ g_{f,a}(b) & = f(a,b) \in C \end{array}$$

Checking that h is indeed a bijection is 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.

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

For example:

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

Note that this way of expressing binary functions also allows *partial application*: we can just apply 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.

Notation: chains of abstractions. Repeated abstractions such as

$$\lambda x.\, \lambda y.\, \lambda z.\, M$$

are also written in the compact form

$$\lambda xyz.\, M$$

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

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

Terms having no free variables are said to be closed. *The set of closed λ-terms is denoted with $\Lambda^0 = \{M \mid \mathsf{free}(M) = \emptyset\}$.*

Exercise 45. *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 46. *The result of applying a substitution $M\{N/x\}$ is defined as follows.*

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

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 $\lambda x_j.\, M$, as is different from x_i. Then, we 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 47 (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. 42)*
- *rule α*
  $\lambda x.\, M =_\alpha \lambda y.\, M\{y/x\}$ *when* $y \notin \mathsf{free}(M)$

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.

Definition 48 (Up-to-α rule)**.** *The "up-to-α" inference rule for a relation* R *between λ-terms is the following.*

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

2.4 β and η Rules

Definition 49 (β rule)**.** *Here's the β rule, used to compute the result of function application.*

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

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

[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.

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

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

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 51. *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 52. *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.*

- *λ-structural rules for $\rightarrow_R$ (see Def. 42)*
- *up-to-α rule for $\rightarrow_R$ (see Def. 48)*
- *top-level rule R*
  $M \rightarrow_R N$ *when* $M \rightarrow^t_R N$

Example 53. *Here's an example which shows that in some cases it is* mandatory *to α-convert λ-bound variables.*

$$\begin{array}{l}(\lambda x.\, ((\lambda y.(\lambda x.\, y\ x))\,(x\ x)))\\ \rightarrow_\beta (\lambda x.\, (\lambda x.\, y\ x)\{(x\ x)/y\})\\ = (\lambda x.\, (\lambda \hat{x}.\, x\ x\ \hat{x}))\end{array}$$

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 $\rightarrow_\beta$ 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

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

Exercise 54. *Prove that the relations $\rightarrow_R, R \in \{\beta, \eta\}$ above are non-deterministic, i.e.*

$$M \rightarrow_R M_1 \wedge M \rightarrow_R M_2 \wedge M_1 \neq M_2$$

for some M, M_1, M_2.

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

Definition 55. *We let*

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

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

Definition 56 (Normal form)**.** *Given a reduction relation $\rightarrow_R$ (e.g. with $R = \beta$, $R = \eta$, or $R = \beta\eta$), we say that a term M is in R-normal form iff $M \not\rightarrow_R$.*

2.4.1 β Normal Forms

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

Definition 57. *A β-reduction[2] for M is a finite or infinite sequence of terms M_i such that:*

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

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

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 58. *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 59. *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 60. *Show that $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ has no halting β-reduction, hence no β normal form.*

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

Exercise 61. *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 62. *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 63. *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 64. *A relation* $\to_R$ *is a Church-Rosser relation iff* $\forall M, N_1, N_2$

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

Theorem 65 (Church-Rosser)**.** *The relation* $\to_\beta$ *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 "the β-normal form" instead of "a 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 66 (Leftmost-outermost reduction relation)**.** *The leftmost-outermost* β*-reduction relation is* $\to_\beta$ *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 the* λ*-term left-to-right. Below we show a procedure to compute the leftmost-outermost residual.*

procedure $\mathsf{L}(M)$
Input: a λ*-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

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

Exercise 67. *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 68 (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.*

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.

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 69. *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 λ-term has an infinite η-reduction.

Exercise 70. *Prove that $\to_\eta$ is Church-Rosser.*

Theorem 71 (Existence and uniqueness of η-normal form)**.** *Each given M admits exactly one η-normal form.*

Proof. The above exercises imply the statement. □

[4] When it exists.

Exercise 72. *Let M be a β-normal form, and $M \to_\eta N$. Prove that N is still a β-normal form.*

Exercise 73 (Commuting η and β)**.** *(Hard) Prove the following property. If $M \to_\eta^* N \to_\beta^* O$, then $M \to_\beta^* N' \to_\eta^* O$ for some N'.*

See Sol. 319 for some hints.

Theorem 74. *M has a β-normal form if and only if M has a $\beta\eta$-normal form.*

Proof. ($\Rightarrow$) Immediate from Ex.72 and Th. 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. 73 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 75 (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 76 (Axiomatic semantics for the untyped λ-calculus)**.** *The equivalence relation $=_{\beta\eta}$ between λ-terms is inductively defined below.*

- *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 77. *Prove that $=_{\beta\eta}$ is closed under the λ-structural rules of Def. 42.*

Exercise 78. *Use the η rule to prove the* **ext** *rule.*

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

Exercise 79. *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 80. *If $M =_{\beta\eta} N$ and N is a $\beta\eta$-normal form, then $M \to^*_{\beta\eta} N$.*

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.

λ-terms (M, Λ)

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

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

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

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

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

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

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

equivalence relation rules for R

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

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 rule for R

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

β reduction relation ($\rightarrow^t_\beta, \rightarrow_\beta$)

- $(\lambda x.\ M)N \rightarrow^t_\beta M\{N/x\}$
- λ-structural rules for $\rightarrow_\beta$
- up-to-α rule for $\rightarrow_\beta$
- top-level rule $\dfrac{M \rightarrow^t_\beta N}{M \rightarrow_\beta N}$

η reduction relation ($\rightarrow^t_\eta, \rightarrow_\eta$)

- $(\lambda x.\ M\ x) \rightarrow^t_\eta M$ if $x \notin \mathsf{free}(M)$
- λ-structural rules for $\rightarrow_\eta$
- up-to-α rule for $\rightarrow_\eta$
- top-level rule $\dfrac{M \rightarrow^t_\eta N}{M \rightarrow_\eta N}$

$\beta\eta$ reduction relation ($\rightarrow^t_{\beta\eta}, \rightarrow_{\beta\eta}$)

$M \rightarrow^t_{\beta\eta} N$ iff $M \rightarrow^t_\beta N$ or $M \rightarrow^t_\eta N$

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

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

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

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

2.5 Some Useful Combinators

Definition 81. *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 $\mathbf{I}$ *represents the identity function. The λ-term* $\mathbf{K}$ *is used to build constant functions: e.g.* $\mathbf{K}\,5$ *is a function which always returns* 5*, since* $\mathbf{K}\ 5\ x =_{\beta\eta} 5$ *for all* x*.*

Example 82. *We have the following:*

$$\mathbf{KISS} =_{\beta\eta} ((\mathbf{KI})\mathbf{S})\mathbf{S} =_{\beta\eta} \mathbf{IS} =_{\beta\eta} \mathbf{S}$$

Another example:

$$\mathbf{SKK}x =_{\beta\eta} \mathbf{K}x(\mathbf{K}x) =_{\beta\eta} x =_{\beta\eta} \mathbf{I}x$$

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

$$\mathbf{SKK} =_{\beta\eta} \mathbf{I}$$

Another example:

$$\mathbf{KI}xy =_{\beta\eta} \mathbf{I}y =_{\beta\eta} y =_{\beta\eta} \mathbf{F}xy$$

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

$$\mathbf{KI}x =_{\beta\eta} \mathbf{F}x$$

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

$$\mathbf{KI} =_{\beta\eta} \mathbf{F}$$

Exercise 83. *Prove that we do not have* $\mathbf{T} =_{\beta\eta} \mathbf{F}$*. See Sol. 320.*

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

$$\neg\forall MNO.\,(MN)O =_{\beta\eta} M(NO)$$

Proof. By contradiction,

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

□

General Hint. To prove that some equation does 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, $\mathbf{K}NO =_{\beta\eta} \mathbf{K}(NO)$. 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 $\mathbf{K}NOP =_{\beta\eta} \mathbf{K}(NO)P$, implying $NP =_{\beta\eta} NO$. Now it is easy to guess $N = \mathbf{I}$, so to obtain $P =_{\beta\eta} O$. Guessing P, O is then made trivial.

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

$$MN =_{\beta\eta} NM$$
$$M(NO) =_{\beta\eta} O(MN)$$
$$M(MO) =_{\beta\eta} MO$$
$$MO =_{\beta\eta} MOO$$
$$MM =_{\beta\eta} M$$
$$MN =_{\beta\eta} \lambda x.\,M(Nx)$$

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

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

Exercise 87. *Check that the following composition operator is associative:*

$$\circ = \lambda f g x.\, f(gx)$$

2.6 Programming in the λ-calculus

In this section we argue that we can indeed "program" in the λ-calculus. That is, that inside the λ-calculus it is possible to represent *data*, and it is possible to manipulate them through *algorithms*.

We start by representing *booleans*, and providing "programs" which perform the usual logical operations (and, or, not). We also show how to write an "if-then-else".

We proceed to represent *pairs* of arbitrary values $\langle x, y\rangle$. We provide a pair constructor (given x and y, build $\langle x, y\rangle$), as well as projections (e.g., given $\langle x, y\rangle$, extract x from it).

We then represent *natural numbers*, and implement all the common arithmetic operators $(+, *, \ldots)$, as well as the logical comparisons $(\leq, \neq, \ldots)$. We show how to perform a kind of "FOR loop", i.e. how to repeat the same operation a given number n of times.

Finally, we present a fundamental technique to define functions in a *recursive* way. This is very important, since e.g. it allows our programs to simulate "WHILE loops", i.e. to loop until an exit condition is met.

2.6.1 Representing Booleans

We pick two specific λ-terms to represent the constants "true" and "false" in the λ-calculus. These are the λ-terms $\mathbf{T}$ and $\mathbf{F}$ we introduced earlier.

$$\mathbf{T} = \lambda x y.\, x$$
$$\mathbf{F} = \lambda x y.\, y$$

If we define "if-then-else" as a simple application

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

we have that the above indeed respects the usual behaviour of if-then-else, i.e. the laws below:

$$\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ N\ \mathsf{else}\ O =_{\beta\eta} N$$
$$\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ N\ \mathsf{else}\ O =_{\beta\eta} O$$

Exercise 88. *Check the claim above.*

Exploiting the above if-then-else it is then possible to derive all the standard logical operators.

Exercise 89. *Define the logical operators* **And**, **Or**, **Not**. *(See Sol. 321)*

2.6.2 Representing Pairs

When programming, it is sometimes convenient to use pairs of values to represent data. To this aim, we need to be able to construct a pair given its two components, and then to project the first/second component out of a pair. The specification of these operations is then the following:

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

A possible implementation of these operations is then:

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

Exercise 90. *Prove that the above implementation satisfies the pair laws given in the specification.*

Exercise 91. *Define F_1, F_2 so that:*

- $F_1(\mathbf{Cons}\, x\, y) = \mathbf{Cons}\, x\, (\mathbf{Cons}\, y\, x)$
- $F_2(\mathbf{Cons}\, x\, (\mathbf{Cons}\, y\, z)) = \mathbf{Cons}\, z\, (\mathbf{Cons}\, x\, y)$

2.6.3 Representing Natural Numbers: 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 92. *The sequence of Church numerals is inductively defined as follows. Let s and z be variables*[5]*.*

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

Zero, Successor We can define a "zero" and "successor" λ-terms as follows

$$\begin{aligned} \mathbf{0} &= \lambda sz.\, z \\ \mathbf{Succ} &= \lambda nsz.\, s(n\, s\, z) \end{aligned}$$

The above definitions indeed satisfy the following.

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

Test against zero An operator for testing a numeral against zero should satisfy the following:

$$\begin{aligned} \mathbf{IsZero} \ulcorner 0 \urcorner &=_{\beta\eta} \mathbf{T} \\ \mathbf{IsZero} \ulcorner n+1 \urcorner &=_{\beta\eta} \mathbf{F} \end{aligned}$$

A possible implementation is:

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

Exercise 93. *Check that the above implementation is indeed correct.*

Predecessor We want to define a predecessor function with the following properties:

$$\begin{aligned} \mathbf{Pred} \ulcorner 0 \urcorner &=_{\beta\eta} \ulcorner 0 \urcorner \\ \mathbf{Pred} \ulcorner n+1 \urcorner &=_{\beta\eta} \ulcorner n \urcorner \end{aligned}$$

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

Note that we let, roughly speaking, $0 - 1 = 0$ above, since we do not have negative numbers in our numerals. A possible implementation of the above specification is as follows:

$$\begin{aligned}\mathbf{Pred} &= \lambda n.\, \mathbf{Snd}(n\, G\, (\mathbf{Cons}\, \mathbf{F}\, \ulcorner 0 \urcorner)) \\ G &= \lambda p.\, \mathbf{Cons}\, \mathbf{T} (\mathbf{Fst}\, p\, (\mathbf{Succ}(\mathbf{Snd}\, p))\, (\mathbf{Snd}\, p))\end{aligned}$$

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

Hint: the above λ-term repeatedly applies a function g, defined as follows:

$$g(\langle b, x\rangle) = \begin{cases} \langle \mathsf{true}, x+1\rangle & \text{if } b = \mathsf{true} \\ \langle \mathsf{true}, x\rangle & \text{if } b = \mathsf{false} \end{cases}$$

The correctness then comes from $g(g(\cdots g(\langle \mathsf{false}, 0\rangle))) = \langle b', n-1\rangle$, where b' is some boolean value.

Arithmetic Operators Addition can be implemented following this idea:

$$\ulcorner n + m \urcorner =_{\beta\eta} \mathbf{Succ}(\mathbf{Succ}(\cdots(\mathbf{Succ} \ulcorner m \urcorner)))$$

where **Succ** is applied n times. The above suggests the following program:

$$\mathbf{Add} = \lambda nm.\, n\, \mathbf{Succ}\, m$$

Exercise 95. *Prove that the above is correct.*

Subtraction is done similarly:

$$\mathbf{Sub} = \lambda nm.\, m\, \mathbf{Pred}\, n$$

Note however that since $\mathbf{Pred} \ulcorner 0 \urcorner =_{\beta\eta} \ulcorner 0 \urcorner$, we have $\mathbf{Sub} \ulcorner n \urcorner \ulcorner m \urcorner =_{\beta\eta} \ulcorner 0 \urcorner$ if and only if $n \leq m$.

Exercise 96. *Prove that the above is correct.*

Similarly, for multiplication we have:

$$\ulcorner n \cdot m \urcorner =_{\beta\eta} \mathbf{Add} \ulcorner m \urcorner (\mathbf{Add} \ulcorner m \urcorner (\cdots (\mathbf{Add} \ulcorner m \urcorner \ulcorner 0 \urcorner)))$$

where $\mathbf{Add} \ulcorner m \urcorner$ is applied n times. Hence,

$$\mathbf{Mul} = \lambda nm.\, n\, (\mathbf{Add}\, m)\, \ulcorner 0 \urcorner$$

Exercise 97. *Prove that the above is correct.*

Exercise 98. *(Tricky) Define a λ-term to implement division. (See Sol. 321).*

Logical Comparisons Above, we anticipated that the test for "less-than-or-equal" can be performed exploiting **Sub**.

$$\mathbf{Leq} = \lambda nm.\, \mathbf{IsZero}(\mathbf{Sub}\, n\, m)$$

Exercise 99. *Prove that the above is correct.*

The test for equality is then easy to derive.

Exercise 100. *Define the equality test* **Eq**. *(See Sol. 321).*

Exercise 101. *Prove that $n = m$ if and only if $\ulcorner n \urcorner =_{\beta\eta} \ulcorner m \urcorner$.*

2.6.4 Defining Functions Recursively through Fixed Points

Can we build recursive functions? For instance, consider the factorial function below. (To improve readability, here we use a liberal syntax for arithmetics, instead of the actual λ-terms we saw in then previous sections.)

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

Is there some λ-term F that satisfies the equation above (at least when using $=_{\beta\eta}$)? Of course, the equation itself has F on both sides so it does not immediately define a λ-term F, like e.g. $x = x/2 + 1$ does not immediately define x.

What if we abstract the recursive call, transforming it into a call of some arbitrary function g?

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

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 adapt 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 102. *Use the above definition of F to check* (2.1).

Exercise 103. *Use the above definition of F to compute the factorial of* 3.

Exercise 104. *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

$$f =_{\beta\eta} Ff$$

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 $\mathbf{\Theta}F =_{\beta\eta} F(\mathbf{\Theta}F)$ 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 105. *Write such a* $\mathbf{\Theta}$.
Hint. This seems hard, but we know all the tricks now. Start from the equation $\mathbf{\Theta} = \lambda F.\, F(\mathbf{\Theta}F)$*, and apply the technique shown above.*
After you solved this, compare your solution to that in Sol. 322.

Definition

Exercise 106. *Check whether these terms have a β-normal form*

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

Further Exercises

Exercise 107. *Assume lists of positive naturals such as* $[1, 2, 3]$ *are encoded as* $\mathbf{Cons}\ulcorner 1\urcorner(\mathbf{Cons}\ulcorner 2\urcorner(\mathbf{Cons}\ulcorner 3\urcorner(\mathbf{Cons}\ulcorner 0\urcorner, \Omega)))$*, using* $\ulcorner 0\urcorner$ *to mark the end of the list. Write the following functions:*

- **Length** *returning the length of a list*

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

See Sol. 325.

Exercise 108. *Find an encoding for lists of arbitrary (opaque) data, and adapt the functions seen above. What about binary trees?*

2.6.5 Computing the Standard Bijections

We previously introduced bijections between the set of natural numbers $\mathbb{N}$ and the cartesian product $\mathbb{N} \times \mathbb{N}$, as well as the disjoint union $\mathbb{N} \uplus \mathbb{N}$. We now can implement these, and their inverses, in the λ-calculus.

For the cartesian product we proceed as follows:

Exercise 109. *Construct* $\mathbf{Pair}, \mathbf{Proj1}, \mathbf{Proj2}$ *such that:*

$$\begin{aligned}
&\mathbf{Pair}\ \ulcorner n \urcorner\ \ulcorner m \urcorner =_{\beta\eta} \ulcorner \mathsf{pair}(n, m) \urcorner \\
&\mathbf{Proj1}\ \ulcorner n \urcorner =_{\beta\eta} \ulcorner \mathsf{proj1}(n) \urcorner \\
&\mathbf{Proj2}\ \ulcorner n \urcorner =_{\beta\eta} \ulcorner \mathsf{proj2}(n) \urcorner
\end{aligned}$$

Also see Sol. 321.

For the disjoint union, we instead proceed as below. Note that inverting $\mathsf{inL}/\mathsf{inR}$ actually amounts to perform a kind of if-then-else; that is, first we need to check whether a given number is of the form $\mathsf{inL}(n)$ (even) or $\mathsf{inR}(n)$ (odd), and then we have to compute n accordingly. This is what is done by **Case** below.

Exercise 110. *Construct* $\mathbf{InL}, \mathbf{InR}, \mathbf{Case}$ *such that:*

$$\begin{aligned}
&\mathbf{InL}\ \ulcorner n \urcorner =_{\beta\eta} \ulcorner \mathsf{inL}(n) \urcorner \\
&\mathbf{InR}\ \ulcorner n \urcorner =_{\beta\eta} \ulcorner \mathsf{inR}(n) \urcorner \\
&\mathbf{Case}\ \ulcorner \mathsf{inL}(n) \urcorner\ L\ R =_{\beta\eta} L\ \ulcorner n \urcorner \\
&\mathbf{Case}\ \ulcorner \mathsf{inR}(n) \urcorner\ L\ R =_{\beta\eta} R\ \ulcorner n \urcorner
\end{aligned}$$

Also see Sol. 321.

Exercise 111. *Use the functions above to construct a* G *such that*

$$\begin{aligned}
&G\ \ulcorner \mathsf{inL}(n) \urcorner =_{\beta\eta} \ulcorner \mathsf{inR}(\mathsf{inL}(n)) \urcorner \\
&G\ \ulcorner \mathsf{inR}(\mathsf{inL}(n)) \urcorner =_{\beta\eta} \ulcorner \mathsf{inR}(\mathsf{inR}(n)) \urcorner \\
&G\ \ulcorner \mathsf{inR}(\mathsf{inR}(n)) \urcorner =_{\beta\eta} \ulcorner \mathsf{inL}(n) \urcorner
\end{aligned}$$

2.6.6 Representing λ-terms

In this section, we find a way to represent the *syntax* of λ-terms in the λ-calculus itself. In this way, we can construct λ-terms which manipulate the syntax of other λ-terms.

Enumeration of λ-terms. We start by enumerating the λ-terms by constructing a bijection $\# \in (\Lambda \leftrightarrow \mathbb{N})$. This associates to any λ-term $M \in \Lambda$ a unique numeric *index* $\#M \in \mathbb{N}$, forming a one-to-one correspondence. The definition of $\#$ is given by induction over the syntax of M, and closely follows the recursive set definition $\Lambda \simeq \mathsf{Var} \uplus ((\Lambda \times \Lambda) \uplus (\mathsf{Var} \times \Lambda))$.

Definition 112. *The bijection $\# \in (\Lambda \leftrightarrow \mathbb{N})$ is defined as follows.*

$$\#M = \begin{cases} \mathsf{inL}(i) & \text{if } M = x_i \\ \mathsf{inR}(\mathsf{inL}(\mathsf{pair}(\#N, \#O))) & \text{if } M = NO \\ \mathsf{inR}(\mathsf{inR}(\mathsf{pair}(i, \#N))) & \text{if } M = \lambda x_i.\, N \end{cases}$$

Exercise 113. *Check that the function $\#$ above is indeed a bijection.*

Exercise 114. *Compute $\#(\lambda x_3.\, x_3(\lambda x_0.x_0))$. Then find the λ-term M having $\#M = 51$.*

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 115. *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 116. *Show that $\#(\lambda x_0.\, x_0) \neq \#(\lambda x_1.\, x_1)$.*

Representing programs We can then represent the natural $\#M$ in the calculus exploiting Church's numerals.

Definition 117. *The function $\ulcorner M \urcorner$ is defined as follows.*

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

Constructing programs It is possible, within the λ-calculus, to construct (representations of) λ-terms using the following programs.

Exercise 118. *Define* $\mathbf{Var}, \mathbf{App}, \mathbf{Lam}$ *such that*

$$\begin{aligned}\mathbf{Var}\ulcorner i\urcorner &=_{\beta\eta} \ulcorner x_i\urcorner\\ \mathbf{App}\ulcorner M\urcorner\ulcorner N\urcorner &=_{\beta\eta} \ulcorner MN\urcorner\\ \mathbf{Lam}\ulcorner i\urcorner\ulcorner M\urcorner &=_{\beta\eta} \ulcorner \lambda x_i.\ M\urcorner\end{aligned}$$

Also see Solution 323.

For example,

$$\mathbf{Lam}\ulcorner 3\urcorner\left(\mathbf{App}\ (\mathbf{Var}\ulcorner 3\urcorner)\ (\mathbf{Var}\ulcorner 2\urcorner)\right) =_{\beta\eta} \ulcorner \lambda x_3.\ x_3\ x_2\urcorner$$

Note that the existence of the above **App** is a (special case of a) result known as the Parameter lemma, or the *s-m-n* lemma.

Lemma 119 (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. See Solution 323. □

Destructing programs It is also possible, within the λ-calculus, to decode the representation of the λ-terms: given $\ulcorner M\urcorner$ we can detect whether M is a variable, application, or abstraction, and in any case split its syntax into the sub-components. The result is a kind of three-way if-then-else, as shown below.

Exercise 120. *Construct a "shallow decoder" for our bijection #, satisfying the following.*

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

See also Solution 324.

Exploiting constructing and destructing operators it is possible to manipulate syntax in significant ways.

Exercise 121. *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 327 in the Appendix.

Constructing representations for naturals Given (the representation of) a natural n, it is possible to construct (the representation of) its Church numeral as follows.

Lemma 122. *There exists* $\mathbf{Num} \in \Lambda^0$ *such that for all* $n \in \mathbb{N}$

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

Proof.

$$\mathbf{Num} = \lambda n.\, \mathbf{Lam}\, {\ulcorner\urcorner} 0^{\urcorner\urcorner} \left(\mathbf{Lam}\, {\ulcorner\urcorner} 1^{\urcorner\urcorner} \left(n\, (\mathbf{App}\, {\ulcorner} x_0^{\urcorner})\, {\ulcorner} x_1^{\urcorner} \right) \right)$$

Indeed, $\mathbf{Num}^{\ulcorner\urcorner} n^{\urcorner\urcorner} =_{\beta\eta} {\ulcorner} \lambda x_0.\, \lambda x_1.\, x_0(x_0(\cdots(x_0\, x_1)))^{\urcorner} = {\ulcorner\urcorner} n^{\urcorner\urcorner}{}^{\urcorner}$. □

Note that, in particular, we have that

$$\mathbf{Num}^{\ulcorner} M^{\urcorner} = \mathbf{Num}^{\ulcorner\urcorner} \#M^{\urcorner\urcorner} =_{\beta\eta} {\ulcorner\urcorner} \#M^{\urcorner\urcorner}{}^{\urcorner} = {\ulcorner} M^{\urcorner}{}^{\urcorner}$$

Destructing representations for naturals It is also possible to check whether a λ-term is a numeral, as well as recovering its associated natural value.

Exercise 123. *Define* **IsNumeral** *to check, given* $\ulcorner M \urcorner$*, whether* M *is syntactically a numeral, in any possible* α*-converted form. That is:*

$$\begin{array}{ll} \mathbf{IsNumeral}\ulcorner M\urcorner =_{\beta\eta} \mathbf{T} & \text{if } M = (\lambda x_i.\ \lambda x_j.\ x_i\ (x_i \cdots (x_i\ x_j))) \wedge i \neq j \\ \mathbf{IsNumeral}\ulcorner M\urcorner =_{\beta\eta} \mathbf{F} & \text{otherwise} \end{array}$$

Using the same technique, construct **Extract** *such that:*

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

See Sol. 330 and 331.

2.7 λ-definable Functions

In the previous sections we focused on performing several operations in the λ-calculus, e.g. all the usual arithmetic operators. We now want to define more formally when a λ-term M implements a given k-ary partial function $f \in (\mathbb{N} \rightsquigarrow \mathbb{N})$. When that happens, we say that M λ-defines f.

For *total* functions f, we clearly desire the following:

$$M\ulcorner n \urcorner =_{\beta\eta} \ulcorner f(n) \urcorner$$

Indeed, the above is what we did when we implemented total functions such as $\mathsf{inL}, \mathsf{inR}, \mathsf{pair}$. Generalizing this to the general case, in which f may be not total is unfortunately not straightforward. We want something of this form:

$$\begin{array}{c} \text{when } f(n) \text{ is defined, then } M\ulcorner n \urcorner =_{\beta\eta} \ulcorner f(n) \urcorner \\ \text{when } f(n) \text{ is not defined, then } M\ulcorner n \urcorner \ldots \text{(what to put here?)} \ldots \end{array}$$

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

Definition 124. *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.

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 125 (solvability)**.** *A closed λ-term M is solvable if there exist $N_1, \ldots, N_k$, with $k \geq 0$, such that $MN_1 \cdots N_k =_{\beta\eta} \mathbf{I}$.*

[Barendregt 8.3.1]

Exercise 126. *Show that if M is unsolvable, then MN is also unsolvable, for any N.*

Exercise 127. *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 128. *Show that Church's numerals can be* uniformly solved *by finding M, N such that $\forall n \in \mathbb{N}.\ulcorner n\urcorner MN = \mathbf{I}$.*

Theorem 129. *Any closed β-normal form is solvable.*

Proof. We leave this as an exercise.
Hint: first, show that a β-normal form must have the form

$$\lambda x_1 \dots x_n.\ x_i M_1 \dots M_k$$

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

Exercise 130. *Let M be a closed λ-term. Show that M is solvable if and only if there exist $N_1, \dots, N_k$ ($k \geq 0$) such that $MN_1 \dots N_k$ has a $\beta\eta$-normal form.*

The above exercise states that option 3 actually requires us to represent undefinedness with unsolvable terms. We can exploit this to define λ-definability for partial functions:

Definition 131 (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 $n \in \mathbb{N}$*

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

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

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. The following exercise ensures that indeed now we can compose functions avoiding the issues we found with terms such as H.

Exercise 132. *Show that if f, g are partial λ-definable functions, then their composition $f \circ g$ is such.*
Hint: exploit Ex. 128, 126. See also Sol 326.

A set $A \subseteq \mathbb{N}$ is said to be λ-definable whenever it is feasible, in the λ-calculus, to check whether any given natural $n \in \mathbb{N}$ belongs to A. In other words: A is λ-definable when the membership test "$n \in A$" is implementable in the λ-calculus.

Definition 133. *A set $A \subseteq \mathbb{N}$ is λ-defined by V_A iff* **Definition**

$$n \in A \implies V_A \ulcorner n \urcorner = \mathbf{T}$$
$$n \notin A \implies V_A \ulcorner n \urcorner = \mathbf{F}$$

Such a V_A is said to be a verifier *for A. A set A is said to be λ-definable whenever it is λ-defined by some V_A.*

Exercise 134. *Change* **T** *with* 1 *and* **F** *with* 0 *in the definition above, and prove this alternative notion of λ-definability for sets to be equivalent.*

Conclude that A is λ-definable if and only if its characteristic function χ_A is λ-definable.

Lemma 135. *λ-definable sets are closed under*

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

Proof. Suppose that two sets are λ-defined by V_A, V_B, respectively. Then, their union is λ-defined by

$$\lambda x.\ \mathbf{Or}\ (V_A\ x)\ (V_B\ x)$$

Their intersection is λ-defined by

$$\lambda x.\ \mathbf{And}\ (V_A\ x)\ (V_B\ x)$$

Their complement is λ-defined by

$$\lambda x.\ \mathbf{And}\ (V_A\ x)\ (\mathbf{Not}\ (V_B\ x))$$

□

Exercise 136. *Prove that the empty set $\emptyset$, as well as the whole set $\mathbb{N}$ are λ-definable.*

Exercise 137. *Show that finite subsets of $\mathbb{N}$ are λ-definable. See Sol 333.*

Exercise 138. *Let $A, B \subseteq \mathbb{N}$. Show that, if A is a λ-definable set, and $(A \setminus B) \cup (B \setminus A)$ is finite, then B is λ-definable.*

Lemma 139. *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 $\mathbf{F}$. □

2.8 Classical Computability Results in the λ-calculus

We can now finally state some classical computability results.

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.

Existence of a non-λ-definable function. 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

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

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

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

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

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

has a normal form — another contradiction.

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

Lemma 140. *The function f defined above is not λ-definable.*

Proof. The discussion above the statement actually proved it. □

Exercise 141. *Compare this result with Th. 37. You should find the proof to be similar.*

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

Definition 142. $\mathsf{K}_\lambda = \{\#M | M\ulcorner M\urcorner \text{ has a } \beta\text{-normal form}\}$

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

Lemma 143. K_λ *is not λ-definable*

Proof. By contradiction, if K_λ were λ-definable by e.g. G, then we could λ-define the function f of Lemma 140 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, which we proved to be non λ-definable in Lemma 140. □

2.8.1 Reduction Arguments

Suppose that, given a verifier V_A for a set A, one is able to construct a verifier $V_B = (\lambda n.\ \cdots V_A \cdots)$ for another set B. This actually establishes that:

$$A \text{ is } \lambda\text{-definable} \implies B \text{ is } \lambda\text{-definable}$$

Interestingly, the above can be equivalently stated as

$$B \text{ is not } \lambda\text{-definable} \implies A \text{ is not } \lambda\text{-definable}$$

If B is known to be non-λ-definable, the above actually proves that A is non-λ-definable.

In other words, in order to prove that a set A is not λ-definable, it is sufficient to:

- Find a set B which is known to be non-λ-definable
- Construct a verifier V_B exploiting the (hypothetical) existence of a verifier V_A

For example, the set K^0_λ below is similar to the set K_λ. As for K_λ, this set is not λ-definable: this can be proved following a reduction argument.

Definition 144. $\mathsf{K}^0_\lambda = \{\#M |\ M\ulcorner 0\urcorner \text{ has a } \beta\text{-normal form}\}$

Exercise 145. *Prove that* K^0_λ *is not* λ*-definable. (See sol. 334)*

Exercise 146. *Prove that* $\{2 \cdot n \mid n \in \mathsf{K}_\lambda\}$ *is not* λ*-definable.*

Exercise 147. *Prove that* $\{2 \cdot n \mid n \in \mathsf{K}_\lambda\} \cup \{2 \cdot n + 1 \mid n \in \mathbb{N}\}$ *is not* λ*-definable.*

Exercise 148. *Prove that* $\{2 \cdot n \mid n \in \mathsf{K}_\lambda\} \cup \{2 \cdot n \mid n \in \mathbb{N}\}$ *is* λ*-definable.*

Exercise 149. *Prove that* $\{\lfloor \frac{100}{n^2+1} \rfloor \mid n \in \mathsf{K}_\lambda\}$ *is* λ*-definable.*

Exercise 150. *Prove that* $\overline{\mathsf{K}_\lambda} = \mathbb{N} \setminus \mathsf{K}_\lambda$ *is not* λ*-definable.*

Exercise 151. *Prove that function* $f(n) = \chi_{\mathsf{K}_\lambda}(n) + n^2$ *is not* λ*-definable.*

Exercise 152. *Prove that function* $f(n) = \chi_{\mathsf{K}_\lambda}(\lfloor \frac{n}{42} \rfloor)$ *is not* λ*-definable.*

Exercise 153. *Prove that function* $f(n) = \chi_{\mathsf{K}_\lambda}(\lfloor \frac{42}{n} \rfloor)$ *is* λ*-definable.*

Exercise 154. *Prove that*

$$f(n) = \begin{cases} \#\mathbf{I} & \text{if } n \in \mathsf{K}_\lambda \\ \#(\mathbf{I}\ \mathbf{I}) & \text{otherwise} \end{cases}$$

is not λ*-definable. Then prove that* $(f \circ f)$ *is instead* λ*-definable.*

2.8.2 Padding Lemma

Intuitively, many different programs actually have the same semantics. Indeed, recall Ex. 115. We can actually automatically generate an infinite number of equivalent programs.

Lemma 155 (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 335. □

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 $n \in \mathbb{N}$.

2.8.3 The "Denotational" Interpreter: a Universal Program

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

Lemma 156 (Self-interpreter)**.** *There exists* $\mathbf{E} \in \Lambda^0$ *such that*

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

for all closed M*. More precisely, for any* M *we have:*

$$\mathbf{E} \ulcorner M \urcorner =_{\beta\eta} M\{\Omega/\mathsf{free}(M)\}$$

where in the right hand side all the free variables of M *have been substituted by* Ω*.*

Proof. We proceed by defining two auxiliary operators.

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

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

  These equations are satisfied by

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

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

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

These equations are satisfied by:

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

After defining $\mathbf{E}'$, we can just let $\mathbf{E} = \lambda m.\, \mathbf{E}'\, m\, (\mathbf{K}\ \Omega)$. Here we use $\mathbf{K}\Omega$ as the initial environment, so that all the free variables of the input M are mapped to Ω. Note that, when $M \in \Lambda^0$, the λ-term M has no free variables, so the initial environment will never be used by $\mathbf{E}'$. That is, we only invoke the environment ρ on variables that have been defined through **Upd**. □

Exercise 157. *Check the correctness of* $\mathbf{E}$ *in some concrete (small) cases. For instance check that* $\mathbf{E}\ulcorner\mathbf{I}\urcorner = \mathbf{I}$ *and* $\mathbf{E}\ulcorner\mathbf{K}\urcorner = \mathbf{K}$.

2.8.4 The "Operational" Interpreter: a Step-by-step Interpreter

Here we build a more "traditional" interpreter, i.e. another version of $\mathbf{E}$. 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 a given number k of steps, and tell us whether M reached normal form within that time constraint k. This will be exploited in the next chapters.

Exercise 158. *Define* **Subst** *such that*

$$\mathbf{Subst}\ulcorner i\urcorner\ulcorner M\urcorner\ulcorner N\urcorner =_{\beta\eta} \ulcorner N\{M/x_i\}\urcorner$$

Watch out for the needed α*-conversions. See Sol. 328.*

Exercise 159. *Define* **Beta** *so that it performs exactly one step of* β*-reduction in a leftmost-outermost fashion (recall Def. 66). Make it be a no-op for normal forms. Formally:*

$$\begin{array}{ll} \mathbf{Beta}\ulcorner M\urcorner =_{\beta\eta} \ulcorner M'\urcorner & \text{where } M \xrightarrow{LO}_{\beta} M' \\ \mathbf{Beta}\ulcorner M\urcorner =_{\beta\eta} \ulcorner M\urcorner & \text{where } M \not\to_{\beta} \end{array}$$

See also Sol. 329.

Exercise 160. *Define* **Eta** *to apply* $\to_\eta$ *until* η*-normal form is reached.*

$$\mathbf{Eta}\ulcorner M\urcorner =_{\beta\eta} \ulcorner M'\urcorner \quad \text{where } M \to_\eta^* M' \not\to_\eta$$

Note that this requires many steps of η*, in general.*

Exercise 161. *Define* **IsNF** *to check, given* $\ulcorner M\urcorner$*, whether* M *is in* $\beta\eta$*-normal form. Formally:*

$$\begin{array}{ll} \mathbf{IsNF}\ulcorner M\urcorner =_{\beta\eta} \mathbf{T} & \text{if } M \not\to_{\beta\eta} \\ \mathbf{IsNF}\ulcorner M\urcorner =_{\beta\eta} \mathbf{F} & \text{otherwise} \end{array}$$

Exercise 162. *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 **Θ** operator, which implements recursive calls. While **Θ** 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 163. *The size of* M*, written* $|M|$ *is defined as*

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

Exercise 164. *Show that* $\#M + 1 \geq |M|$*, for all* M*.*

So, all the function seen above can be rewritten, roughly, replacing **Θ** 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 $\mathbf{LimFix}\, F \ulcorner 3 \urcorner \Omega =_{\beta\eta} F(F(F\Omega))$. By comparison, $\mathbf{\Theta} F$ would generate an unbounded number of F's.

Exercise 165. *(Long) Write* **Subst** *using* **LimFix** *instead of* **Θ***. 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 166. *Construct another version of* **E** *using the results above (see Lemma 156). Name this variant* **Eval***. Its specification is the following:*

$\mathbf{Eval} \ulcorner M \urcorner =_{\beta\eta} \ulcorner r \urcorner$	*if* $M =_{\beta\eta} \ulcorner r \urcorner$
$\mathbf{Eval} \ulcorner M \urcorner$ *is unsolvable*	*if* M *has no numeral as* $\beta\eta$*-normal form*

Note that, unlike **E***, the above works only when* M *evaluates to a numeral.*

See also Sol. 332.

Note in passing that the above function intuitively can *not* be constructed using **LimFix**, since we do not know in advance how many steps of **Beta** we need to reach the result. Indeed, we really need something like **Θ** here. This idea will be made clearer in Chapter 3.

Exercise 167. *Construct* **Eval1** *as follows:*

$\mathbf{Eval1}\ulcorner M\urcorner\ulcorner n\urcorner =_{\beta\eta} \ulcorner r\urcorner$	*if* $M\ulcorner n\urcorner =_{\beta\eta} \ulcorner r\urcorner$
$\mathbf{Eval1}\ulcorner M\urcorner\ulcorner n\urcorner$ *is unsolvable*	*if* $M\ulcorner n\urcorner$ *has no numeral as* $\beta\eta$*-normal form*

Hint: reuse **Eval** *accordingly.*

Exercise 168. *It can be often useful to consider only the* λ*-terms that produce numerals. To this aim define a* **Term** *operator such that*

$\mathbf{Term}\ulcorner M\urcorner = \mathbf{I}$	*if* $M =_{\beta\eta} \ulcorner n\urcorner$ *for some* n
$\mathbf{Term}\ulcorner M\urcorner$ *is unsolvable*	*otherwise*

You might want to start from:

$\mathbf{TermIn}\ulcorner k\urcorner\ulcorner M\urcorner = \mathbf{T}$	*if* $M \xrightarrow{LO}{}^{*}_{\beta} N \to^{*}_{\eta} \ulcorner n\urcorner$ *for some* n *and* N *using at most* k β*-steps*
$\mathbf{TermIn}\ulcorner k\urcorner\ulcorner M\urcorner = \mathbf{F}$	*otherwise*

which can be constructed by adapting **Eval** *accordingly. (See Sol. 332 for that.)*

2.8.5 Second Recursion Theorem

We previously met fixed points as a way to model recursive programs. In particular, we saw that constructing a recursively defined program amounts to solve an equation of the form $X = F\ X$ in which X is the unknown, and F models the actual body of the recursive program. This allows one, loosely speaking, to construct a program X such that the behaviour of X is recursively defined in terms of the behaviour of X.

We now consider another form of recursive definition, modeled instead by the equation $X = F\ulcorner X\urcorner$. Here, the behaviour of X is recursively defined in terms of the *source code* of X. That is, program X is not just recursively invoking itself, but it is actually aware of its own source code. For instance, X could scan its own code and count the number of applications which occur there.

Recall that, for any F, we are always able to construct a solution for $X = F\ X$. The same technique, slightly adapted, is also able to construct a solution for $X = F\ulcorner X\urcorner$. This result is known as the *second recursion theorem*.

Theorem 169 (second recursion theorem)**.** *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 the fixed point combinator $\mathbf{Y}$). We adapt this to obtain:

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

Hence,

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

□

Note the difference between Th. 169 and Lemma 156. 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 170. *Show whether it is possible to construct a program $P \in \Lambda^0$ such that... (each point below is a standalone exercise)*

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

- $P^{\ulcorner} n^{\urcorner} = \ulcorner P(P^{\ulcorner} n^{\urcorner}) \urcorner$
- $\#P = \#P + 1$
- $\#P = \#(P^{\ulcorner} P^{\urcorner})$
- $\#P = \#\mathbf{K}$

Exercise 171. *Show that there exists a $G \in \Lambda^0$ such that for all $F \in \Lambda^0$*

$$F^{\ulcorner} G^{\ulcorner} F^{\urcorner\urcorner} =_{\beta\eta} G^{\ulcorner} F^{\urcorner}$$

2.8.6 Rice's Theorem

This is one of the most important results in computability, since it shows that a large class of interesting problems are not λ-definable.

Definition

Definition 172. *A set $A \subseteq \mathbb{N}$ is closed under $\beta\eta$ iff $\forall M, N$*

$$\#M \in A \ \wedge \ M =_{\beta\eta} N \implies \#N \in A$$

Example 173. *The set $A = \{\#M \mid M =_{\beta\eta} \mathbf{I}\}$ is closed under $\beta\eta$. This is because if we change M into an equivalent program N the property $M =_{\beta\eta} I$ is preserved:*

$$\#M \in A \wedge M =_{\beta\eta} N \implies M =_{\beta\eta} \mathbf{I} \wedge M =_{\beta\eta} N \implies N =_{\beta\eta} \mathbf{I} \implies \#N \in A$$

Similarly, the set $B = \{\#M \mid M^{\ulcorner} 4^{\urcorner} =_{\beta\eta} {}^{\ulcorner} 7^{\urcorner}\}$ is closed under $\beta\eta$. Instead, the set $C = \{\#M \mid M^{\ulcorner} M^{\urcorner} =_{\beta\eta} \mathbf{I}\}$ is not *closed under $\beta\eta$ (see the exercise below). Intuitively, from $M^{\ulcorner} M^{\urcorner} =_{\beta\eta} \mathbf{I}$ and $M =_{\beta\eta} N$ we can get $N^{\ulcorner} M^{\urcorner} =_{\beta\eta} \mathbf{I}$ but not necessarily $N^{\ulcorner} N^{\urcorner} =_{\beta\eta} \mathbf{I}$.*

Exercise 174. *(Non trivial) Prove that the above C is not closed under $\beta\eta$.*

Informally speaking, sets closed under $\beta\eta$ are those sets which involve a *semantic* property of programs, i.e. they contain all the indexes of programs having a certain kind of behaviour. This is in contrast with sets involving *syntactic* properties, such as the above C which mentions $\ulcorner M \urcorner$. As a thumb rule, if the definition of a set applies operations to the index of a program M (e.g. it involves $\#M + 1$, $2 \cdot \#M$, or $\ulcorner M \urcorner$) then the set is *likely* to be not closed under $\beta\eta$, since it is referring to the actual syntax of the program and not just to its behaviour[6].

[6]Note however that $\{\#M | \mathbf{E}\, \ulcorner M \urcorner =_{\beta\eta} \mathbf{I}\}$ *is* closed under $\beta\eta$: the property involves the syntax, but only the behaviour of M matters.

Exercise 175. *Prove that the following are equivalent, for any $A \subseteq \mathbb{N}$:*

- *A is closed under $\beta\eta$*
- *for some $B \subseteq \mathbb{N}$ we have $A = \{\#M \mid \exists N.\ M =_{\beta\eta} N \wedge \#N \in B\}$*

We can now state the main theorem:

Theorem 176 (Rice's theorem)**.** *Let $A \subseteq \mathbb{N}$. If*

1. *A is closed under $\beta\eta$*
2. *$A \neq \emptyset$*
3. *$A \neq \mathbb{N}$*

Then, A is not *λ-definable.*

Proof

Proof. By contradiction, assume hypotheses $1, 2, 3$ and that A is λ-defined by some V_A. Since $A \neq \emptyset$ and $\#$ is a bijection, we have $\#M_1 \in A$ for some λ-term M_1. Similarly, since $A \neq \mathbb{N}$ and $\#$ is a bijection, we have $\#M_0 \notin A$ for some λ-term M_0. Then, by Kleene's fixpoint theorem, there is a G such that

$$G =_{\beta\eta} (\lambda g.\ V_A\ g\ M_0\ M_1)\ \ulcorner G \urcorner =_{\beta\eta} V_A\ \ulcorner G \urcorner\ M_0\ M_1$$

Clearly, we have $\#G \in A$ or $\#G \notin A$. We now consider both cases:

- If $\#G \in A$, we have $V_A \ulcorner G \urcorner =_{\beta\eta} \mathbf{T}$, hence

  $$G =_{\beta\eta} V_A\ \ulcorner G \urcorner\ M_0\ M_1 =_{\beta\eta} \mathbf{T}\ M_0\ M_1 =_{\beta\eta} M_0$$

  Since A is closed under $\beta\eta$, from $\#G \in A$ and the above we get $\#M_0 \in A$. This contradicts the previous $\#M_0 \notin A$.

- If $\#G \notin A$, we have $V_A \ulcorner G \urcorner =_{\beta\eta} \mathbf{F}$, hence

  $$G =_{\beta\eta} V_A\ \ulcorner G \urcorner\ M_0\ M_1 =_{\beta\eta} \mathbf{F}\ M_0\ M_1 =_{\beta\eta} M_1$$

  Since A is closed under $\beta\eta$, from $\#M_1 \in A$ and the above we get $\#G \in A$. This contradicts $\#G \notin A$.

Since in each case, we reach a contradiction, we have to conclude that a verifier V_A can not exist. □

[see also Barendregt 6.5.9 to 6.6]

Nota Bene: If a set A is not closed under $\beta\eta$, Rice provides no guarantees about A being λ-definable or not.

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 177. *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* $n\}$
- $\{2 \cdot \#M + 1 | M^{\ulcorner}0^{\urcorner} =_{\beta\eta} \mathbf{I}\}$
- $\{f(\#M) | M^{\ulcorner}0^{\urcorner} =_{\beta\eta} \mathbf{I}\}$ *where* $f(n) = 3$ *if* n *is even;* $f(n) = 2$ *o.w.*
- $\{2 \cdot \#M + 1 | M^{\ulcorner}M^{\urcorner} =_{\beta\eta} \mathbf{I}\}$

2.9 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

So far, our investigation focused much on the λ-calculus. Indeed, we studied the set of functions (and sets) which are λ-definable, providing some results and techniques for establishing λ-definability.

Henceforth, we shall gradually depart from the λ-calculus. We will generalize our results in a more abstract setting which is not defined in terms of the λ-calculus, or any other programming language. There, we shall no longer speak about functions which are "computable in the λ-calculus"; we will rather talk about "computable" functions, simply.

In order to do that, in this chapter we provide an alternative, equivalent definition for "f is a λ-definable partial function". The main point in doing this is that this alternative characterization *only involves functions*. That is, we can define the set of computable functions without referring to a specific programming language.

3.1 Primitive Recursive Functions

Lemma 178. *The function* $\mathsf{zero}(n) = 0$ *is* λ*-definable.* **Proof**

Proof. Take $\mathbf{K}^{\ulcorner}0^{\urcorner}$. □

Lemma 179. *The function* $\mathsf{succ}(n) = n + 1$ *is* λ*-definable.* **Proof**

Proof. Take **Succ**. □

Lemma 180. *The projection functions* $\pi_i^k(n_1, \ldots, n_k) = n_i$ *with* $1 \leq i \leq k$ *are* λ*-definable.* **Proof**

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

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

Statement

Lemma 181. *The λ-definable (partial) functions are closed under composition.*

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. 128 and Ex 132. 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 $G\ulcorner n \urcorner$ is unsolvable, so J is also unsolvable by Ex. 126, so $M\ulcorner n \urcorner$ is also unsolvable by the same Exercise.
  - If $g(n) = m \in \mathbb{N}$ but $f(m)$ is undefined, then $J = \mathbf{I}$, $G\ulcorner n \urcorner = \ulcorner m \urcorner$, and $F\ulcorner m \urcorner$ is unsolvable. So, $M\ulcorner n \urcorner = J(F\ulcorner m \urcorner) = F\ulcorner m \urcorner$ is unsolvable as well.

□

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

Statement

Lemma 182. *The λ-definable partial functions are closed under general composition. That is, if $f \in (\mathbb{N}^k \rightsquigarrow \mathbb{N})$ and $g_1, \ldots, g_k \in (\mathbb{N}^j \rightsquigarrow \mathbb{N})$ are 'lambda-definable partial functions, , then the partial 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 181. □

Lemma 183. *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:*

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

Proof. Let G, H be the λ-terms defining g, h. Then f is λ-defined by

$$F = \lambda n n_1 \cdots n_k . J\, \mathbf{Snd} \left(n\ A\ (\mathbf{Cons}\, \ulcorner 0 \urcorner\, (G\, n_1 \cdots n_k)) \right)$$
$$A = \lambda c.\, J'\, \mathbf{Cons}\, (\mathbf{Succ}\, (\mathbf{Fst}\, c))\ (H\, (\mathbf{Fst}\, c)\ n_1 \cdots n_k\ (\mathbf{Snd}\, c))$$

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

$$J = G\, n_1 \cdots n_k\, (\mathbf{K}\, \mathbf{I})\, \mathbf{I}$$
$$J' = H\, (\mathbf{Fst}\, c)\, n_1 \cdots n_k\, (\mathbf{Snd}\, c)\, (\mathbf{K}\, \mathbf{I})\, \mathbf{I}$$

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 above results actually prove that the λ-calculus is able to implement the so-called *primitive recursive functions*, defined below. This class of functions plays an important role in Computability theory.

Definition 184. *The set of the* primitive recursive *functions $\mathcal{PR}$ is defined as the smallest set of (total) functions in $\mathbb{N}^k \to \mathbb{N}$ which:*

- *includes the constant zero function* zero*, the successor function* succ*, and the projections ("the initial functions") π_i^k; and*
- *is closed under general composition; (see Lemma 182 for the definition); and*
- *is closed under primitive recursion (see Lemma 183 for the definition).*

Lemma 185. *If $f \in \mathcal{PR}$, then f is a* total *function.*

Proof. Direct from the definition of $\mathcal{PR}$. □

Lemma 186. *Each $f \in \mathcal{PR}$ is λ-definable.*

Proof. Direct from the previous lemmata. □

Exercise 187. *Show that the following functions are in $\mathcal{PR}$.*

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

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

- *boolean operators* $\mathsf{and}, \mathsf{or}, \mathsf{not}$ *(let* 0 *denote "false", and the rest denote "true")*
- *the addition,subtraction (return e.g.* 0 *when negative), multiplication,division (return e.g.* 0 *when impossible):* $\mathsf{add}, \mathsf{mul}, \mathsf{sub}, \mathsf{div}$
- *the less-than-or-equal comparison:* $\mathsf{leq}(x, x+k) = 1$*, and* 0 *otherwise*
- *the equality comparison:* $\mathsf{eq}(x, x) = 1$*, and* 0 *otherwise*
- *the factorial function*
- *the* $\mathsf{pair}, \mathsf{inL}, \mathsf{inR}$ *functions for pairs and disjoint union (easy), as well as their inverses (not so easy).*

Exercise 188. *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 further compare $\mathcal{PR}$ to the set of λ-definable functions. We know that 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. 184. As we did for the λ-calculus we can enumerate this $\mathcal{PR}$ language using the $\mathsf{pair}, \mathsf{inL}, \mathsf{inR}$ 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 be done as we did for $\mathbf{E}$.

Exercise 189. *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.

Theorem 190. *The set of λ-definable functions is strictly larger than $\mathcal{PR}$ functions.*

Statement

Proof. See the discussion above. □

3.1.1 Ackermann's Function

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

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

[also see Cutland page 46]

Exercise 191. *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, which is non trivial). 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 192. *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 193. *Let* $f(x, y)$ *be a total* λ*-definable function. Show that*

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

is a total λ*-definable function. By* $\mu y < z.\big(f(x, y) = 0\big)$ *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 194. *Let* $f(x, y)$ *be in* $\mathcal{PR}$*. Show that*

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

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. 166. 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

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

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 195. *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 196. *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 197. *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:* **Definition**

- *includes the constant zero function* zero, *the successor function* succ, *and the projections ("the initial functions") π^k_i; and*
- *is closed under general composition (see Lemma 182 for the definition); and*
- *is closed under primitive recursion (see Lemma 183 for the definition); and*
- *is closed under* unbounded minimalisation *(defined below).*

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

- $f = \emptyset$ *(the always-undefined function)*
- $f(x) = 524$ *(constant function)*
- $f(0) = 1$ *and* $f(n+1) =$ *undefined*
- $f(2 \cdot n) = 1$ *and* $f(2 \cdot n + 1) =$ *undefined*
- $\mathsf{ack}(x,y)$ *(hard)*
  Hint: one way to do it is by implementing a stack using pair.

Lemma 199. *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

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

□

Lemma 200. *The set of recursive functions $\mathcal{R}$ is included in the set of λ-definable functions.*

Proof. Immediate by all the lemmata above. □

Exercise 201. *Show that $g(x) = (\mu y.\, f(x,y) = 1)$ is recursive when f is such. (Hint: use* eq *and negation.)*

Theorem 202. *The set of λ-definable functions coincides with 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 187, $\mathsf{proj1}, \mathsf{proj2} \in \mathcal{PR}$, so by Lemma 195, and Exercise 196, we can define the following functions in $\mathcal{R}$:

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

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

- If $f(y)$ is undefined, then $F \ulcorner y \urcorner$ has no normal form. So, no matter what $\mathsf{proj2}(n)$ evaluates to, the function 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. zero), 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 $F \ulcorner y \urcorner$ has as its normal form the numeral $\ulcorner z \urcorner$. Define k as the number of leftmost $\to_\beta$ steps needed to reach normal form. Therefore, $\mathsf{eta}(\mathsf{steps}(\#(F \ulcorner y \urcorner), k))$ will completely evaluate $F \ulcorner y \urcorner$ until $\beta\eta$ normal form, producing the index of a λ-term M, which is an α-conversion[1] of $\ulcorner z \urcorner$. The minimalisation operator μn will try each $n \in \mathbb{N}$, from 0 upwards.

  - When $0 \leq n < \mathsf{pair}(\#M, k)$, we show that $\mathsf{and}(A, B)$ returns "false" (zero), so that the minimalisation will try the next n. By contradiction, assume that $\mathsf{and}(A, B)$ returns "true". This means that A and B are both "true". Since A is "true", $\mathsf{eta}(\mathsf{steps}(\#(F \ulcorner y \urcorner), \mathsf{proj2}(n)))$

[1] Recall Exercise 116. 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 $\ulcorner z \urcorner$.

is the index i of a numeral, hence the index of a normal form of $F^{\ulcorner}y^{\urcorner}$. Since we need to do k steps to reach normal form, we have[2] $\mathsf{proj2}(n) \geq k$. This implies that $i = \#M$. Since B is "true", $\mathsf{proj1}(n) = i = \#M$. Hence $n = \mathsf{pair}(\mathsf{proj1}(n), \mathsf{proj2}(n)) = \mathsf{pair}(\#M, \mathsf{proj2}(n)) \geq \mathsf{pair}(\#M, k)$, 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 $\mathsf{eta}(\mathsf{steps}(\#(F^{\ulcorner}y^{\urcorner}), k))$ is a numeral (so A is "true"[3]), and indeed $\mathsf{eta}(\mathsf{steps}(\#(F^{\ulcorner}y^{\urcorner}), k)) = \#M = \mathsf{proj1}(n)$ (so B is "true").

So, the result of the whole $\mu n. \cdots$ expression is $\mathsf{pair}(\#M, k)$. 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 $M =_\alpha {}^{\ulcorner}z^{\urcorner}$. We conclude that, when $f(y) = z$, we have $\mathsf{f}(y) = z$.

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

Exercise 203. *Provide an alternative proof for Th. 202, 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.*

Exercise 204. *Exploiting the functions seen above, prove that the following*

[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.

are total recursive functions:

$$\mathsf{termAt}(\#M, x, k) = \begin{cases} 1 & \textit{if } M^{\ulcorner}x^{\urcorner} \textit{ reaches some } \beta\textit{-normal form } N \textit{ in} \\ & \quad \textit{exactly } k \textit{ } \beta\textit{-steps, with } N =_\eta {}^{\ulcorner}y^{\urcorner} \textit{ for some } y \in \mathbb{N} \\ 0 & \textit{otherwise} \end{cases}$$

$$\mathsf{termIn}(\#M, x, k) = \begin{cases} 1 & \textit{if } M^{\ulcorner}x^{\urcorner} \textit{ reaches some } \beta\textit{-normal form } N \textit{ in} \\ & \quad \textit{at most } k \textit{ } \beta\textit{-steps, with } N =_\eta {}^{\ulcorner}y^{\urcorner} \textit{ for some } y \in \mathbb{N} \\ 0 & \textit{otherwise} \end{cases}$$

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

$$f(x) = \mathsf{U}(\mu n.\mathsf{T}(i, x, n) = 0)$$

for some suitable natural i *(which depends on* f*).*

Proof. We have already proved this when we proved Theorem 202. 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 natural number 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 200. □

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

Statement

Theorem 208. *The set of FOR-definable functions is exactly $\mathcal{PR}$.*

Proof. Left as a (rather long) exercise. You basically have to 1) simulate all the constructs of $\mathcal{PR}$ using the FOR-commands, and 2) simulate all FOR-commands using the $\mathcal{PR}$-constructs. This can be done by exploiting the 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 209. *Define the semantics of WHILE programs.*

Definition 210. *A function f is WHILE-definable if there is some WHILE-program that has semantics f.*

Theorem 211. *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. 202.
($\supseteq$): Let $f \in \mathcal{R}$. We must find a WHILE program defining f. Take T,U as in Th. 205. By Th. 208, T and U are FOR-definable, hence WHILE-definable. Following again Th. 205, 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. □

Theorem 212. *Every WHILE-definable function can be WHILE-defined by a program having a single* `while` *loop.*

Proof. Direct consequence of Th. 205. □

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, ... $\}$-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 ⟨insert your favourite programming language here⟩).

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. It should *only* describe a document. It allows for general recursion, so it could take a long time just to output one page. It can also loop, and fail to terminate, while requiring more and more memory. PostScript can also 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 (deterministic and non-deterministic ones)

- "Conventional" imperative programming languages: C, C++, Pascal, Basic (and dialects), Java, C#, Perl, Python, Ruby, PHP, Fortran, Algol, Cobol, ...

- "Conventional" functional programming languages: Lisp (and dialects), ML (and dialects such as Caml, Ocaml, F#), Haskell, ...
- Some languages based on other paradigms: prolog, π-calculus
- Type 0 grammars
- The language in which the configuration file `sendmail.cf` of Sendmail is written.
- Conway's game of Life

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 this chapter we present some classical Computability results. We will mainly focus on recursive functions here, without referring to the λ-calculus except in rare cases. In this way, the classical results we will establish are not merely facts which hold in the world of the λ-calculus, but are instead general results about *any programming language*, or any formalism which is able to describe what is a "computation".

More precisely, we depend on the λ-calculus for the following results:

- the existence of a T, U-standard form (seen in Th. 205)
- the s-m-n theorem (to be stated — Th. 218)
- the padding lemma (to be stated — Lemma 217)

Therefore, should we wish to apply the other classical results to another programming language, e.g. Java, we would merely need to re-establish the results above.

Enumerating the recursive functions The T, U-standard form provides a means to explicitly enumerate the recursive functions:

$$\begin{aligned}\mathcal{R} = & \{\mathsf{U}(\mu n.\mathsf{T}(i,x,n)=0) \mid i \in \mathbb{N}\} \\ = & \{\mathsf{U}(\mu n.\mathsf{T}(0,x,n)=0) \\ & , \mathsf{U}(\mu n.\mathsf{T}(1,x,n)=0) \\ & , \mathsf{U}(\mu n.\mathsf{T}(2,x,n)=0) \\ & \dots \\ & \}\end{aligned}$$

We will use the following notation for this:

Definition 213. *Let i be a natural. We then define the partial function ϕ_i as $\phi_i(x) = \mathsf{U}(\mu n.\mathsf{T}(i,x,n) = 0)$. This definition is also extended to k-ary partial functions $\phi_i(x_1,\ldots,x_k)$.*

Definition

Proof

Lemma 214. $\mathcal{R} = \{\phi_i \mid i \in \mathbb{N}\}$

Proof. Immediate from the definition of ϕ and Th. 205. □

Note that, since we used an interpreter of the λ-calculus to define our functions T,U, indexes i actually correspond to λ-terms and ϕ_i can be concretely defined as follows:

Lemma 215. *Let M be the program corresponding to i, i.e. $\#M = i$. Then:*

Statement

$$\phi_i(x) = \begin{cases} y & \text{if } M\ulcorner x\urcorner =_{\beta\eta} \ulcorner y\urcorner \\ undefined & \text{if } M\ulcorner x\urcorner \text{ has no numeral } \beta\eta\text{-normal form} \end{cases}$$

Proof. Left as an exercise. □

In general, we shall not rely on the above lemma, so to account for the possibility that recursive functions ϕ_i are enumerated differently, e.g. following Java programs.

4.1 Universal Function Theorem

There is a recursive function which is "universal": this is a binary function $\mathsf{eval1}$ such that any recursive function $f(x)$ can be written as $\mathsf{eval1}(i,x)$ for some i. So, in a sense, the universal function is the "most general" recursive function. Pragmatically, this universal function represents the behaviour of an interpreter for a programming language, where the natural i encodes a program which implements function f.

Theorem 216 (Universal Function)**.**
The partial function $\mathsf{eval1}(i,x) = \phi_i(x)$ is recursive.
This can be generalized to n-ary partial functions as well.

Proof

Proof. This is a direct consequence of the T,U-standard form. Indeed,

$$\mathsf{eval1}(i,x) = \mathsf{U}(\mu n.\ \mathsf{T}(i,x,n) = 0)$$

hence $\mathsf{eval1}$ is a composition of recursive functions, hence it is computable. □

Also see [Cutland]

4.2 Padding Lemma

Lemma 217 (Padding Lemma)**.**
There is a total injective function $\mathsf{pad} \in \mathcal{R}$ *such that* **Statement**

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

Proof. Immediate from the Padding Lemma for the λ-calculus. Indeed, it is enough to implement the function

$$\mathsf{pad}(\#M) = \#(\mathbf{I}\ M)$$

which is clearly recursive (it is λ-defined by $\mathbf{App}\ \ulcorner I \urcorner$) and also injective, since $\#M \neq \#N \implies \#(\mathbf{I}\,M) \neq \#(\mathbf{I}\,N)$. The fact that $\mathsf{pad}(n) > n$ derives directly from the definition of $\#$, while $\phi_{\#M} = \phi_{\#(\mathbf{I}\,M)}$ derives from M and $\mathbf{I}\,M$ having the same behaviour. □

Also see [Cutland]

The above proof exploits the λ-calculus, so it would need to be adapted if we would like to work with other programming languages. For instance, in Java one could instead perform padding by adding dummy statements ("`x = x + 0;`") instead of adding an extra $\mathbf{I}$ as we do above.

An immediate consequence of the Padding Lemma is that if a function f is recursive, then there is an infinite number of indexes i such that $\phi_i = f$. Indeed, given just one such i, we can construct an infinite strictly-increasing sequence $\mathsf{pad}(i), \mathsf{pad}(\mathsf{pad}(i)), \ldots$ of alternative indexes for f. Further, note that this sequence can be generated by a program, since pad is recursive. Finally, note that the generated sequence is *not* composed of all the possible indexes of function f, but merely lists an infinite number of them. This can be intuitively seen from the fact that from $\#(\lambda x.\ulcorner 0 \urcorner)$ we can not reach $\#(\lambda x.\ \mathbf{Mul}\ x\ \ulcorner 0 \urcorner)$ through repeated padding, yet both are indexes for the same function zero.

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

The Parameter Theorem, or s-m-n Theorem, states that it is possible to mechanically transform an index i of a $(n{+}m)$-ary function $f(x_1, \ldots, x_m, y_1, \ldots, y_n)$ into an index j of the n-ary function $g(y_1, \ldots, y_n) = f(k_1, \ldots, k_m, y_1, \ldots, y_n)$ where k_i are given constants. Technically, it is stated as follows:

Theorem 218 (Parameter Theorem, *s-m-n* Theorem)**.**
For all naturals $m, n > 0$ there exists a total recursive injective function $\mathsf{s}_n^m(i, x_1, \ldots, x_m)$ such that for all $i, x_1, \ldots, x_m, y_1, \ldots, y_n$ we have **Statement**

$$\phi_i(x_1, \ldots, x_m, y_1, \ldots, y_n) = \phi_{\mathsf{s}_n^m(i,x_1,\ldots,x_m)}(y_1, \ldots, y_n)$$

Proof. Easy adaptation of the Parameter Lemma for the λ-calculus. Indeed, it is enough to implement the function

$$\mathsf{s}_n^m(\#M, x_1, \ldots, x_m) = \#(M \ulcorner x_1 \urcorner \ldots \ulcorner x_m \urcorner)$$

which is clearly recursive and injective. □

Also see [Cutland]

Again, we exploit the λ-calculus in above proof, so it would need to be adapted in the case we would like to work with another language. For instance, in Java, function s could work as follows. Suppose we are given an index i of the following program:

```
int f(int x, int y) { ...⟨ some code here ⟩ ...}
```

We can then compute, say, $\mathsf{s}_1^1(i, 42)$ by syntactically changing the above code, so to return an index of the following:

```
int f(int y) { int x=42; ...⟨ some code here ⟩ ...}
```

The above indeed behaves as wanted.

Exercise 219. *Show that* pad *and* s_n^m *actually are primitive recursive functions.*

Typical application. The typical application of the s-m-n theorem is, broadly speaking, to construct computable functions *returning indexes of programs* without having to explicitly manipulate the program syntax.

To better explain this, we use an example in the λ-calculus. Consider a λ-term H which syntactically manipulates λ-terms according to the following specification. (For the sake of simplicity, we assume that F is closed)

$$H \ulcorner F \urcorner =_{\beta\eta} \ulcorner \lambda x_0.\ \mathbf{Mul}\ (F\ x_0)\ x_0 \urcorner$$

This can be implemented by e.g.:

$$H = \lambda f.\ \mathbf{Lam}\ \ulcorner 0 \urcorner \Big(\mathbf{App}\ (\mathbf{App}\ \ulcorner \mathbf{Mul} \urcorner\ (\mathbf{App}\ f\ (\mathbf{Var}\ \ulcorner 0 \urcorner)))\ (\mathbf{Var}\ \ulcorner 0 \urcorner)\Big)$$

Whenever F implements function $f(x)$, the above evaluates to an index of a program which on input x returns $f(x) \cdot x$. That is, program H transforms any implementation of $f(x)$ into an implementation of $g(x) = f(x) \cdot x$. Further, we note that this transformation is total: $H^{\ulcorner}F^{\urcorner}$ always halts returning some index, whatever F might be.

The above transformation relies on the fact that function f is implemented in the λ-calculus, so that we can exploit our term constructors $\mathbf{Var}, \mathbf{App}, \mathbf{Lam}$. However, the same goal, i.e. turning implementations of f into implementations of g, can be achieved in the framework of the general recursive functions $\mathcal{R}$ without making any reference to the λ-calculus, by exploiting the s-m-n theorem as follows.

First, we start by defining a binary auxiliary function $\mathsf{aux}(i, x)$:

$$\mathsf{aux}(i, x) = \phi_i(x) \cdot x$$

The above function can be written as $\mathsf{aux}(i, x) = \mathsf{mul}(\mathsf{eval1}(i, x), x)$, hence it is a composition of recursive functions, and so it is recursive. We can then pick an index for that: let a be such that $\phi_a(i, x) = \mathsf{aux}(i, x)$. Then, we exploit the s-m-n theorem to define

$$h(i) = \mathsf{s}_1^1(a, i)$$

By construction, h is a recursive total function which satisfies

$$\phi_{h(i)}(x) = \mathsf{aux}(i, x) = \phi_i(x) \cdot x$$

When i is an index of f, i.e. when $\phi_i = f$, we have then

$$\phi_{h(i)}(x) = \phi_i(x) \cdot x = f(x) \cdot x = g(x)$$

Hence, h transforms any index of f into an index of g, and this transformation is computable.

Note that the above h performs essentially the same operation of the above λ-term H. The advantage of constructing the recursive function h instead of writing the program H lies in the fact that writing H requires one to work inside the λ-calculus, while the construction of h relies on the s-m-n theorem and the universal function, only, so it is independent from our choice of the λ-calculus as a reference programming language.

More concretely: had we chosen to use Java programs (or Turing Machines, or ...) to enumerate the set of recursive functions, letting ϕ_i be the function computed by the Java program with index i, then the construction of the λ-term H above becomes irrelevant, since we would need to work with

Java instead of λ, while the construction of h is still solid. Indeed, once the universal function theorem and the s-m-n theorem are established for Java, the definition of h is unchanged.

Convention. The above technique is frequently used when dealing with m-reductions (which we shall see in Sect. 4.8.2). Indeed, the reasoning shown above for constructing h is so common that it is convenient to introduce a custom notation for that. Consequently, with some abuse of notation, we shall write

$$h(i) = \#\Big(\lambda x.\ \phi_i(x) \cdot x\Big)$$

for denoting the h constructed above. More generally:

Definition

Definition 220. *Let b be a* recursive *partial function. We write*

$$h(i) = \#\Big(\lambda x.\ b(i,x)\Big)$$

for the total recursive injective function defined as $h(i) = \mathsf{s}_1^1(j,i)$, where j is an index of b, i.e. $\phi_j(i,x) = b(i,x)$.

Nota Bene. The actual result of the h above depends on the choice of the index j. However, no matter which index j of b is chosen, we always have that $\phi_{h(i)}(x) = b(i,x)$.

Nota Bene 2. While a λ occurs in the notation above, the construction of h does not really involve the λ-calculus, but only the s_n^m function.

Nota Bene 3. Do not forget to check that $b \in \mathcal{R}$ before using the above notation!

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

This is the generalization of Th. 169 for the λ-calculus.

Theorem 221 (Kleene's Fixed Point Theorem (a.k.a. Second Recursion Theorem))**.**

Proof

For each total recursive function f, there is some $n \in \mathbb{N}$ such that

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

Proof. We adapt the proof of Th. 169. By Th. 216, the following is recursive:

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

so $\phi_a = g$ for some a. Taking $n = s(a, a)$, we 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]

Example. This is typically used whenever we want to construct a function which "depends on one of its indexes i". For instance, suppose we want to construct a recursive function $f = \phi_i$ such that

$$f(x) = \phi_i(x) = \chi_{\{i\}}(x) = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

The above ϕ_i detects whether the input x is actually the index i: in such case it returns 1, otherwise it returns 0. In the λ-calculus, we would have used the second recursion theorem for λ on the equation

$$F =_{\beta\eta} (\lambda i x.\ \mathbf{Eq}\ i\ x\ \ulcorner 1 \urcorner\ \ulcorner 0 \urcorner) \ulcorner F \urcorner$$

In the context of the recursive functions, instead we consider the function

$$h(i) = \#\left(\lambda x.\ \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}\right)$$

The above is well defined, since its body $b(i, x) = \chi_{\{i\}}(x)$ is indeed recursive. Therefore, $h \in \mathcal{R}$ is total, and so by the second recursion theorem above we have that for some i

$$\phi_i = \phi_{h(i)}$$

Hence, we get what we wanted:

$$\phi_i(x) = \phi_{h(i)}(x) = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

4.5 Recursive Sets

Recall that a function is *recursive* if and only if it can be implemented by some computer program. We extend this notion to sets: a set A is called *recursive* whenever there exists a program V_A which, given an input n, is able to check whether $n \in A$ or not. That is, V_A returns 1 ("true") on

$n \in A$, and 0 ("false") otherwise. Any such program V_A is called a *verifier* for A. Formally, the existence of such a V_A is equivalent to requiring the characteristic function χ_A to be recursive.

Definition 222. *A set $A \subseteq \mathbb{N}$ is recursive iff the function χ_A is recursive. With some abuse of notation, we write $A \in \mathcal{R}$ iff A is recursive. A program implementing χ_A is called a* verifier.

Definition

Of course, the above coincides with λ-definability for sets.

Exercise 223. *Prove that $A \in \mathcal{R}$ if and only if A is λ-definable.*

Let us state some basic facts. We could exploit the exercise above and transfer these facts from the analogous results on the λ-calculus. Instead, we prove them from scratch, showing how the definition of recursive set can be used.

Lemma 224. *Recursive sets are closed under binary union, binary intersection, and complement. If $A, B \in \mathcal{R}$, then $A \cup B, A \cap B, \bar{A} \in Rec$*

Proof

Proof. $(A \cap B)$ Assume $A, B \in \mathcal{R}$. To prove $A \cap B \in \mathcal{R}$, it suffices to note that

$$\chi_{A \cap B}(n) = \chi_A(n) \cdot \chi_B(n)$$

Then, since the right hand side is a composition of recursive functions, $A \cap B \in \mathcal{R}$.

$(\bar{A})$ Assuming $A \in \mathcal{R}$, we prove that $\bar{A} = A \in \mathcal{R}$ by noting that

$$\chi_{\bar{A}}(n) = 1 - \chi_A(n)$$

Again, since the right hand side is a composition of recursive functions, $\bar{A} \in \mathcal{R}$.

$(A \cup B)$ Finally, assuming $A, B \in \mathcal{R}$ we prove that $A \cup B \in \mathcal{R}$ by applying De Morgan:

$$A \cup B = \overline{\bar{A} \cap \bar{B}}$$

By the previous results, the right hand side is recursive, hence $A \cup B \in \mathcal{R}$. □

Iterating the previous lemma we get to:

Lemma 225. *The union and intersection of* finitely many *recursive sets is recursive. That is: if $A_1, \ldots, A_n \in \mathcal{R}$, then $A_1 \cup \ldots \cup A_n \in \mathcal{R}$ and $A_1 \cap \ldots \cap A_n \in \mathcal{R}$.*

Proof. By induction, iterating the previous lemma. □

A simple, yet useful result is established below.

Lemma 226. *Finite sets are recursive.*

Proof. Let A be a finite set. We consider three cases.

If $A = \emptyset$, then $\chi_A(n) = 0$ is the constant zero function, which is recursive by definition of $\mathcal{R}$, hence $A \in \mathcal{R}$.

Otherwise, if A is a singleton, i.e. $A = \{a\}$ for some $a \in \mathbb{N}$, then $\chi_A(n) = \mathsf{eq}(n, a)$ and the right hand side is a composition of recursive functions. Hence, $\chi_A \in \mathcal{R}$, so $A \in \mathcal{R}$.

Otherwise, $A = \{a_1, \ldots, a_n\}$ for some $n > 1$. Hence, $A = \{a_1\} \cup \{a_2\} \cup \cdots \cup \{a_n\}$. We just proved that each of the singletons $\{a_i\}$ is recursive. Hence their (finite) union A is recursive by Lemma 225.

□

The lemma below basically states that functions defined by cases are (partial) recursive, provided that we use a $\mathcal{R}$ condition, a $\mathcal{R}$ "then" branch, and a $\mathcal{R}$ "else" branch.

Lemma 227 (if-then-else with a $\mathcal{R}$ guard)**.** *Let $f, g \in \mathcal{R}$ and $A \in \mathcal{R}$. Then,*

$$h(x) = \begin{cases} f(x) & \text{if } x \in A \\ g(x) & \text{otherwise} \end{cases}$$

is a recursive function.

The above can be generalized to multiple variables, e.g. using $h(x, y)$, $f(x, y)$, $g(x, y)$, $\mathsf{pair}(x, y) \in A$.

Proof. Let i, j such that $\phi_i = f, \phi_j = g$. Then, $h(x) = \mathsf{eval1}(\mathsf{cond}(\chi_A(x), i, j), x)$ is a composition of recursive functions. □

Exercise 228. *Why in the proof above it is not sufficient to write h as follows?*

$$h(x) = \chi_A(x) \cdot f(x) + (1 - \chi_A(x)) \cdot g(x)$$

Kleene's Set. The set K_λ is the typical example of non-λ-definable set. We now generalize its definition so that it is no longer centered on the λ-calculus, and we prove that non recursive by adapting our previous proof accordingly.

Definition 229. $\mathsf{K} = \{n | \phi_n(n) \text{ is defined}\}$

Lemma 230. $\mathsf{K} \notin \mathcal{R}$ **Proof**

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

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

would be a recursive function by Lemma 227. Hence, $f = \phi_a$ for some a. However,

- if $a \in \mathsf{K}$, then $\phi_a(a) = f(a) = undefined$, so $a \notin \mathsf{K}$: a contradiction;
- if $a \notin \mathsf{K}$, then $\phi_a(a) = f(a) = 0$, so $a \in \mathsf{K}$: a contradiction.

In any case we reach a contradiction, hence $\mathsf{K} \notin \mathcal{R}$. □

4.6 Rice's theorem.

We now re-state Rice's theorem, essentially repeating the same proof we saw for the λ-calculus in the framework of the recursive functions. We start by generalizing the notion of "A closed under $\beta\eta$" as follows:

Definition **Definition 231.** *A set $A \subseteq \mathbb{N}$ is semantically closed if and only if*

$$\forall i, j. \Big(i \in A \wedge \phi_i = \phi_j \implies j \in A \Big)$$

A useful characterization of semantically closed sets is shown below. Basically, a set is semantically closed if and only if it contains *all* the indexes for some set of recursive functions.

Statement **Lemma 232.** *A is semantically closed if and only if $A = \{i | \phi_i \in \mathcal{F}\}$ for some set of functions $\mathcal{F} \subseteq \mathcal{R}$.*

Proof. ($\Rightarrow$) Let A be a semantically closed set. Then, define $\mathcal{F} = \{\phi_i | i \in A\}$. It is then easy to check the thesis: indeed,

- If $i \in A$, $\phi_i \in \mathcal{F}$ by definition of $\mathcal{F}$.
- If $\phi_i \in \mathcal{F}$, then by definition of $\mathcal{F}$ there exists j such that $\phi_i = \phi_j$ and $j \in A$. Since A is semantically closed, this implies $i \in A$.

($\Leftarrow$) Let $\mathcal{F} \subseteq \mathcal{R}$ and A defined as in the statement. Then A is semantically closed because,

$$i \in A \wedge \phi_i = \phi_j \implies \phi_i \in \mathcal{F} \wedge \phi_i = \phi_j \implies \phi_j \in \mathcal{F} \implies j \in A$$

□

We can now state Rice's theorem. Compare this with Th. 176.

Theorem 233 (Rice)**.** *Let $A \subseteq \mathbb{N}$ such that*

1. *A is semantically closed;*
2. *$A \neq \emptyset$;*
3. *$A \neq \mathbb{N}$.*

Then, $A \notin \mathcal{R}$.

Proof

Proof. (*Note*: for the exam, you can choose between this proof or the alternative one which we provide after the Rice-Shapiro theorem.)

By contradiction, assume 1,2,3 hold but we have $A \in \mathcal{R}$. Since $A \neq \emptyset$, we can pick $a_1 \in A$. Also, since $A \neq \mathbb{N}$, we can pick $a_0 \notin A$. Then, we let

$$h(n) = \#\left(\lambda x. \begin{cases} \phi_{a_0}(x) & \text{if } n \in A \\ \phi_{a_1}(x) & \text{otherwise} \end{cases}\right)$$

The body of above is recursive by Lemma 227, hence h is recursive total. By the second recursion theorem, we have $\phi_i = \phi_{h(i)}$ for some i. However:

- If $i \in A$, then $\phi_i(x) = \phi_{h(i)}(x) = \phi_{a_0}(x)$ by definition of h. Since A is semantically closed, this implies $a_0 \in A$: a contradiction.
- If $i \notin A$, then $\phi_i(x) = \phi_{h(i)}(x) = \phi_{a_1}(x)$ by definition of h. Since A is semantically closed, and $a_1 \in A$, we have $i \in A$: a contradiction.

In any case we reach a contradiction; hence we conclude that A can not be recursive. □

4.7 Recursively Enumerable Sets

For the Kleene set K there is no verifier V_{K}, as we proved. Indeed, there is no way to implement the *characteristic* function

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

It is however possible to implement the *semi-characteristic* function

$$\tilde{\chi}_{\mathsf{K}}(n) = \begin{cases} 1 & \text{if } n \in \mathsf{K} \\ undefined & \text{otherwise} \end{cases}$$

Indeed, $\tilde{\chi}_{\mathsf{K}}(n) = \phi_n(n) \cdot 0 + 1 = \mathsf{eval1}(n, n) \cdot 0 + 1$, so it can be computed using a universal program.

Sets having a recursive semi-characteristic function are said to be *recursively enumerable*.

Definition

Definition 234. *A set $A \subseteq \mathbb{N}$ is recursively enumerable ($A \in \mathcal{RE}$) if and only if $\tilde{\chi}_A \in \mathcal{R}$.*

The above definition is actually equivalent to requiring the existence of a program S_A which halts on A, and loops forever on $\bar{A}$. That is, a program which implements a function f whose domain is exactly A. Such a S_A is called a *semi-verifier*.

The next two lemmata formalize the above statement.

Proof

Lemma 235. *$A \in \mathcal{RE}$ if and only if $A = \mathsf{dom}(f)$ for some partial $f \in \mathcal{R}$*

Proof. ($\Rightarrow$) Immediate from $A = \mathsf{dom}(\tilde{\chi}_A)$ and the definition of $\mathcal{RE}$.

($\Leftarrow$) Let $f \in \mathcal{R}$ be such that $\mathsf{dom}(f) = A$. Then, we have $\tilde{\chi}_A(x) = 1 + 0 \cdot f(x)$ since this evaluates to 1 exactly when $f(x)$ is defined, i.e. when $x \in \mathsf{dom}(f)$; otherwise it is undefined. Hence $\tilde{\chi}_A$ is a composition of recursive functions, so it is recursive. □

More concretely, one can characterize $\mathcal{RE}$ sets using the λ-calculus as follows, hence relating $\mathcal{RE}$ sets to their semi-verifiers.

Exercise 236. *$A \in \mathcal{RE}$ if and only if there exists a λ-term S_A such that*

$$\begin{array}{ll} S_A\ulcorner n\urcorner =_{\beta\eta} \mathbf{I} & \text{if } n \in A \\ S_A\ulcorner n\urcorner \text{ unsolvable} & \text{if } n \notin A \end{array}$$

Such a S_A is said to be a semi-verifier *of A.*

Note the difference between a verifier V_A and a semi-verifier S_A for a given set A. A verifier has to halt on all inputs, and always return "true/false" according to whether the input belong to A or not. A semi-verifier instead halts whenever the input belongs to A, and loops forever otherwise. Note that, for instance, a verifier V_A can be used in a guard of an if-then-else to check whether $n \in A$. Instead, a semi-verifier S_A could similarly be used in such a guard, but with the proviso that the "else" branch now becomes unreachable: this in because when $n \notin A$ the evaluation of the guard would loop forever, instead of yielding "false". We shall formalize this fact later, in Lemma 245.

Relating $\mathcal{RE}$ with $\mathcal{R}$. $\mathcal{RE}$ sets are strictly related to recursive sets by the following result: a set A is recursive if and only if both A and $\bar{A}$ are recursively enumerable sets.

Lemma 237. $A \in \mathcal{R} \implies A \in \mathcal{RE}$ **Proof**

Proof. Given V_A, one can construct a semi-verifier by letting

$$S_A = \lambda n.\, V_A\ n\ \mathbf{I}\ \Omega$$

Indeed,

- $n \in A \implies S_A{}^{\ulcorner} n^{\urcorner} =_{\beta\eta} \mathbf{T}\ \mathbf{I}\ \Omega =_{\beta\eta} \mathbf{I}$
- $n \notin A \implies S_A{}^{\ulcorner} n^{\urcorner} =_{\beta\eta} \mathbf{F}\ \mathbf{I}\ \Omega =_{\beta\eta} \Omega$ hence it is unsolvable

Alternative proof, without referring to the λ-calculus:
Direct from the definition of $\hat{\chi}_A$ and Lemma 227 since

$$\tilde{\chi}_A(x) = \begin{cases} 1 & x \in A \\ undefined & \text{otherwise} \end{cases}$$

and the constant 1 function and the always-undefined function are recursive. □

Lemma 238. $A \in \mathcal{RE} \wedge \bar{A} \in \mathcal{RE} \implies A \in \mathcal{R}$ **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. In order to check whether $x \in A$, we run this program:

> For k ranging from 0 to $+\infty$:
> run $S_A{}^{\ulcorner} x^{\urcorner}$ for k steps; if it halts, return "true"
> run $S_{\bar{A}}{}^{\ulcorner} x^{\urcorner}$ for k steps; if it halts, return "false"

Each single iteration of the above loop halts, since we are running the semi-verifiers for a given amount of steps, only. Further, the whole loop eventually *has* to stop. Indeed, either $x \in A$ or $x \in \bar{A}$; therefore either $S_A{}^{\ulcorner} x^{\urcorner}$ halts or $S_{\bar{A}}{}^{\ulcorner} x^{\urcorner}$ halts; hence as soon as k reaches the right number of steps, a semi-verifier is found to stop. When we detect that, we discovered whether $x \in A$ or not. So we "abort" the parallel execution of the other semi-verifier and return the result.

Alternative proof, without referring to the λ-calculus:
We have $\tilde{\chi}_A, \tilde{\chi}_{\bar{A}} \in \mathcal{R}$. Let i, j such that $\phi_i = \tilde{\chi}_A$ and $\phi_j = \tilde{\chi}_{\bar{A}}$. Then, let

$$g(x) = \mu n.(\mathsf{or}(\mathsf{isZero}(\mathsf{T}(i, x, n)), \mathsf{isZero}(\mathsf{T}(j, x, n))) = 1)$$

Such g is recursive, since it is a composition of recursive functions. Also, g is *total* because it composes total functions, and an n satisfying the above always exists. Indeed, whenever $x \in A$, $\mathsf{T}(i, x, n) = 0$ for some n. Instead, when $x \in \bar{A}$, we have $\mathsf{T}(j, x, n) = 0$ for some n.

Then, it is easy to check that

$$\chi_A(x) = \begin{cases} 1 & \text{if } \mathsf{T}(i, x, g(x)) = 0 \\ 0 & \text{otherwise} \end{cases}$$

which is recursive by Lemma 227. □

Exercise 239. *Construct the λ-term of the verifier used in the proof above.*

Proof

Lemma 240. $A \in \mathcal{RE} \wedge \bar{A} \in \mathcal{RE} \iff A \in \mathcal{R}$

Proof. Immediate by the lemmata above. □

More on Kleene's set While K is not recursive, it is recursively enumerable.

Proof

Lemma 241. $\mathsf{K} \in \mathcal{RE}$

Proof. Indeed, using the universal program, we can write a semi-verifier for K. On input n, we just execute the program having index n in input n.

Alternative proof.
We have $\mathsf{K} = \mathsf{dom}(f)$ where $f(n) = \phi_n(n) = \mathsf{eval1}(n, n)$, which is recursive by Th. 216. □

The complement of K instead is the typical example of a non recursively enumerable set.

Proof

Lemma 242. $\bar{\mathsf{K}} \notin \mathcal{RE}$

Proof. Immediate by Lemma 240 and $\mathsf{K} \in \mathcal{RE} \setminus \mathcal{R}$. □

Fundamental Properties of $\mathcal{RE}$ Sets.

Lemma 243. *$\mathcal{RE}$ sets are closed under binary union and intersection. That is:*

$$A, B \in \mathcal{RE} \implies A \cup B \in \mathcal{RE}$$
$$A, B \in \mathcal{RE} \implies A \cap B \in \mathcal{RE}$$

Proof. Let $A, B \in \mathcal{RE}$.

($A \cap B$) We have $\tilde{\chi}_{A\cap B}(x) = \tilde{\chi}_A(x) \cdot \tilde{\chi}_B(x)$ which is recursive, being a composition of recursive functions. **Proof**

($A \cup B$) The proof is similar to the one of Lemma 238. Given two semi-verifiers S_A, S_B for A and B, we execute them "in parallel" to construct a semi-verifier for $A \cup B$. In order to check whether $x \in (A \cup B)$, we run this program:

For k ranging from 0 to $+\infty$:
run $S_A{}^{\ulcorner}x^{\urcorner}$ for k steps; if that halts, stop
run $S_B{}^{\ulcorner}x^{\urcorner}$ for k steps; if that halts, stop

Each single iteration of the above loop halts, since we are running the semi-verifiers for a given amount of steps, only. Further, the whole loop stops only when either $S_A{}^{\ulcorner}x^{\urcorner}$ or $S_B{}^{\ulcorner}x^{\urcorner}$ halt. Indeed, when $x \in A \cup B$ the loop stops as soon as we find a number of steps k which makes a semi-verifier halt. Instead, when $x \notin A \cup B$, the above loop will try all possible values for k, hence it never halts.

Since the program above stops exactly on $A \cup B$, that is therefore $\mathcal{RE}$.

Alternative proof, without referring to the λ-calculus:
We have $\tilde{\chi}_A, \tilde{\chi}_B \in \mathcal{R}$. Let i, j such that $\phi_i = \tilde{\chi}_A$ and $\phi_j = \tilde{\chi}_B$. Then, let

$$g(x) = \mu n.(\mathsf{or}(\mathsf{isZero}(\mathsf{T}(i, x, n)), \mathsf{isZero}(\mathsf{T}(j, x, n))) = 1)$$

Such g is recursive, since it is a composition of recursive functions. It is then easy to check that

$$\tilde{\chi}_{A\cup B}(x) = 1 + 0 \cdot g(x)$$

which is recursive (being a composition of recursive functions). □

Lemma 244. *$\mathcal{RE}$ sets are not closed under complement.* **Proof**

Proof. $\mathsf{K} \in \mathcal{RE}$ but $\bar{\mathsf{K}} \notin \mathcal{RE}$. □

The lemma below is a variant of Lemma 227. It basically states that functions defined by cases are (partial) recursive provided that we use a $\mathcal{RE}$ condition, a recursive "then" branch, and an *undefined* "else" branch. Note that, in general, if we use something other than *undefined* in the "else" branch the function at hand is no longer guaranteed to be recursive.

Lemma 245 (if-then-else with a $\mathcal{RE}$ guard)**.** *Let $f \in \mathcal{R}$ and $A \in \mathcal{RE}$. Then,*

Proof

$$h(n) = \begin{cases} f(n) & \text{if } n \in A \\ undefined & \text{otherwise} \end{cases}$$

is recursive.

The above can be generalized to many variables, as for Lemma 227.

Proof. Direct from $h(n) = f(n) \cdot \tilde{\chi}_A(n)$. □

Alternative Characterizations for $\mathcal{RE}$. The following lemma provides several different characterizations of $\mathcal{RE}$ sets.

Lemma 246 (Characterizations of $\mathcal{RE}$)**.** *All the following properties of a set $A \subseteq \mathbb{N}$ are equivalent*

Proof

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

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

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

The above f is a total recursive function, since it can be implemented using the step-by-step interpreter (or, more pedantically, defined using Kleene's function T).

We have $\mathsf{ran}(f) = A$ because:

- $(\mathsf{ran}(f) \subseteq A)$ If $y \in \mathsf{ran}(f)$ implies $y = f(n)$ for some n, so either $y = x \in A$, or $y = \mathsf{proj1}(n) \in \mathsf{dom}(\phi_a) = A$.

- ($A \subseteq \mathsf{ran}(f)$) If $y \in A$, then $\phi_a(y)$ halts in some number of steps k. Hence $f(\mathsf{pair}(y,k)) = y$, which so belongs to $\mathsf{ran}(f)$.

(2 $\implies$ 3) If $A = \emptyset$, taking $B = \emptyset$ suffices. Otherwise, let $A = \mathsf{ran}(f)$, for a total recursive f. In this case, take $B = \{\mathsf{pair}(x, f(x)) | x \in \mathbb{N}\}$. This B is recursive since:

$$\chi_B(n) = \mathsf{eq}(\mathsf{proj2}(n), f(\mathsf{proj1}(n)))$$

Note that the above is true because f is total: otherwise, the right hand side might be undefined.

Then, we conclude by

$$\{n|\exists m.\ \mathsf{pair}(m,n) \in B\} = \{n|\exists m, x.\ \mathsf{pair}(m,n) = \mathsf{pair}(x, f(x))\} =$$
$$= \{n|\exists m, x.\ m = x \ \wedge\ n = f(x)\} = \{n|\exists x.\ n = f(x)\} = \mathsf{ran}(f) = A$$

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

$$f(x) = \begin{cases} \mathsf{proj2}(x) & \text{if } x \in B \\ undefined & \text{otherwise} \end{cases}$$

Clearly, $f \in \mathcal{R}$ by Lemma 227. Also, $\mathsf{ran}(f) = \{\mathsf{proj2}(x) \mid x \in B\} = A$.

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

> For i ranging from 0 to $+\infty$:
> 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 1)

This can be actually implemented in the λ-calculus using the step-by-step interpreter; alternatively, the function computed by the program above can be defined using the minimalisation operator μ (to try all possible i's) and the Kleene's function T.

Let j be the index of the program above (or any index of the associated function). In order to conclude that $A = \mathsf{ran}(f) \in \mathcal{RE}$ it suffices to check check that $\mathsf{dom}(\phi_j) = \mathsf{ran}(f)$. Concretely:

- ($\mathsf{ran}(f) \subseteq \mathsf{dom}(\phi_j)$) 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 $n \in \mathsf{dom}(\phi_j)$.

- ($\mathsf{dom}(\phi_j) \subseteq \mathsf{ran}(f)$) 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 we proved above form a cycle $1 \implies 2 \implies 3 \implies 4 \implies 1$, so the properties $1, 2, 3, 4$ are equivalent. □

Relating $\mathcal{RE}$ with K_λ.

Lemma 247. $\mathsf{K}_\lambda \in \mathcal{RE}$

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 246, $\mathsf{K}_\lambda \in \mathcal{RE}$. □

Lemma 248. $\overline{\mathsf{K}_\lambda} \notin \mathcal{RE}$

Proof. Immediate by Lemma 240 and $\mathsf{K}_\lambda \in \mathcal{RE} \setminus \mathcal{R}$. □

4.7.1 $\mathcal{R}$ and $\mathcal{RE}$ Predicates

So far, we focused our theory on *sets*, and studied some techniques for proving whether a given set is $\mathcal{R}$ (or $\mathcal{RE}$). In other words, we focused on how we can prove the (non-) existence of a verifier or a semi-verifier which would answer the question "$x \in A$?".

The theory we have seen so far can be extended in a natural way to other properties beyond membership checks of the form "$x \in A$". Indeed, given a predicate $p(n_1, \ldots, n_k)$ over natural numbers having arity k, one can directly define its characteristic and semi-characteristic function as follows.

Definition

Definition 249. *Let $p(n_1, \ldots, n_k)$ be a predicate over $\mathbb{N}^k$. Its characteristic function χ_p and its semi-characteristic function $\tilde{\chi}_p$ are defined as:*

$$\chi_p(n_1, \ldots, n_k) = \begin{cases} 1 & \text{if } p(n_1, \ldots, n_k) \\ 0 & \text{otherwise} \end{cases}$$

$$\tilde{\chi}_p(n_1, \ldots, n_k) = \begin{cases} 1 & \text{if } p(n_1, \ldots, n_k) \\ undefined & \text{otherwise} \end{cases}$$

Then, the definition of $\mathcal{R}$ and $\mathcal{RE}$ predicates can be given.

Definition

Definition 250. *Let $p(n_1, \ldots, n_k)$ be a predicate over $\mathbb{N}^k$. Predicate p is said to be recursive ($p \in \mathcal{R}$) whenever $\chi_p \in \mathcal{R}$, and recursively enumerable ($p \in \mathcal{RE}$) whenever $\tilde{\chi}_p \in \mathcal{R}$. Any implementation of χ_p is said to be a verifier for p, while any implementation of $\tilde{\chi}_p$ is said to be a semi-verifier for p.*

Since a k-tuple of natural numbers can be encoded in/decoded from a single natural number in a recursive way, the notions of $\mathcal{R}$ and $\mathcal{RE}$ predicate are tightly connected with those of $\mathcal{R}$ and $\mathcal{RE}$ set, as we now establish.

Lemma 251 (Transfer lemma)**.** *Let $p(n_1, \ldots, n_k)$ be a predicate over $\mathbb{N}^k$, and let:*

$$A = \{\mathsf{pair}(n_1, \mathsf{pair}(n_2, \mathsf{pair}(\ldots, \mathsf{pair}(n_{k-1}, n_k)))) \mid p(n_1, \ldots, n_k)\}$$

Then:

$$p \in \mathcal{R} \iff A \in \mathcal{R} \qquad\qquad p \in \mathcal{RE} \iff A \in \mathcal{RE}$$

Proof. Left as a simple exercise. It is enough to prove that if either of χ_A or χ_p is recursive, then the other is also such, and similarly for $\tilde{\chi}_A$, $\tilde{\chi}_p$. □

Exercise 252. *Prove that $p(x, y)$ is $\mathcal{R}$ (respectively, $\mathcal{RE}$), then $q(y, x) = p(x, y)$ is also $\mathcal{R}$ ($\mathcal{RE}$).*

Example 253.

- *The predicate $p_1(i, x, k) =$ "the execution of $\phi_i(x)$ halts in exactly k steps" is recursive.*
- *The predicate $p_2(i, x, k) =$ "the execution of $\phi_i(x)$ halts in at most k steps" is recursive.*
- *The predicate $p_3(i, x) =$ "the execution of $\phi_i(x)$ halts (in some number of steps)" is $\mathcal{RE}$ but* not *recursive.*

The examples above immediately descend from previous results. The last predicate p_3 can be proved not to be recursive by showing that if V_{p_3} would exist, then we would have $\mathsf{K} \in \mathcal{R}$ since we could construct a V_{K} — a contradiction. Also, the existence of S_{p_3} can be proved using the fact that there is a universal program, i.e. that the universal function $\mathsf{eval1}(i, x) = \phi_i(x)$ is recursive.

The basic properties of $\mathcal{R}$ and $\mathcal{RE}$ predicates are inherited from those of sets.

Lemma 254. **Statement**

- *The conjunction ($\wedge$), disjunction ($\vee$) of two $\mathcal{R}$ predicates is a $\mathcal{R}$ predicate. The negation ($\neg$) of a $\mathcal{R}$ predicate is a $\mathcal{R}$ predicate.*

- *The conjunction ($\wedge$), disjunction ($\vee$) of two $\mathcal{RE}$ predicates is a $\mathcal{RE}$ predicate.*
- *A predicate p is $\mathcal{R}$ if and only if both p and $\neg p$ are $\mathcal{RE}$.*

Proof. Immediate from Lemma 251 and the analogous results for sets. □

Note that the negation ($\neg$) of a $\mathcal{RE}$ predicate may or may not be a $\mathcal{RE}$ predicate, as for sets.

The above result describes how the decidability of predicates is affected by the *logical connectives* $\wedge, \vee, \neg$. We now proceed by studying how it is affected by *logical quantifiers* $\forall, \exists$. A main fundamental results is the following.

Lemma 255. *A predicate $p(n_1, \ldots, n_k)$ is $\mathcal{RE}$ if and only if there exists a $\mathcal{R}$ predicate $q(x, n_1, \ldots, n_k)$ such that*

$$p(n_1, \ldots, n_k) \iff \exists x \in \mathbb{N}.\ q(x, n_1, \ldots, n_k)$$

Proof. We start by observing a general fact. Let p, q be arbitrary predicates. Consider the sets

$$\begin{array}{ll} A_p = \{\mathsf{pair}(n_1, \mathsf{pair}(n_2, \ldots)) & \mid p(n_1, n_2, \ldots)\} \\ A_q = \{\mathsf{pair}(x, \mathsf{pair}(n_1, \mathsf{pair}(n_2, \ldots))) & \mid q(x, n_1, n_2, \ldots)\} \end{array}$$

The above equations can be simplified by letting $z = \mathsf{pair}(n_1, \mathsf{pair}(n_2, \ldots))$, and exploiting the fact that pair is a bijection. We have:

$$\begin{array}{ll} A_p = \{z & \mid p(\mathsf{proj1}(z), \mathsf{proj1}(\mathsf{proj2}(z)), \ldots)\} \\ A_q = \{\mathsf{pair}(x, z) & \mid q(x, \mathsf{proj1}(z), \mathsf{proj1}(\mathsf{proj2}(z)), \ldots)\} \end{array}$$

By the transfer Lemma 251, "$p \in \mathcal{RE}$" is equivalent to "$A_p \in \mathcal{RE}$". Similarly, $q \in \mathcal{R}$ is equivalent to $A_q \in \mathcal{R}$.

We now prove the statement of the lemma.

($\Rightarrow$) If $p \in \mathcal{RE}$, then $A_p \in \mathcal{RE}$. By Lemma 246(3) there is a $B \in \mathcal{R}$ such that $A_p = \{z \mid \exists x.\mathsf{pair}(x, z) \in B\}$. If we let

$$q(x, n_1, n_2, \ldots) = \text{“}\mathsf{pair}(x, \mathsf{pair}(n_1, \mathsf{pair}(n_2, \ldots))) \in B\text{”}$$

Then we have that $A_q = B$ (simple exercise), hence $A_q \in \mathcal{R}$, and so $q \in \mathcal{R}$. Moreover, $A_p = \{z \mid \exists x.\mathsf{pair}(x, z) \in A_q\}$ implies

$$p(n_1, \ldots, n_k) \iff \exists x \in \mathbb{N}.\ q(x, n_1, \ldots, n_k)$$

($\Leftarrow$) Let q be a recursive predicate satisfying

$$p(n_1, \ldots, n_k) \iff \exists x \in \mathbb{N}.\ q(x, n_1, \ldots, n_k)$$

Then, we have that $A_q \in \mathcal{R}$ and that $A_p = \{z \mid \exists x.\mathsf{pair}(x, z) \in A_q\}$. Hence $A_p \in \mathcal{RE}$ by Lemma 246(3), and so $p \in \mathcal{RE}$. □

The next result proves that $\mathcal{RE}$ predicates are closed under existential quantification.

Proof

Lemma 256. *Let $q(x, n_1, \ldots, n_k)$ be a $\mathcal{RE}$ predicate, and let*

$$p(n_1, \ldots, n_k) = \text{“}\exists x.\ q(x, n_1, \ldots, n_k)\text{”}$$

Then $p \in \mathcal{RE}$.

Proof. By Lemma 255, since $q \in \mathcal{RE}$, there is a predicate $r(y, x, n_1, \ldots) \in \mathcal{R}$ such that:

$$q(x, n_1, \ldots, n_k) \iff \exists y.\ r(y, x, n_1, \ldots, n_k)$$

The above implies

$$p(n_1, \ldots, n_k) \iff \exists x, y.\ r(y, x, n_1, \ldots, n_k)$$

which is equivalent to, since pair is a bijection,

$$p(n_1, \ldots, n_k) \iff \exists z.\ r(\mathsf{proj1}(z), \mathsf{proj2}(z), n_1, \ldots, n_k)$$

To prove that $p \in \mathcal{RE}$ it is then enough to prove that

$$t(z, n_1, \ldots, n_k) = \text{“}r(\mathsf{proj1}(z), \mathsf{proj2}(z), n_1, \ldots, n_k)\text{”}$$

is $\mathcal{R}$. This is easy to see because

$$\chi_t(z, n_1, \ldots, n_k) = \chi_r(\mathsf{proj1}(z), \mathsf{proj2}(z), n_1, \ldots, n_k)$$

and the right hand side is a composition of recursive functions. □

Exercise 257. *Let $p(x, y, z, w)$ be a recursive predicate. Prove that $q(w) =$ “$\exists x, y, z.\ p(x, y, z, w)$” is a $\mathcal{RE}$ predicate.*

Exercise 258. *(Recommended) Prove that $A = \{i \mid \phi_i(3) = \phi_i(5) = 4\} \in \mathcal{RE}$. Also see Sol .337.*

Exercise 259. *Prove that $A \in \mathcal{RE}$ if and only if $A = \{\mathsf{proj2}(b) \mid b \in B\}$ for some $B \in \mathcal{R}$.*

Exercise 260. *Prove that $A \in \mathcal{RE}$ if and only if $A = \{\mathsf{proj2}(b) \mid b \in B\}$ for some $B \in \mathcal{RE}$.*

4.8 Reductions

4.8.1 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 261. *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

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

The notion of $(\lambda + A)$-definability is then derived from the notion of 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 262 (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 263. *Prove the following statements:*

- *$\leq_T$ is a preorder (i.e. 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 264. *Consider the* $(\lambda + \mathsf{K}_\lambda)$*-calculus. Can every partial function* $f \in \mathbb{N} \rightsquigarrow \mathbb{N}$ *be* $(\lambda + \mathsf{K}_\lambda)$*-defined?*

4.8.2 Many-one Reduction

Another useful notion of reduction is the following:

Definition 265 (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**

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

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

To check whether some function f is a m-reduction usually one exploits the following property:

Exercise 266. *Prove that a total* $f \in \mathcal{R}$ *is an m-reduction between* A *and* B *if and only if, for all* n **Statement**

- $n \in A \implies f(n) \in B$
- $n \notin A \implies f(n) \notin B$

Lemma 267. $A \leq_m B \implies A \leq_T B$

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

Proof

Lemma 268. *$\leq_m$ is a preorder, i.e., a reflexive and transitive relation.*

Proof. We have $A \leq_m A$ with m-reduction id (which is total recursive).

Further, if $A \leq_m B$ with m-reduction f and $B \leq_m C$ with m-reduction g, then we have $A \leq_m C$ with m-reduction $h(n) = g(f(n))$ which is indeed total and recursive. □

Proof

Lemma 269. $A \leq_m B \iff \bar{A} \leq_m \bar{B}$

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

$$\Big(n \in A \iff f(n) \in B\Big)$$
is equivalent to
$$\Big(n \notin A \iff f(n) \notin B\Big)$$
which is equivalent to
$$\Big(n \in \bar{A} \iff f(n) \in \bar{B}\Big)$$

□

The following is a fundamental property: any set which is m-reducible to a $\mathcal{R}$ set is $\mathcal{R}$, and any set which is m-reducible to a $\mathcal{RE}$ set is $\mathcal{RE}$.

Lemma 270. *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. If f is any m-reduction between B and A, it is easy to check that

$$\chi_A(x) = \chi_B(f(x)) \qquad \tilde{\chi}_A(x) = \tilde{\chi}_B(f(x))$$

So, $A \in \mathcal{R}$ whenever $B \in \mathcal{R}$. Also, $A \in \mathcal{RE}$ whenever $B \in \mathcal{RE}$.

□

Proof

Lemma 271. $A \leq_m \mathsf{K} \implies A \in \mathcal{RE}$

Proof. Immediate from the lemma above, since $\mathsf{K} \in \mathcal{RE}$. □

Proof

Exercise 272. *Prove that $\mathsf{K} \not\leq_m \bar{\mathsf{K}}$ and $\bar{\mathsf{K}} \not\leq_m \mathsf{K}$.*

See Sol. 336.

Lemma 273. K *is* $\mathcal{RE}$*-complete (or m-complete), that is:* $\mathsf{K} \in \mathcal{RE}$ *and for any* $A \in \mathcal{RE}$, $A \leq_m \mathsf{K}$.

Proof. We have already proved that $\mathsf{K} \in \mathcal{RE}$. For the second part, let A be any $\mathcal{RE}$ set. Consider

$$f(n) = \#\left(\lambda x.\, \tilde{\chi}_A(n)\right)$$

That is, $f(n)$ is returning an index of a program which discards its input, and computes instead $\tilde{\chi}_A(n)$. This f is well-defined since $\tilde{\chi}_A \in \mathcal{R}$. So, f is a total recursive function (by Def. 220).

Let us check that f is an m-reduction from A to K.

$$n \in A \iff \tilde{\chi}_A(n) \text{ defined} \iff \phi_{f(n)}(f(n)) \text{ defined} \iff f(n) \in \mathsf{K}$$

□

Lemma 274. $A \in \mathcal{RE}$ *if and only if* $A \leq_m \mathsf{K}$

Proof. Immediate from the lemmata above. □

Exercise 275. *Verify that*

$$h(n) = \#(\lambda x.\, \phi_n(0))$$

m-reduces $\mathsf{K}^0 = \{n | \phi_n(0)$ *is defined*$\}$ *to* K. *Then, state whether* $\mathsf{K}^0 \in \mathcal{RE}$.

Exercise 276. *Verify that*

$$h(n) = \#(\lambda x.\, \phi_n(n))$$

m-reduces K *to* $\mathsf{K}^0 = \{n | \phi_n(0)$ *is defined*$\}$. *Then, state whether* $\mathsf{K}^0 \in \mathcal{R}$.

Exercise 277. *Verify that*

$$h(n) = \#(\lambda x.\, \phi_n(n))$$

m-reduces $\bar{\mathsf{K}}$ *to* $A = \{n \mid \mathsf{dom}(\phi_n)$ *finite*$\}$. *Then, state whether* $A \in \mathcal{RE}$.

Exercise 278. *Verify that*

$$h(n) = \#\left(\lambda x.\, \begin{cases} undefined & \text{if running } \phi_n(n) \text{ halts within } x \text{ steps} \\ 0 & \text{otherwise} \end{cases}\right)$$

m-reduces $\bar{\mathsf{K}}$ *to* $A = \{n \mid \mathsf{dom}(\phi_n)$ *infinite*$\}$. *Then, state whether* $A \in \mathcal{RE}$.

Exercise 279. *Verify that* $\bar{\mathsf{K}} \leq_m A = \{n \mid \mathsf{dom}(\phi_n) = \mathbb{N}\}$. *Then, state whether* $A \in \mathcal{RE}$.

Exercise 280. *Define* A *so that* $A, \bar{A} \notin \mathcal{RE}$.

Exercise 281. *Verify that*

$$h(n) = \#\left(\lambda x.\ \begin{cases} undefined & \text{if running } \phi_n(n) \text{ halts within } x \text{ steps} \\ x & \text{otherwise} \end{cases}\right)$$

m-reduces $\bar{\mathsf{K}}$ *to* $A = \{n \mid \mathsf{ran}(\phi_n)$ *infinite*$\}$. *Then, state whether* $A \in \mathcal{RE}$.

Exercise 282. *Let* $A = \{n+1 \mid \phi_n(n)$ *is defined*$\}$. *Prove that* $A \leq_m \mathsf{K}$ *and* $\mathsf{K} \leq_m A$.

Exercise 283. *Let* $x \in A$ *and* $y \notin A$. *Prove that*

$$A \setminus \{x\} \leq_m A \qquad \wedge \qquad A \cup \{y\} \leq_m A$$

Exercise 284. *Prove that* $A \in \mathcal{R}$ *if and only if* $A \leq_m \{0, 1, 2, 3\}$.

Exercise 285. *Prove that when* $A \in \mathcal{RE}$ *we have*

$$A \leq_m \{n \mid \forall x.\ \phi_n(2 \cdot x) = 42\}$$

Exercise 286. *Let* $A = \{n \mid \phi_n(2) \neq 42\}$.

- *Let* i *such that* $\phi_i(x) = undefined$ *for all* x. *State whether* $i \in A$.
- *Prove that* A *is not recursive. Then prove that* $\bar{A} = \{n \mid \phi_n(2) = 42\}$ *is also not recursive.*
- *Prove that* $\bar{A}$ *is* $\mathcal{RE}$, *and conclude that* $A \notin \mathcal{RE}$.
- *Also check that* $\bar{\mathsf{K}} \leq_m A$.

Exercise 287. *(Technical) Prove that* $\mathsf{K} \equiv_m \mathsf{K}_\lambda$. *From this, deduce that* $A \in \mathcal{RE}$ *if and only if* $A \leq_m \mathsf{K}_\lambda$.

4.9 Rice-Shapiro Theorem

The Rice-Shapiro theorem provides a general *sufficient* criterion for proving that a set is not $\mathcal{RE}$.

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 288 (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**

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

Proof. Let f be a recursive function. 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:

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

  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:

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

  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 $\mathcal{F}$.

4.9.1 Rice's Theorem, again

Here's an alternative proof to Th. 233, which exploits the Rice-Shapiro theorem instead of the second recursion theorem.

Theorem 289 (Rice)**.** *Let $A \subseteq \mathbb{N}$ be a semantically closed set, $A \neq \emptyset$ and $A \neq \mathbb{N}$. Then, $A \notin \mathcal{R}$.*

Proof. We have $A, \bar{A} \in \mathcal{R}$, so $A, \bar{A} \in \mathcal{RE}$. By Lemma 232, $A = \{i | \phi_i \in \mathcal{F}\}$ for some set of functions $\mathcal{F} \subseteq \mathcal{R}$. That implies that $\bar{A} = \{i | \phi_i \notin \mathcal{F}\} = \{i | \phi_i \in (\mathcal{R} \setminus \mathcal{F})\}$. Let ϕ_i be the always-undefined function $g_\emptyset$, which 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 $\phi_i = g_\emptyset \in \mathcal{F}$. By Rice-Shapiro ($\Leftarrow$), we have $f \in \mathcal{F}$ for all recursive f, hence $\mathcal{F} = \mathcal{R}$, and so $A = \mathbb{N}$.
- If $i \in \bar{A}$, then $\phi_i = g_\emptyset \in (\mathcal{R} \setminus \mathcal{F})$. By Rice-Shapiro ($\Leftarrow$), we have $f \in (\mathcal{R} \setminus \mathcal{F})$ for all recursive f, hence $\mathcal{R} \setminus \mathcal{F} = \mathcal{R}$, and so $\bar{A} = \mathbb{N}$, implying $A = \emptyset$.

□

Also see [Cutland]

Exercise 290. *Check that the statement above is indeed equivalent to the one given in Th. 233. (Exploit Lemma 232)*

Exercise 291. *For any of these sets, state whether the set is $\mathcal{R}$, or $\mathcal{RE} \setminus \mathcal{R}$, or not in $\mathcal{RE}$.*

- $\mathsf{K} \cup \{5\}$
- $\{1, 2, 3, 4\}$
- $\{n | \phi_n(2) = 6\}$
- $\{n | \exists y \in \mathbb{N}.\, \phi_n(y) = 6\}$
- $\{n | \forall y \in \mathbb{N}.\, \phi_n(y) = 6\}$
- $\{n | \phi_n(n) = 6\}$
- $\{n | \mathsf{dom}(\phi_n)$ *is finite*$\}$
- $\{n | \mathsf{dom}(\phi_n)$ *is infinite*$\}$

- $\{n | \phi_n \textit{ is total}\}$
- $\{n+4 | \mathsf{dom}(\phi_n) \textit{ is finite}\}$
- $\{ \lfloor 100/(n+1)^2 \rfloor \mid \mathsf{dom}(\phi_n) \textit{ is infinite}\}$
- $A \cup B$, $A \cap B$, $A \setminus B$ *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* $A \in \mathcal{RE}$
- $\{\mathsf{pair}(n,m) \mid \mathsf{pair}(m,n) \in A\}$ *where* $A \in \mathcal{RE}$
- $\{f(n) \mid n \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$, f *total*
- $\{f(n) \mid n \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$ *(may be non total)*
- $\{n \mid f(n) \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$, f *total*
- $\{n \mid f(n) \in A\}$ *where* $A \in \mathcal{RE}$ *and* $f \in \mathcal{R}$ *(may be non total)*

Exercise 292. *Solve the following related exercises:*

1. *Let f the following function*

$$f(0,i) = i$$
$$f(n+1,i) = \mathsf{pad}(f(n,i))$$

   Prove that $f \in \mathcal{PR}$.

2. *Given $j \in \mathbb{N}$, consider the two sets*

$$A = \{i \mid \phi_i = \phi_j\}$$
$$B = \{i \mid \exists n.\, i = f(n,j)\}$$

   where f is the function above. Prove that $A \notin \mathcal{RE}$, while $B \in \mathcal{R}$. Conclude that the two sets are (very!) different.

Exercise 293. *Consider the Euler's constant e as an infinite sequence of decimal digits*

$$e = 2.71828182846\cdots = d_0\ .\ d_1\ d_2\ d_3\ \cdots$$

- *State whether $f(n) = d_n$ is a recursive function.*
- *Consider $A = \{n \mid \exists k.\, d_k = d_{k+1} = \cdots = d_{k+n-1} = 7\}$. Is $A \in \mathcal{R}$? Is $A \in \mathcal{RE}$?*
- *Consider*

$$B = \left\{ \sum_{i=0}^{n} 10^{n-i} \cdot d_{k+i} \;\middle|\; k, n \in \mathbb{N} \right\}$$

  Prove that $B \subseteq \mathbb{N}$. Do we have $B \in \mathcal{RE}$? Argue whether you think B to be recursive.

Exercise 294. *(Tricky) Construct $A \subseteq B \subseteq C$ such that $A \notin \mathcal{RE}$, $B \in \mathcal{R}$, $C \notin \mathcal{RE}$.*

Exercise 295. *(Hard) Show that $f \in \mathcal{R}$ where*

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

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

Exercise 296. *(Hard) Prove that there exists a bijection $f \in (\mathbb{N} \leftrightarrow \mathbb{N})$ such that $f \notin \mathcal{R}$.*

Exercise 297. *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 298. *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 299. *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 246. Can we conclude that $B \in \mathcal{RE} \setminus \mathcal{R}$?*

Exercise 300. *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.)

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

Chapter 5

Suggestions for the Exam

The following are random suggestions for the written exam.

- When applying Rice-Shapiro, do not forget to specify whether you are using it in the $\Rightarrow$ direction or in the $\Leftarrow$ one.
- Remember that when equating results of partial functions (as in $\phi_i(0) = \phi_j(1)$), we mean that either 1) both sides of the equation are defined, and evaluate to the same natural number, or 2) both sides are undefined.

  Note that, as a consequence, the sets A and B below are different. Indeed, we have $A \in \mathcal{RE}$ while $B \notin \mathcal{RE}$.

  $$A = \{n \mid \phi_n(0) \text{ defined and } \neq 3\} \qquad B = \{n \mid \phi_n(0) \neq 3\}$$

- Avoid misreading $A \in \mathcal{RE}$ as $A \in (\mathcal{RE} \setminus \mathcal{R})$.
- Avoid memorizing all the details in proofs. You should be able to memorize only the key points you need to reconstruct all the rest. Many proofs in these notes are actually "exercises" you should know to solve with very little help from memory. (E.g. how do you prove $\bar{\mathsf{K}} \notin \mathcal{R}$? $\bar{\mathsf{K}} \notin \mathcal{RE}$?)

5.1 In Practice: Common Techniques Recap

Below we provide a list of some techniques which are frequently used when solving exercises. This list is by no means exhaustive: in general, there is no silver bullet which can answer all the questions. Still, most standard questions can be answered using the techniques below.

5.1.1 Justifying $f \in \mathcal{R}$

- ... by writing a program P implementing f, and arguing why it is indeed doing so. $P(x)$ must halt when $x \in \mathsf{dom}(f)$, and loop forever otherwise. We can use the λ-calculus for writing P, but it is not necessary: pseudo-code is perfectly fine.

- ... by using Lemma 227 or Lemma 245. This is possible when f is defined by cases, as follows:

$$f(x) = \begin{cases} g(x) & \text{if } \langle \text{condition} \rangle \\ h(x) & \text{otherwise} \end{cases}$$

  In this case it is possible to claim $f \in \mathcal{R}$ when either 1) the condition is recursive and g, h are also recursive or 2) the condition is only $\mathcal{RE}$, g is recursive, and h is always *undefined* (just having a recursive h is not enough). In the cases in which both 1) and 2) are false, f may be recursive or not recursive depending on the specific case: no general conclusion can be stated.

  Pitfall. Be careful in not claiming $f \notin \mathcal{R}$ when 1) and 2) do not apply. This would be a serious mistake.

- ... by directly using the definition of recursive function. E.g., we express f as a composition of functions which are already known to be recursive. For instance, polynomials can be computed by composing sum and product, which are recursive.

- ... by proving that $\mathsf{dom}(f)$ is finite. In this case an implementation can be written using a finite number of if-then-elses. If $\mathsf{dom}(f) = \{x_1, \ldots, x_n\}$ and $y_i = f(x_i)$ are the corresponding values, then an implementation is as follows:

$$\begin{array}{ll} F(x): & \mathsf{if} \quad x = k_1 \mathsf{then\ return}\ y_1 \\ & \mathsf{else\ if} x = k_2 \mathsf{then\ return}\ y_2 \\ & \ldots \\ & \mathsf{else\ if} x = k_n \mathsf{then\ return}\ y_n \\ & \mathsf{else\ loop\ forever} \end{array}$$

5.1.2 Justifying $f \notin \mathcal{R}$

- ... by contradiction, through a reduction argument. Typical cases include:

– Proving that if, by contradiction, we had $f \in \mathcal{R}$, we also would have some set $A \in \mathcal{R}$, which we define involving f somehow. Then, we prove $A \notin \mathcal{R}$ and reach a contradiction.

Example 301. *Let*

$$f(n) = \begin{cases} 2 \cdot n + 1 & \text{if } n \in \mathsf{K} \\ 4 & \text{otherwise} \end{cases}$$

If, by contradiction f is recursive, so would be the set $A = \{n \mid f(n) = 4\}$, since a verifier $V_A(n)$ can be built by simply computing $f(n)$ (which is feasible since $f \in \mathcal{R}$), and then comparing the result with 4. This always halts because f is total.
Then, we note that $f(n) = 4$ is actually equivalent to $n \notin \mathsf{K}$. Indeed, if $n \notin \mathsf{K}$, we have $f(n) = 4$ by definition of f. Otherwise, if $n \in \mathsf{K}$, we have $f(n) = 2 \cdot n + 1$ which is odd, hence not equal to 4.
Therefore, $A = \bar{\mathsf{K}}$, which we know to be non recursive. This is a contradiction, hence $f \notin \mathcal{R}$.

Example 302. *Let*

$$f(n) = \begin{cases} 2 \cdot n + 1 & \text{if } \phi_n(3) = 5 \\ 4 & \text{otherwise} \end{cases}$$

We can prove $f \notin \mathcal{R}$ as in the previous example. Here, the set $A = \{n \mid f(n) = 4\} = \{n \mid \phi_n(3) \neq 5\}$ can be proved to be non recursive by Rice.

– Proving that if, by contradiction, we had $f \in \mathcal{R}$, we also would have some other function $g \in \mathcal{R}$, which is however known to be non recursive.

Example 303. *$f(n) = \chi_{\mathsf{K}}(n) + 1$ is not recursive. This is because, otherwise, we would have that $g(n) = f(n) - 1$ is recursive because it is a composition of recursive functions (subtraction, constant 1). However, $g(n) = \chi_{\mathsf{K}}(n)$ is known to be non recursive — contradiction.*

- Use some diagonalization argument (usually, this is hard to do). Example: see Theorem 37.

Pitfall. We can **not** use Lemma 227 to justify "$f \notin \mathcal{R}$". Just because the hypotheses of that Lemma do not hold, we can not conclude that the thesis does not hold!

5.1.3 Justifying $A \in \mathcal{R}$

Proving $A \in \mathcal{R}$ may be done ...

- ... by proving $\chi_A \in \mathcal{R}$ (typically not the most convenient way).
- ... by providing a verifier program $V_A(n)$ (in pseudo-code) and proving that it indeed returns "true" when $n \in A$ and "false" otherwise. Note that V_A must take exactly *one* input parameter n, and must halt on every value of n.
- ... by proving A to be λ-definable, doing as above and writing V_B in the λ-calculus.
- ... by proving A to be a finite set, or by proving $\bar{A}$ to be a finite set.
- ... by proving $A \leq_m B$ for some recursive set B (typically not the most convenient way).

Example 304. *Let $A = \{n \mid \exists m.\ m^2 = n\}$. We can prove $A \in \mathcal{R}$ by defining $V_A(n)$ in pseudo-code as follows:*

```
procedure V_A(n) :
for m := 0 to n do
    if m * m = n then return true;
return false
```

The code above works by comparing n to m^2 for m between 0 and n.

- *If $n \in A$, then $n = m^2$ for some m. This also implies $m \leq n$. Hence, the above loop will find the correct m at the m-th iteration and return "true".*
- *If $n \notin A$, then $n \neq m^2$ for all m. In this case, the loop above will not find any m. The loop will exit, and V_A will return "false" in the last line.*

In a nutshell, the above code works because there's no need to try values of m larger than n.

Alternatively, one could use the λ-calculus. Pseudo-code is usually clearer.

$$V_A = \lambda n.\ \mathbf{\Theta} G\ \ulcorner 0 \urcorner$$
$$G = \lambda g m.\ \mathbf{Eq}\ n\ (\mathbf{Mul}\ m\ m)\ \mathbf{T}\ (\mathbf{Lt}\ n\ m\ \mathbf{F}\ (g\ (\mathbf{Succ}\ m)))$$

Another variant:

$$V_A = \lambda n.\ \mathbf{Succ}\ n\ G\ \ulcorner 0 \urcorner$$
$$G = \textit{(as above)}$$

Pitfall. In the example above, it would be wrong to define a two-inputs procedure:

procedure $V_A(n, m)$:
if $m * m = n$ then return true else return false

The above verifies the binary predicate $p_2(n, m) =$"$n = m^2$", **not** the set A, or equivalently the unary predicate $p_1(n) =$"$\exists m.\ n = m^2$". These are very different predicates. For instance,

$$q_2(n, m) = \text{“}\phi_n(n) \text{ halts in } m \text{ steps”}$$

is a recursive predicate, while

$$q_1(n) = \text{“}\exists m.\ \phi_n(n) \text{ halts in } m \text{ steps”}$$

is *not* recursive, since it is the predicate which defines the Kleene's set K.

5.1.4 Justifying $A \notin \mathcal{R}$

Proving $A \notin \mathcal{R}$ may be done ...

- ... by Rice.
- ... by proving $B \leq_m A$ for some set B and then proving $B \notin \mathcal{R}$.
- ... by contradiction, arguing that $A \in \mathcal{R}$ would be impossible.
- ... by diagonalization (typically hard)

5.1.5 Justifying $A \in \mathcal{RE}$

Proving $A \in \mathcal{RE}$ may be done ...

- ... by proving $\tilde{\chi}_A \in \mathcal{R}$ (typically not the most convenient way).
- ... by providing a semi-verifier program $S_A(n)$ (in pseudo-code) and proving that it indeed halts when $n \in A$ and loops forever otherwise.
- ... as above, but writing S_B in the λ-calculus.
- ... by proving A to be recursive.
- ... by proving $n \in A$ to be equivalent to $\exists m.$ property(n, m) where property is $\mathcal{RE}$. In more informal terms, “an existential quantification of a $\mathcal{RE}$ property is still $\mathcal{RE}$”.
- ... by proving $A \leq_m B$ for some $B \in \mathcal{RE}$ (typically not the most convenient way).

5.1.6 Justifying $A \notin \mathcal{RE}$

Proving $A \notin \mathcal{RE}$ may be done ...

- ... by Rice-Shapiro (either direction).
- ... by proving $B \leq_m A$ for some set B and then proving $B \notin \mathcal{RE}$.
- ... by proving $A \notin \mathcal{R}$ and $\bar{A} \in \mathcal{RE}$.
- ... by contradiction, arguing that $A \in \mathcal{RE}$ would be impossible.

5.1.7 More on Verifiers and Semi-verifiers

When defining a verifier V_A for a set A (or V_p for some predicate p), always check that it *halts on all inputs*. Similarly, when defining a semi-verifier S_A (or S_p) always check that it *halts only when the input belongs to* A (when the inputs satisfy the predicate). When writing these, the most common sources of non-termination are explicitly infinite loops (`for i := 0 to` $\infty \cdots$), while loops, recursion, as well as using a non-recursive guard in an if-then-else (e.g., `if` $\phi_{\texttt{i}}(4) = 1$ `then` $\cdots$).

Some conditions which commonly appear in programs are shown below, together with their classification.

condition	**class**
running $\phi_i(x)$ halts	$\mathcal{RE} \setminus \mathcal{R}$
$\cdots$ with result z	$\mathcal{RE} \setminus \mathcal{R}$
running $\phi_i(x)$ halts in exactly k steps	$\mathcal{R}$
$\cdots$ with result z	$\mathcal{R}$
running $\phi_i(x)$ halts in at most k steps	$\mathcal{R}$
$\cdots$ with result z	$\mathcal{R}$
$\phi_i(2) = 3$	$\mathcal{RE} \setminus \mathcal{R}$
$\phi_i(2) = undefined$	not $\mathcal{RE}$
$\phi_i(2) \neq 3$	not $\mathcal{RE}$
$\exists x.\ \phi_i(x) = 3$	$\mathcal{RE} \setminus \mathcal{R}$
$\exists x.\ \phi_i(x) = undefined$	not $\mathcal{RE}$
$\forall x.\ \phi_i(x) = 3$	not $\mathcal{RE}$
$\forall x.\ \phi_i(x) = undefined$	not $\mathcal{RE}$
$\phi_i(4) = \phi_j(2)$	not $\mathcal{RE}$
$\phi_i = \phi_j$	not $\mathcal{RE}$

Exercise 305. *Justify the claims done in the above table.*

5.1.8 Justifying A Being Semantically Closed

This is done by directly applying the definition:

1. Assume some arbitrary $i, j \in \mathbb{N}$ be given such that $\phi_i = \phi_j$
2. Assume $i \in A$.
3. Prove that j must belong to A as well.

The last step is typically done by expanding the definition of A, replacing each occurrence of ϕ_i with ϕ_j, and then applying the definition of A again.

Example 306. *$A = \{n \mid \phi_n(1) = 10\}$ is semantically closed. Indeed, given $\phi_i = \phi_j$ and $i \in A$, we obtain $\phi_i(1) = 10$, hence $\phi_j(1) = 10$, which implies $j \in A$.*

Pitfall. Note that the above technique fails in all the cases in which n occurs anywhere but in an index ϕ_n. For instance:

$$\begin{aligned} A &= \{n \mid \phi_n(0) = n\} \\ B &= \{n \mid \phi_n(n) = 0\} \\ C &= \{n \mid \phi_n(0) = 0 \wedge n > 400\} \end{aligned}$$

All the above sets can not be proved to be semantically closed by just replacing ϕ_i with ϕ_j. If we try, we fail as follows:

$$i \in A \implies \phi_i(0) = i \implies \phi_j(0) = i \not\implies \phi_j(0) = j \implies j \in A$$

where the third implication fails because we can not replace the standalone i with j. Note that the above attempt, does *not* prove that A is not semantically closed – it is just a failed proof attempt which is inconclusive.

It turns out that the above sets A, B, C are indeed not semantically closed, but actually proving that is much less trivial. Fortunately, no theorem of ours requires us to prove that a set is *not* semantically closed, so we never actually need to do that when applying them.

Note that "replacing ϕ_i with ϕ_j" also fails in sets such as $D = \{f(n) \mid \phi_n(0) = 0\}$, where f is a function different from the identity.

Exercise 307. *Attempt to prove that A, B, C are semantically closed and observe the outcome.*

When we can not prove A to be semantically closed, we must refrain from using Rice or Rice-Shapiro directly. Typically, we need a different approach, such as a reduction, instead.

5.1.9 Rice

We can prove $A \notin \mathcal{R}$ as follows:

1. First, check that A is semantically closed.
2. Then, show that $A \neq \emptyset$ by choosing a natural n and proving $n \in A$.
3. Finally, show that $A \neq \mathbb{N}$ by choosing a natural m and proving $m \notin A$.

It is often convenient to pick n and m to be indexes for some recursive function.

Example 308. *Let $A = \{n \mid \phi_n(1) = 10\}$. We prove $A \notin \mathcal{R}$ by Rice as follows:*

1. *A is semantically closed (see Example 306)*
2. *Take the constant function $f(n) = 10$. Being constant, it is recursive. Take any index a such that $\phi_a = f$. Then $a \in A$, since $f(1) = 10$, hence $A \neq \emptyset$.*
3. *Take the constant function $g(n) = 3$. Being constant, it is recursive. Take any index b such that $\phi_b = g$. Then $b \notin A$, since $g(1) = 3 \neq 10$, hence $A \neq \mathbb{N}$.*

We conclude $A \notin \mathcal{R}$.

Pitfall. Note that even if one is able to prove that A is *not* semantically closed, Rice does not allow one to state that A is recursive.

5.1.10 Rice-Shapiro ($\Rightarrow$)

We can prove $A \notin \mathcal{RE}$ as follows:

1. First, check that A is semantically closed. After having done that, we can refer to the associated set of functions $\mathcal{F}_A = \{\phi_i \mid i \in A\}$.
2. Then, choose a function $f \in \mathcal{F}_A$.
3. Finally, prove that for *any* finite restriction g of f we have $g \notin \mathcal{F}_A$.

Pitfall. The last step is the most critical one: we need to prove $g \notin \mathcal{F}_A$ for all possible g, and not just for one such g. Hence, we must avoid to choose a specific g, and instead write a general argument. This means that

by relying *only* on that 1) g is a restriction of f (in symbols, $g \subseteq f$), and 2) the domain of g is finite, we need to prove $g \notin \mathcal{F}_A$, which is typically done by exploiting the definition of A.

Pitfall. Note that if f is already a finite domain function, we will *not* be able to prove the last hypothesis above. This is because in this case f is a finite restriction of itself ($f \subseteq f$ is always true), hence a function g belonging to $\mathcal{F}_A$ does exist, namely $g = f$ itself!

Example 309. *Let* $A = \{n \mid \forall x \in \mathbb{N}.\ \phi_n(x) = 3 \cdot x\}$.

1. *A is semantically closed: indeed, give $\phi_i = \phi_j$, we have*

   $$i \in A \implies (\forall x \in \mathbb{N}.\ \phi_i(x) = 3{\cdot}x) \implies (\forall x \in \mathbb{N}.\ \phi_j(x) = 3{\cdot}x) \implies j \in A$$

   Hence, we have $\mathcal{F}_A = \{f \in \mathcal{R} \mid \forall x \in \mathbb{N}.\ f(x) = 3 \cdot x\}$.

2. *Take $f(n) = 3 \cdot n$. It trivially belongs to $\mathcal{F}_A$.*

3. *Take* any *finite restriction g of f. Since g has finite domain, we must have $g(k) = undefined$ as soon as k is large enough. Taking any such k, we have $g(k) = undefined \neq 3 \cdot k$, hence $g \notin \mathcal{F}_A$.*

We conclude $A \notin \mathcal{RE}$.

Final note: in the example above we could alternatively have exploited the fact that $\mathcal{F}_A = \{f\}$, where $f(n) = 3 \cdot n$.

Exercise 310. *Apply the above technique to prove $A = \{n \mid \forall x \in \mathbb{N}.\ \phi_n(2 \cdot x) = 3\} \notin \mathcal{RE}$.*

5.1.11 Rice-Shapiro ($\Leftarrow$)

We can prove $A \notin \mathcal{RE}$ as follows:

1. First, check that A is semantically closed. After having done that, we can refer to the associated set of functions $\mathcal{F}_A = \{\phi_i \mid i \in A\}$.

2. Then, choose a function $g \in \mathcal{F}_A$ having a finite domain. (Justify why the domain is finite, and why $g \in \mathcal{F}_A$)

3. Finally, extend g so to define a recursive function f such that $f \notin \mathcal{F}_A$. (Check that f is indeed an extension of g, that it is recursive, and that it does not belong to $\mathcal{F}_A$.)

Note that unlike for Rice-Shapiro ($\Rightarrow$), here we have to choose both f and g. To simplify this task, it is best to choose g to be as much "undefined" as possible, while satisfying the constraint $g \in \mathcal{F}_A$. E.g., when possible, take g to be the always undefined function $g(n) = undefined$ for all n, or take g to be defined only on one or two points. Doing this greatly simplifies the choice of f, which can then be define freely on all points outside the domain of g (as long as f is kept recursive). This freedom can then be exploited to define f outside $\mathcal{F}_A$ as required.

Example 311. *Let* $A = \{n \mid \phi_n(4) = 5 \wedge \phi_n(6) \neq 7\}$.

1. *A is semantically closed: indeed, given $\phi_i = \phi_j$, we have*

   $$i \in A \implies (\phi_i(4) = 5 \wedge \phi_i(6) \neq 7) \implies (\phi_j(4) = 5 \wedge \phi_j(6) \neq 7) \implies j \in A$$

   Hence, we have $\mathcal{F}_A = \{f \in \mathcal{R} \mid f(4) = 5 \wedge f(6) \neq 7\}$.

2. *Take $g(n) = \begin{cases} 5 & \text{if } n = 4 \\ undefined & \text{otherwise} \end{cases}$. Then, g is finite since $\mathsf{dom}(g) = \{4\}$. It belongs to $\mathcal{F}_A$ since $g(4) = 5$ and $g(6) = undefined \neq 7$.*

3. *Take $f(n) = n + 1$ for all n. It is an extension of g, since when $n \in \mathsf{dom}(g)$ we have $n = 4$ hence $g(n) = 5 = 4 + 1 = f(n)$. Moreover, f is recursive since it is the successor function. Finally, f does not belong to $\mathcal{F}_A$ since $f(6) = 6+1 = 7$, violating the requirement $f(6) \neq 7$.*

We conclude $A \notin \mathcal{RE}$.

Exercise 312. *Apply the above technique to prove that $A = \{n \mid \mathsf{dom}(\phi_n) \neq \mathbb{N}\} \notin \mathcal{RE}$.*

5.1.12 Reductions

Justifying $A \leq_m B$ is done as follows:

1. First, we choose h to be a total recursive function (and we justify that claim).

2. Then, we prove that, given $n \in A$, we have $h(n) \in B$.

3. Finally, we prove that, given $n \notin A$, we have $h(n) \notin B$.

Typically, the non trivial part is finding a function h in the first step which allows us to prove the other requirements. The next section provides some suggestions for this.

Alternative approaches to prove $A \leq_m B$ include:

- Proving $A \leq_m C$ and $C \leq_m B$ for some set C.
- When $B = \mathsf{K}$, it suffices to prove $A \in \mathcal{RE}$. (Lemma 273)

How to Construct a Reduction Function h

It sometimes helps to inspect B and check against the following special cases:

- If B is a semantically closed set, it is convenient to look for a reduction function of the following form

$$h(n) = \#(\lambda x.\ b(n, x))$$

It is common for b to be defined by cases, as shown below:

$$h(n) = \#\left(\lambda x. \begin{cases} f(n,x) & \text{if } \langle \text{condition on } n, x\rangle \\ g(n,x) & \text{otherwise} \end{cases}\right)$$

Then, we prove that $b(n, x)$ is a recursive *partial* function. From that, by the s-m-n theorem we get that h is a *total* (and injective) recursive function.

Unfortunately, there is no standard recipe for choosing b (or f, g and the condition). One must try different definitions until one is found to be working.

Note that x may not appear in the actual definition of b, i.e., b might be constant with respect to x. Instead, n has to appear there, since h must return different values according to whether $n \in A$ or not.

When A is defined by a property involving non-termination, such as when $A = \overline{\mathsf{K}}$, it may help picking the condition to be "$\phi_n(\langle\text{something}\rangle)$ halts in $\leq x$ steps", or any variant of such.

Pitfall. Do not use other variables than n, x in the body of the λ, unless you define them separately. E.g., $h(n) = \#(\lambda x.\ x + y)$ is meaningless unless you state what is the y which occurs in such definition.

Pitfall. When defining $b(n, x)$ by cases as shown above, do not forget to justify why b is recursive. Often, Lemma 227 suffices.

- If B is not semantically closed, but is a set of the indexes i for which ϕ_i has some property, it is still worth trying the techniques described in the previous item. (Example: $\mathsf{K} \leq_m B = \{i \mid \phi_i(i) = 0\}$)
- If B is defined as $B = \{f(m) \mid \mathsf{property}(m)\}$, it is worth trying $h = f$.

  Pitfall. In this case, we need to be careful when checking that $h = f$ is a reduction. Note that, for all x we have

  $$x \in B \iff \exists m.\, x = f(m) \wedge \mathsf{property}(m)$$

  Hence, when checking $n \in A \implies f(n) \in B$, the thesis "$f(n) \in B$" is equivalent to $\exists m.\, f(n) = f(m) \wedge \mathsf{property}(m)$. To establish the latter, we can take $m = n$ and just prove $\mathsf{property}(n)$, for instance. This usually poses no problems.

  However, when checking $n \notin A \implies f(n) \notin B$, the thesis "$f(n) \notin B$" is now equivalent to $\neg\exists m.\, f(n) = f(m) \wedge \mathsf{property}(m)$, which is equivalent to $\forall m.\, f(n) = f(m) \implies \neg\mathsf{property}(m)$. To get that, it is *not* enough to just prove $\neg\mathsf{property}(n)$, since other choices of m may exist such that $f(n) = f(m)$, and those have to be checked as well. When f is *injective*, instead, then no other m can exist but n, so in this case checking $\neg\mathsf{property}(n)$ is indeed enough.

  Summary. Always check whether f is injective before proving "$f(n) \notin B$", and act accordingly.

Pitfall. Recall that h has to be *total*. There should be no "undefined" in its definition, except possibly inside a $\#(\lambda x. \cdots)$.

Example 313. *Let $A = \mathsf{K}$ and $B = \{n \mid \phi_n(4) = 5\}$. We prove $A \leq_m B$ as follows. (Note in passing that B is semantically closed)*

1. *Take*

   $$h(n) = \#\left(\lambda x.\, \begin{cases} 5 & \text{if } n \in \mathsf{K} \\ undefined & otherwise \end{cases}\right)$$

   The body of the λx is indeed partial recursive, since the condition "$n \in \mathsf{K}$" is $\mathcal{RE}$, the first case "5" is constant hence recursive, and the second case is "undefined", so Lemma 245 applies. By the s-m-n theorem, h is total and recursive.

2. *Given $n \in A = \mathsf{K}$, we have that $\phi_{h(n)}(x) = \begin{cases} 5 & \text{if } n \in \mathsf{K} \\ undefined & otherwise \end{cases} =$ 5 for all x. In particular, $\phi_{h(n)}(4) = 5$, hence $h(n) \in B$.*

3. *Given* $n \notin A = \mathsf{K}$, *we have that* $\phi_{h(n)}(x) = \begin{cases} 5 & \text{if } n \in \mathsf{K} \\ undefined & otherwise \end{cases} =$ $undefined$ *for all* x. *In particular,* $\phi_{h(n)}(4) = undefined \neq 5$, *hence* $h(n) \notin B$.

Example 314. *Let* $A = \{n \mid \mathsf{dom}(\phi_n) = \{1,2,3\}\}$ *and* $B = \{2 \cdot n + 1 \mid \mathsf{dom}(\phi_n) = \{1,2,3\}\}$. *We prove* $A \leq_m B$ *as follows.*

1. *Take* $h(n) = 2 \cdot n + 1$. *This is obviously total, and it is recursive since it is a composition of recursive functions (sum, multiplication, constants).*

2. *If* $n \in A$, *then* $\mathsf{dom}(\phi_n) = \{1,2,3\}$, *which immediately implies* $h(n) = 2 \cdot n + 1 \in B$.

3. *If* $n \notin A$, *then* $\mathsf{dom}(\phi_n) \neq \{1,2,3\}$. *Since* h *is a strictly increasing function,* h *is injective. Hence, the above implies* $h(n) = 2 \cdot n + 1 \notin B$.

Final note. *Note that since* $A \notin \mathcal{R}$ *(by Rice), the above establishes* $B \notin \mathcal{R}$ *as well. Indeed,* B *is not even* $\mathcal{RE}$, *as we can show by applying Rice-Shapiro (⇐) to* A.

Exercise 315. *Prove that* $\bar{\mathsf{K}} \leq_m B = \{n \mid \mathsf{dom}(\phi_n) = \mathbb{N}\}$ *using the reduction*

$$h(n) = \#\left(\lambda x.\ \begin{cases} 42 & \text{if } \phi_n(n) \text{ does not halt within } x \text{ steps} \\ undefined & otherwise \end{cases}\right)$$

Exercise 316. *Prove that* $A = \{n \mid \forall x.\ \phi_n(x) = 7\} \leq_m B = \{n \mid \mathsf{dom}(\phi_n) = \mathbb{N}\}$ *using the reduction*

$$h(n) = \#\left(\lambda x.\ \begin{cases} 42 & \text{if } \phi_n(x) = 7 \\ undefined & otherwise \end{cases}\right)$$

Exercise 317. *Prove that* $A = \mathsf{K} \leq_m B = \{n \mid \phi_n(0) = 0\}$ *using the reduction*

$$h(n) = \#\left(\lambda x.\ 0 \cdot \phi_n(n)\right)$$

Appendix A

Solutions

Solution 318.

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

Solution 319. *A possible solution is sketched below:*

- *A context $\mathcal{C}[\bullet]$ is a λ-term with a single free occurrence of a variable, denoted as $\bullet$.*
- *Prove that $\mathcal{C}[\bullet] \to_\eta^* \bullet$ if and only if $\mathcal{C}$ is produced by the following grammar:*

$$\mathcal{C} ::= \bullet \ \mid\ \lambda x.\mathcal{C}[\bullet]\ \mathcal{C}[x]$$

- *Prove that, whenever $\mathcal{C}[\bullet] \to_\eta^* \bullet$, we have $\mathcal{C}[\lambda x.\ M]\ N \to_{\beta\eta}^* M\{N/x\}$.*
- *Prove that, if $M \to_\eta^* \to_\beta N$ then $M \to_\beta^* \to_\eta^* N$.*
- *Conclude that, if $M \to_\eta^* \to_\beta^* N$ then $M \to_\beta^* \to_\eta^* N$.*

Solution 320. *By contradiction, assume $\mathbf{T} =_{\beta\eta} \mathbf{F}$. Clearly, $\mathbf{T}$ and $\mathbf{F}$ are $\beta\eta$-normal forms. By Th. 80, 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* □

Solution 321. *Here are many useful combinators:*

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

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

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

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.*

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

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

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

Solution 322. *We want a $\mathbf{\Theta}$ such that $\mathbf{\Theta}F =_{\beta\eta} F(\mathbf{\Theta}F)$. We first write that requirement as $\mathbf{\Theta} =_{\beta\eta} \lambda F.\, F(\mathbf{\Theta}F)$. 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.

$$\begin{aligned}
\$ &= \pounds \\
\pounds &= \lambda abcdefghijklmnopqstuvwxyzr.\, r(thisisafixedpointcombinator)
\end{aligned}$$

Solution 323.

$$\begin{aligned}
&\mathbf{Var} = \mathbf{InL} \\
&\mathbf{App} = \lambda m\ n.\ \mathbf{InR}\ (\mathbf{InL}\ (\mathbf{Pair}\ m\ n)) \\
&\mathbf{Lam} = \lambda i\ m.\ \mathbf{InR}\ (\mathbf{InR}\ (\mathbf{Pair}\ i\ m))
\end{aligned}$$

Solution 324.

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

Solution 325.

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

(Note that the complexity of the above MergeSort is far from the expected $O(n \cdot \log n)$ ...)

Solution 326. *Let F, G be λ-defining partial functions f, g, respectively. Then,*

$$\begin{aligned}
&H = \lambda x.\ J\ F\ (G\ x)\\
&J = G\ x\ (\mathbf{K}\ \mathbf{I})\ \mathbf{I}
\end{aligned}$$

The above H λ-defines $f \circ g$. Indeed:

- *If $g(n)$ is not defined, then $J = G\ \ulcorner n\urcorner\ (\mathbf{K}\ \mathbf{I})\ \mathbf{I}$ is unsolvable because $G\ \ulcorner n\urcorner$ is such. Hence, $H\ \ulcorner n\urcorner =_{\beta\eta} J \cdots$ is unsolvable, as it should since $(f \circ g)(n)$ is undefined.*
- *If $g(n)$ is defined as, say y, then $J = G\ \ulcorner n\urcorner\ (\mathbf{K}\ \mathbf{I})\ \mathbf{I} =_{\beta\eta} \ulcorner y\urcorner\ (\mathbf{K}\ \mathbf{I})\ \mathbf{I} =_{\beta\eta} \mathbf{I}$, hence $H\ \ulcorner n\urcorner =_{\beta\eta} J\ F\ (G\ \ulcorner n\urcorner) =_{\beta\eta} F\ (G\ \ulcorner n\urcorner) =_{\beta\eta} F\ \ulcorner y\urcorner$. The latter is unsolvable whenever $f(y)$ is undefined, or has a numeral $\beta\eta$-normal form. Both these cases agree with the definedness of $(f \circ g)(n)$.*

The λ-term J above is called a "jamming factor": its purpose is to force the evaluation of $G\ulcorner n\urcorner$ before calling F. In this way, composition always works

as expected. For instance, if $F = \lambda x.\ \ulcorner 5 \urcorner$ *and* $G = \Omega$*, naïve composition yields the wrong result* $F(G\ulcorner 7 \urcorner) =_{\beta\eta} \ulcorner 5 \urcorner$*, while* $J\ F(G\ulcorner 7 \urcorner) =_{\beta\eta} \ulcorner 5 \urcorner$ *is unsolvable as it should be, since in this case* g *is the always undefined function.*

Solution 327.

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

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

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

$$G = \lambda x.\ \mathbf{Sd}\ x\ \Omega\ (\lambda mn.\ \mathbf{App}\ n\ m)\ \Omega$$

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

$$G = \lambda x.\ \mathbf{Sd}\ x\ \Omega\ \Omega\ (\lambda im.\ m)$$

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

$$G = \lambda x.\ \mathbf{Sd}\ x\ \Omega\ \Omega\ (\lambda im.\ \mathbf{Sd}\ m\ \Omega\ \Omega\ (\lambda jn.\ \mathbf{Lam}\ j\ (\mathbf{Lam}\ i\ n)))$$

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

$$G = \lambda x.\ \mathbf{Sd}\ x\ \Omega\ (\lambda nm.\ \mathbf{Eq}\ n\ \ulcorner \mathbf{I} \urcorner\ m\ \ulcorner \mathbf{I} \urcorner)\ \Omega$$

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

$$G = \lambda x.\ \mathbf{Sd}\ x\ \Omega\ \Omega\ (\lambda in.\ \mathbf{Lam}\ (\mathbf{Succ}\ i)\ n)$$

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

$$G = \mathbf{\Theta}(\lambda gx.\ \mathbf{Sd}\ x\ \mathbf{Succ}\ (\lambda nm.\ \mathbf{App}\ (gn)\ (gm))\ (\lambda in.\ \mathbf{Lam}\ (\mathbf{Succ}\ i)\ (gn)))$$

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

$$\begin{aligned} G &= \mathbf{\Theta}(\lambda gx.\ \mathbf{Sd}\ x\ V\ A\ L) \\ V &= \lambda i.\ \mathbf{IsZero}\ i\ \ulcorner \mathbf{I} \urcorner\ x \\ A &= \lambda nm.\ \mathbf{App}\ (gn)\ (gm) \\ L &= \lambda in.\ \mathbf{IsZero}\ i\ x\ (\mathbf{Lam}\ i\ (gn)) \end{aligned}$$

Solution 328. *The following program performs substitution while α converting variables.*

$$\begin{aligned}
&\mathbf{Subst} = \mathbf{\Theta}(\lambda s\ i\ m\ n.\ \mathbf{Sd}\ n\ V\ A\ L)\\
&V = \lambda j.\ \mathbf{Eq}\ j\ i\ m\ n\\
&A = \lambda m'\ n'.\ \mathbf{App}\ (s\ i\ m\ m')\ (s\ i\ m\ n')\\
&L = \lambda j\ n'.\ \mathbf{Eq}\ i\ j\ n\ (\mathbf{Lam}\ j'\ (s\ i\ m\ (s\ j\ (\mathbf{Var}\ j')\ n')))\\
&j' = \mathbf{Succ}(\mathbf{Add}\ m\ n)
\end{aligned}$$

Above j' is taken to be $\#M + \#N + 1$, which is larger than the index of any variable occurring in M or N. This ensures that it is fresh, i.e. not free in any of those λ-terms. A "minimum" fresh index could instead be computed with a little more effort.

Note that for this whole program we do not really need the unbounded recursion provided by $\mathbf{\Theta}$: it suffices to go only as deep as the λ-term $\#N$ itself. Since the depth is bounded by $\#N + 1$, we could have exploited that instead.

Solution 329. *The following program follows the algorithm for the leftmost-outermost strategy of Def. 66. The "shallow decoder"* **Sd** *of Ex. 120 is also exploited.*

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

Note that for this task we do not really need unbounded recursion: it suffices to go only as deep as the λ-term itself.

Solution 330.

$$\begin{aligned}
&\mathbf{IsNumeral} = \lambda n.\ \mathbf{Sd}\ n\ (\mathbf{K}\ \mathbf{F})\ (\mathbf{K}(\mathbf{K}\ \mathbf{F}))\ L_1\\
&L_1 = \lambda s\ m.\ \mathbf{Sd}\ m\ (\mathbf{K}\ \mathbf{F})\ (\mathbf{K}(\mathbf{K}\ \mathbf{F}))\ L_2\\
&L_2 = \lambda z\ o.\ \mathbf{And}\ (\mathbf{Neq}\ s\ z)\ (C\ o)\\
&C = \mathbf{\Theta}\Big(\lambda c\ t.\ \mathbf{Sd}\ t\ (\mathbf{Eq}\ z)\ \Big(\lambda m'\ n'.\ \mathbf{And}\ (\mathbf{Eq}\ m'\ (\mathbf{Var}\ s))\ (c\ n')\Big)(\mathbf{K}(\mathbf{K}\ \mathbf{F}))\Big)
\end{aligned}$$

Note that for this task we do not really need unbounded recursion: it suffices to go only as deep as the λ-term itself.

Solution 331.

$$\begin{aligned}
&\mathbf{Extract} = \lambda n.\ \mathbf{Sd}\ n\ (\mathbf{K}\ \Omega)\ (\mathbf{K}(\mathbf{K}\ \Omega))\ L_1\\
&L_1 = \lambda s\ m.\ \mathbf{Sd}\ m\ (\mathbf{K}\ \Omega)\ (\mathbf{K}(\mathbf{K}\ \Omega))\ L_2\\
&L_2 = \lambda z\ o.\ \mathbf{Neq}\ s\ z\ (C\ o)\ \Omega\\
&C = \mathbf{\Theta}(\lambda c\ t.\ \mathbf{Sd}\ t\ V\ A\ L)\\
&V = \lambda v.\ \mathbf{Eq}\ z\ v\ \ulcorner 0 \urcorner\ \Omega\\
&A = \Big(\lambda m'\ n'.\ \mathbf{Eq}\ m'\ (\mathbf{Var}\ s)\ (\mathbf{Succ}\ (c\ n'))\Omega\Big)\\
&L = \mathbf{K}\ (\mathbf{K}\ \Omega)
\end{aligned}$$

Note that for this task we do not really need unbounded recursion: it suffices to go only as deep as the λ-term itself.

Solution 332. *A possible solution is:*

$$\mathbf{Eval} = \mathbf{\Theta}\Big(\lambda e\ n.\ \mathbf{IsBetaNF}\ n\ (\mathbf{Extract}\ (\mathbf{Eta}\ n))(e\ (\mathbf{Beta}\ n))\Big)$$

One can then carefully check that indeed $\mathbf{Eval}\ulcorner M \urcorner$ *performs the required task. When* M *has a* $\beta\eta$*-normal form* N*, that is computed from* M *by repeated leftmost-outermost* β*-reductions, followed by as many steps of* η *as required. If* N *is a numeral,* **Extract** *collects it. If* N *is not a numeral,* **Extract** *returns* Ω*, which is unsolvable. Finally, if* M *has no* $\beta\eta$*-normal form at all, then a leftmost-outermost execution of* $\mathbf{Eval}\ulcorner M \urcorner$ *indeed gets stuck in an infinite recursion, since* **IsBetaNF** n *will always return* **F**. *This can not be unstuck by providing further arguments, hence* $\mathbf{Eval}\ulcorner M \urcorner$ *is unsolvable in this case.*

Solution 333. *Let* A *be a finite set with* n *elements:* $A = \{a_1, \ldots, a_n\}$*. A verifier for* A *is then*

$$\begin{aligned}
V_A = \lambda x.\ &\mathbf{Or}\ (\mathbf{Eq}\ x\ \ulcorner a_1 \urcorner)\ (\\
&\mathbf{Or}\ (\mathbf{Eq}\ x\ \ulcorner a_2 \urcorner)\ (\\
&\ldots\\
&\mathbf{Or}\ (\mathbf{Eq}\ x\ \ulcorner a_n \urcorner)\ (\\
&\mathbf{F}\ )\cdots))
\end{aligned}$$

Solution 334. *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* $\mathbf{K}(M\ulcorner M\urcorner)\ulcorner 0\urcorner = M\ulcorner M\urcorner$ *has a normal form. So* G *actually* λ*-defines* K_λ*, which is a contradiction.* □

Solution 335. *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*

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

Solution 336. *We have* $\bar{\mathsf{K}} \not\leq_m \mathsf{K}$ *because otherwise we would have* $\bar{\mathsf{K}} \in \mathcal{RE}$ *by Lemma 270.*

Also, we have $\mathsf{K} \not\leq_m \bar{\mathsf{K}}$ *because otherwise by Lemma 269 we would have* $\mathbb{N} \setminus \mathsf{K} \leq_m \mathbb{N} \setminus \bar{\mathsf{K}}$ *i.e.* $\bar{\mathsf{K}} \leq_m \mathsf{K}$ *which is false.*

Solution 337. *Let* $A = \{i \mid \phi_i(3) = \phi_i(5) = 4\}$. *We have that*

$$A = \left\{ i \;\middle|\; \exists k_1, k_2. \begin{array}{c} \textit{running } \phi_i(3) \textit{ halts within } k_1 \textit{ steps with output } 4 \\ \wedge \\ \textit{running } \phi_i(5) \textit{ halts within } k_2 \textit{ steps with output } 4 \end{array} \right\}$$

The predicate (of variable i, k_1, k_2*) under the "*$\exists k_1, k_2$*" is recursive, since it is a conjunction of two recursive predicates. Hence, said predicate is also* $\mathcal{RE}$*. By adding the existential quantifiers we get a predicate (of variable* i *alone) which is still* $\mathcal{RE}$ *by Lemma 256. A semi-verifier of the last predicate is actually semi-deciding "*$i \in A$*?", hence* $A \in \mathcal{RE}$*.*

A.1 More Proofs

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

Definition 338. *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 339. *While $\rightarrow_\beta$ and $\rightarrow_p$ are different relations, they have the same transitive reflexive closure, i.e. $\rightarrow_\beta^* = \rightarrow_p^*$.*

Proof. By simple induction. □

Lemma 340. *The (one-step, parallel) relation $\rightarrow_p$ is Church-Rosser:*

$$\forall M\ M_1\ M_2.\ M \rightarrow_p M_1 \wedge M \rightarrow_p M_2 \implies \exists N.\ M_1 \rightarrow_p N \wedge M_2 \rightarrow_p N$$

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

Lemma 341. *The (many-steps, parallel) relation $\rightarrow_p^*$ is Church-Rosser:*

$$\forall M\ M_1\ M_2.\ M \rightarrow_p^* M_1 \wedge M \rightarrow_p^* M_2 \implies \exists N.\ M_1 \rightarrow_p^* N \wedge M_2 \rightarrow_p^* N$$

Proof. By Lemma 340 and induction on the number of steps. □