Computability Theory

Z.L. Low

23rd September 2012

Preface

These notes were originally written as revision notes for a course given at the University of Cambridge by Thomas Forster in Lent 2012. They differ from the official lecture notes in several aspects. For example:

- The official definition of μ-recursive function appearing in these notes allows the application of $\boldsymbol{\mu}[-]$ to a μ-recursive total function. Of course, this is equivalent to the definition where $\boldsymbol{\mu}[-]$ can only be applied to primitive recursive functions.
- The λ-calculus is treated in more detail and is used in an essential way in some proofs in the chapter on machines.
- Register machines are not treated (in the current version).

Acknowledgements

The statements and proofs of some of the results appearing in these notes are taken from Thomas Forster's official lecture notes.

The section on λ-calculus is partly based on Larry Paulson's *Foundations of functional programming* lecture notes.

Contents

— I —

Recursion theory

Before we can begin in earnest, we must first discuss the difference between functions-in-intension and functions-in-extension.

A **function-in-extension** is a (partial) function in the familiar sense: a set Γ of ordered pairs (x, y) such that (x, y) and (x, y') are both in Γ if and only if $y = y'$. The set Γ is also called the **graph** of a function.

For our purposes, a **function-in-intension** is a *finite description* of a (partial) function; for example, it could be a formula like

$$x \mapsto x^2$$

or it could be defined by a logical predicate like

$$n \in \mathbb{N} \text{ and } f(0) = 0 \text{ and } f(1) = 1 \text{ and } f(n + 2) = f(n) + f(n + 1)$$

Every function-in-intension defines a function-in-extension, but we do *not* say that two functions-in-intension are equal even if their graphs are equal. In fact, when two partial functions are said to be equal, what is meant is that their graphs are equal. The definition of function-in-intension is deliberately left vague, as we will be studying several formalisations of this notion which are *a priori* different.

Let $\mathbb{N}$ be the set of natural numbers; we shall always have $0 \in \mathbb{N}$. In this course we study partial functions $\mathbb{N}^k \rightharpoonup \mathbb{N}$ and subsets of $\mathbb{N}$. An elementary counting argument shows that the set of all partial functions $\mathbb{N}^k \rightharpoonup \mathbb{N}$ is uncountable, so we immediately conclude that there are more such functions than we can describe by any finite means. Clearly, this means there are functions we cannot effectively compute, but to prove this we must formalise the theory of effective computability.

1 Primitive recursive functions

Definition 1.1.1. Let $f : X \rightharpoonup Y$ be a partial function. The **domain** of f is the subset $\operatorname{dom} f$ of X such that

$$\operatorname{dom} f = \{x \in X \mid f(x) \text{ is defined}\}$$

and we write $f(x)\downarrow$ if $x \in \operatorname{dom} f$, and $f(x)\uparrow$ if $x \notin \operatorname{dom} f$. A **total function** is a partial function $f : X \rightharpoonup Y$ such that $\operatorname{dom} f = X$; we write $f : X \to Y$ in this case.

Definition 1.1.2. Let $g_1, \dots, g_\ell : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^\ell \rightharpoonup \mathbb{N}$ be functions. A function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ is the **composite** of h and $g_1, \dots, g_\ell$ just when f satisfies the following condition: for all $x_1, \dots, x_k$, if each $g_i(x_1, \dots, x_k)\downarrow$, then

$$f(x_1, \dots, x_k) = h(g_1(x_1, \dots, x_k), \dots, g_\ell(x_1, \dots, x_k))$$

provided the RHS is defined; otherwise $f(x_1, \dots, x_k)\uparrow$.

We write $f = h \circ (g_1, \dots, g_\ell)$ when f is the composite of h and $g_1, \dots, g_\ell$.

Lemma 1.1.3. *If* $g_1, \dots, g_\ell : \mathbb{N}^k \to \mathbb{N}$ *and* $h : \mathbb{N}^\ell \to \mathbb{N}$ *are* total *functions, then their composite* $h \circ (g_1, \dots, g_\ell)$ *is also a total function.* ■

Definition 1.1.4. Let $g : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^k \rightharpoonup \mathbb{N}$ be partial functions. A function $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ is obtained by **primitive recursion** from h and g just when f satisfies the following conditions:

- For all $x_1, \dots, x_k$, if $g(x_1, \dots, x_k)\downarrow$, then

  $$f(x_1, \dots, x_k, 0) = g(x_1, \dots, x_k)$$

  and otherwise $f(x_1, \dots, x_k, 0)\uparrow$.

- For all $x_1, \dots, x_k, y$, if $f(x_1, \dots, x_k, y)\downarrow$, then

  $$f(x_1, \dots, x_k, y + 1) = h(x_1, \dots, x_k, y, f(x_1, \dots, x_k, y))$$

  provided the RHS is defined; otherwise $f(x_1, \dots, x_k, y + 1)\uparrow$.

We write $f = \boldsymbol{\rho}[h; g]$ when this holds.

We will often abuse notation and write $f(\vec{x}, \dots)$ instead of $f(x_1, \dots, x_k, \dots)$ when $\vec{x} = (x_1, \dots, x_k)$. Note that $k = 0$ is allowed in the above definition.

Lemma 1.1.5. *If* $g : \mathbb{N}^k \to \mathbb{N}$ *and* $h : \mathbb{N}^{k+2} \to \mathbb{N}$ *are both* total *functions, then* $\rho[h; g]$ *is a total function* $\mathbb{N}^{k+1} \to \mathbb{N}$.

Proof. Use induction on y in the definition above. ∎

Definition 1.1.6. A **constant function** is a *total* function $f : \mathbb{N}^k \to \mathbb{N}$ with the following property: there is a natural number n such that $f(x_1, \dots, x_k) = n$ for all $x_1, \dots, x_k$.

Remark 1.1.7. Note that every total function $\mathbb{N}^0 \to \mathbb{N}$ is vacuously a constant function, but a partial function $f : \mathbb{N}^0 \rightharpoonup \mathbb{N}$ may have $\operatorname{dom} f = \varnothing$!

Definition 1.1.8. Let $k > 0$, and let $i \in \{1, \dots, k\}$. The i**-th projection function** is the *total* function $\mathsf{pr}^k_i : \mathbb{N}^k \to \mathbb{N}$ such that $\mathsf{pr}^k_i(x_1, \dots, x_k) = x_i$ for all $x_1, \dots, x_k$.

Definition 1.1.9. The **successor function** is the *total* function $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$ such that $\mathsf{suc}(x) = x + 1$ for all x.

Definition 1.1.10. A **primitive recursive function** is any one of the following:

- A constant function $\mathbb{N}^k \to \mathbb{N}$.
- A projection function $\mathsf{pr}^k_i : \mathbb{N}^k \to \mathbb{N}$.
- The successor function $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$.
- A composite $h \circ (g_1, \dots, g_\ell) : \mathbb{N}^k \to \mathbb{N}$ of some primitive recursive functions $g_1, \dots, g_\ell : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^\ell \to \mathbb{N}$.
- A function $\rho[h; g] : \mathbb{N}^{k+1} \to \mathbb{N}$ obtained by primitive recursion on primitive recursive functions $h : \mathbb{N}^{k+2} \to \mathbb{N}$ and $g : \mathbb{N}^k \to \mathbb{N}$.

Formally speaking, we are defining a subset of the set $\{f : \mathbb{N}^k \rightharpoonup \mathbb{N} \mid k \in \mathbb{N}\}$: it is the smallest subset closed under composition and primitive recursion which also contains all constant functions, all projection functions, and the successor function. Lemmas 1.1.3 and 1.1.5 imply that every primitive recursive function is indeed a total function, so there is no abuse of notation in the above definition.

Example 1.1.11. The **predecessor function** is the function $\mathsf{pre} : \mathbb{N} \to \mathbb{N}$ such that

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

This is clearly primitive recursive *by declaration*, i.e. by the form of its definition.

Example 1.1.12. **Bounded subtraction** is the binary operation $\dot{-}$ on $\mathbb{N}$ such that

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= \mathsf{pre}(x \dot{-} y) \end{aligned}$$

This is also primitive recursive by declaration, since pre is primitive recursive.

We could also have defined primitive recursive functions as functions-in-intension constructed by the above rules, but formalising this would take us too far afield into pure syntax. Nonetheless, it is clear that a primitive recursive function $f : \mathbb{N}^k \to \mathbb{N}$, given in the form of a tree of functions-in-intension combined by composition and primitive recursion, is effectively computable in the sense that one could find the value of $f(x_1, \ldots, x_k)$ by unwinding the definitions.

Example 1.1.13. The function-in-extension $\mathbb{N}^2 \to \mathbb{N}$ defined by $(x, y) \mapsto x + y$ is a primitive recursive function, even though the corresponding function-in-intension is not. Indeed, let $g : \mathbb{N} \to \mathbb{N}$ be the identity function $\mathrm{id} : \mathbb{N} \to \mathbb{N}$ (i.e. the projection function $\mathsf{pr}^1_1 : \mathbb{N} \to \mathbb{N}$) and let $h : \mathbb{N}^3 \to \mathbb{N}$ be the composite $\mathsf{suc} \circ \mathsf{pr}^1_3$; one may check that the function $f = \boldsymbol{\rho}[h; g] : \mathbb{N}^2 \to \mathbb{N}$ obtained by primitive recursion satisfies $f(x, y) = x + y$.

Example 1.1.14. Similarly, the function-in-extension defined by $(x, y) \mapsto x \times y$ is (equal to) the graph of a primitive recursive function.

Lemma 1.1.15. *If* $f : \mathbb{N}^{k+1} \to \mathbb{N}$ *is a primitive recursive function, then the two functions*

$$\begin{aligned} (x_1, \ldots, x_k, z) &\mapsto \sum_{y=0}^{z-1} f(x_1, \ldots, x_k, y) \\ (x_1, \ldots, x_k, z) &\mapsto \prod_{y=0}^{z-1} f(x_1, \ldots, x_k, y) \end{aligned}$$

are primitive recursive.

Proof. Write Σ for the first function and Π for the second. One may check that the following are primitive recursive declarations of Σ and Π:

$$\begin{aligned} \Sigma(\vec{x}, 0) &= 0 \\ \Sigma(\vec{x}, z+1) &= \Sigma(\vec{x}, z) + f(\vec{x}, z) \\ \Pi(\vec{x}, 0) &= 1 \\ \Pi(\vec{x}, z+1) &= \Pi(\vec{x}, z) \times f(\vec{x}, z) \end{aligned}$$

■

2 Primitive recursive predicates

In mathematics, one often has to make definitions by cases. Our definition of primitive recursive function does not permit declarations of the form

$$\mathsf{is\text{-}positive}(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x \neq 0 \end{cases}$$

so it seems as if such functions are not primitive recursive. Fortunately, they are, and the class of primitive recursive predicates has some good properties.

If $\varphi(x_1, \ldots, x_k)$ is a logical predicate with free variables $x_1, \ldots, x_k$ ranging over $\mathbb{N}$, let us write $[\varphi]$ for the function $\mathbb{N}^k \to \mathbb{N}$ such that $[\varphi](\vec{x}) = 0$ if $\varphi(\vec{x})$ is false and $[\varphi](\vec{x}) = 1$ if $\varphi(\vec{x})$ is true.

Definition 1.2.1. A **primitive recursive predicate** is a logical predicate φ such that the function $[\varphi]$ is (extensionally equal to) a primitive recursive function.

Lemma 1.2.2. *The function* $\mathsf{is\text{-}positive}$ *defined above is a primitive recursive function, and the predicate* $x \neq 0$ *with one free variable* x *is a primitive recursive predicate.*

Proof. Let $g : \mathbb{N}^0 \to \mathbb{N}$ be the constant 0 function, and let $h : \mathbb{N}^2 \to \mathbb{N}$ be the constant 1 function. One may check that is-positive is extensionally equal to the function $\boldsymbol{\rho}[h; g] : \mathbb{N} \to \mathbb{N}$. ■

Proposition 1.2.3. *The predicate* $x = y$ *with two free variables* x *and* y *is a primitive recursive predicate.*

Proof. Let $\mathsf{is\text{-}zero}(x) = [x = 0]$; one may check that

$$\begin{aligned} \mathsf{is\text{-}zero}(0) &= 1 \\ \mathsf{is\text{-}zero}(x + 1) &= 0 \end{aligned}$$

is a primitive recursive declaration of is-zero. Clearly, $x = y$ if and only if $x \dot{-} y = 0$ and $y \dot{-} x = 0$, so the function $\mathsf{is\text{-}equal} : \mathbb{N}^2 \to \mathbb{N}$ given by

$$\mathsf{is\text{-}equal}(x, y) = \mathsf{is\text{-}zero}(x \dot{-} y) \times \mathsf{is\text{-}zero}(y \dot{-} x)$$

is a primitive recursive function implementing $[x = y]$. ■

Proposition 1.2.4. *Let φ and ψ be primitive recursive predicates with the same free variables. The following are also primitive recursive predicates:*

(i) *The negation $\neg\varphi$.*

(ii) *The conjunction $\varphi \wedge \psi$.*

(iii) *The disjunction $\varphi \vee \psi$.*

(iv) *The implication $\varphi \to \psi$.*

Proof. (i). We use bounded subtraction: $[\neg\varphi]$ is extensionally equal to $1 \dot{-} [\varphi]$.

(ii). We use multiplication:

$$[\varphi \wedge \psi](\vec{x}) = [\varphi](\vec{x}) \times [\psi](\vec{x})$$

(iii). We use addition and is-positive:

$$[\varphi \vee \psi](\vec{x}) = \text{is-positive}([\varphi](\vec{x}) + [\psi](\vec{x}))$$

(iv). Since $\varphi \to \psi$ is logically equivalent to $(\neg\varphi) \vee \psi$, this follows from (i) and (iii). ■

Proposition 1.2.5. *Let φ be a logical predicate with free variables $x_1, \dots, x_k, y$. If φ is a primitive recursive predicate, then the two predicates*

$$\forall y < z.\, \varphi \qquad\qquad \exists y < z.\, \varphi$$

with free variables $x_1, \dots, x_k, z$ are also primitive recursive.

Proof. One readily checks that

$$[\forall y < z.\, \varphi](\vec{x}, z) = \prod_{y=0}^{z-1} [\varphi](\vec{x}, y)$$

$$[\exists y < z.\, \varphi](\vec{x}, z) = \text{is-positive}\left(\sum_{y=0}^{z-1} [\varphi](\vec{x}, y)\right)$$

and by lemma 1.1.15, these are primitive recursive. ■

Proposition 1.2.6. *The function* if-then-else : $\mathbb{N}^3 \to \mathbb{N}$ *defined below is primitive recursive:*

$$\text{if-then-else}(x, y, z) = \begin{cases} y & \textit{if } x \neq 0 \\ z & \textit{if } x = 0 \end{cases}$$

Proof. We may use addition and multiplication to implement if-then-else:

$$\text{if-then-else}(x, y, z) = \text{is-positive}(x) \times y + \text{is-zero}(x) \times z$$

This is primitive recursive by declaration. ∎

3 Pairs and lists

Lemma 1.3.1. *The predicate $x < y$ with two free variables x and y is a primitive recursive predicate.*

Proof. One may check that the primitive recursive declaration below works:

$$\text{is-less-than}(x, y) = \text{is-positive}(y \mathbin{\dot{-}} x)$$ ∎

Theorem 1.3.2. *Bounded minimisation is primitive recursive: if $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is primitive recursive, then there is a primitive recursive function $g : \mathbb{N}^{k+2} \to \mathbb{N}$ with the property that $g(x_1, \dots, x_k, z, w)$ is the smallest natural number y less than w such that $f(x_1, \dots, x_k, y) = z$, and $g(x_1, \dots, x_k, z, w) = w$ if there is no such y.*

Proof. The function g can be defined as follows:

$$g(\vec{x}, z, 0) = 0$$

$$g(\vec{x}, z, w + 1) = \begin{cases} g(\vec{x}, z, w) & \text{if } g(\vec{x}, z, w) < w \\ w & \text{if } g(\vec{x}, z, w) = w \text{ and } f(\vec{x}, w) = z \\ w + 1 & \text{otherwise} \end{cases}$$

By the previous lemma and propositions 1.2.3 and 1.2.6, the above is a primitive recursive declaration for g as required. ∎

Example 1.3.3. Integer division is primitive recursive: we define quotient(x, y) to be the least z less than $x + 1$ such that $x < y \times (z + 1)$, so that quotient(x, y) is the greatest z such that $y \times z \leqslant x$ and $x - y \times z < y$.

Proposition 1.3.4. *There exist three primitive recursive functions,* $\mathsf{pair} : \mathbb{N}^2 \to \mathbb{N}$, $\mathsf{left} : \mathbb{N} \to \mathbb{N}$, *and* $\mathsf{right} : \mathbb{N} \to \mathbb{N}$, *such that*

$$\begin{aligned}\mathsf{left}(\mathsf{pair}(x, y)) &= x \\ \mathsf{right}(\mathsf{pair}(x, y)) &= y \\ \mathsf{pair}(\mathsf{left}(z), \mathsf{right}(z)) &= z\end{aligned}$$

Moreover, for any fixed x, $\mathsf{pair}(x, -)$ *is strictly increasing, and for any fixed* y, $\mathsf{pair}(-, y)$ *is also strictly increasing.*

Proof. We define pair by the arithmetical formula below:

$$\mathsf{pair}(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x$$

One checks that this is a bijection $\mathbb{N}^2 \to \mathbb{N}$ with the required ordering properties, and the functions left and right may be defined by bounded minimisation as in the above theorem, after permuting function arguments where necessary. ∎

Example 1.3.5. The function $f : \mathbb{N} \to \mathbb{N}$ such that

$$\begin{aligned}f(0) &= 0 \\ f(1) &= 1 \\ f(x + 2) &= f(x) + f(x + 1)\end{aligned}$$

is primitive recursive. Indeed, one checks that $f(x) = \mathsf{left}(g(x))$, where the function $g : \mathbb{N} \to \mathbb{N}$ is given by the primitive recursive declaration below:

$$\begin{aligned}g(0) &= \mathsf{pair}(0, 1) \\ g(x + 1) &= \mathsf{pair}(\mathsf{right}(g(x)), \mathsf{left}(g(x)) + \mathsf{right}(g(x)))\end{aligned}$$

A similar technique can be used to implement recursive functions of the form

$$f(\vec{x}, y + n) = h(\vec{x}, y, f(\vec{x}, y), \ldots, f(\vec{x}, y + n - 1))$$

for any fixed positive integer n, but this is not good enough to implement recursive functions which depend on values computed arbitrarily far in the past. For this, we need variable-length lists.

There are several ways of implementing variable-length lists. For example, one could use linked lists: the number 0 represents the empty list, and the number $r + 1$ represents the list obtained by prepending the number $\mathsf{left}(r)$ to the list represented by $\mathsf{right}(r)$. Or we could use Gödel's β function. The important thing is that we have some primitive recursive functions to work with lists:

Theorem 1.3.6. *There is an encoding of finite lists of natural numbers with the following properties:*

- *There exists a primitive recursive function* $\mathsf{list}_n : \mathbb{N}^n \to \mathbb{N}$ *for each natural number* n*, such that* $\mathsf{list}_n(a_0, \dots, a_{n-1})$ *codes the list* $(a_0, \dots, a_{n-1})$*.*
- *There exist primitive recursive functions* $\mathsf{length} : \mathbb{N} \to \mathbb{N}$, $\mathsf{select} : \mathbb{N}^2 \to \mathbb{N}$, $\mathsf{insert} : \mathbb{N}^2 \to \mathbb{N}$, $\mathsf{replace} : \mathbb{N}^3 \to \mathbb{N}$, *and* $\mathsf{slice} : \mathbb{N}^3 \to \mathbb{N}$ *satisfying the specification below.*

Given numbers $a_0, a_1, a_2, \dots$ *with* $a_i = 0$ *for all* $i \geqslant n$*, if* $x = \mathsf{list}_n(a_0, \dots, a_{n-1})$*, then:*

(i) *The number* $\mathsf{length}(x)$ *is* n*.*

(ii) *If* $y < n$ *then* $\mathsf{select}(x, y) = a_y$*, and otherwise* $\mathsf{select}(x, y) = 0$*.*

(iii) *The number* $\mathsf{insert}(x, y)$ *codes the list* $(a_0, \dots, a_{n-1}, y)$*.*

(iv) *Given* $y = \mathsf{list}_m(b_0, \dots, b_{m-1})$*, the number* $\mathsf{replace}(x, y, z)$ *codes the list* $(a_0, \dots, a_{z-1}, b_0, \dots, b_{m-1})$*.*

(v) *If* $y < z$*, then* $\mathsf{slice}(x, y, z)$ *codes the list* $(a_y, \dots, a_{z-1})$*, and otherwise* $\mathsf{slice}(x, y, z) = \mathsf{list}_0(\,)$*.*

Proof. We will use linked lists as described above, so that we have the following equations for $\mathsf{list}_n : \mathbb{N}^n \to \mathbb{N}$:

$$\begin{aligned} \mathsf{list}_0(\,) &= 0 \\ \mathsf{list}_{n+1}(a_0, \dots, a_n) &= \mathsf{pair}(a_0, \mathsf{list}_n(a_1, \dots, a_n)) + 1 \end{aligned}$$

It is readily checked that each $\mathsf{list}_n : \mathbb{N}^n \to \mathbb{N}$ is primitive recursive.

Define an auxiliary function $\mathsf{tail} : \mathbb{N}^2 \to \mathbb{N}$ as follows:

$$\mathsf{tail}(x, 0) = x$$
$$\mathsf{tail}(x, y+1) = \begin{cases} 0 & \text{if } \mathsf{tail}(x, y) = 0 \\ \mathsf{right}((\mathsf{tail}(x, y)) \mathbin{\dot{-}} 1) & \text{otherwise} \end{cases}$$

This is certainly primitive recursive. Since x is an upper bound on the length of the list coded by x, we may use bounded minimisation to compute $\mathsf{length}(x)$: it is the smallest natural number y such that $\mathsf{tail}(x,y) = 0$. We define $\mathsf{select}(x,y)$ as below:

$$\mathsf{select}(x,y) = \begin{cases} 0 & \text{if } \mathsf{tail}(x,y) = 0 \\ \mathsf{left}(\mathsf{tail}(x,y) \dot{-} 1) & \text{otherwise} \end{cases}$$

We define another auxiliary function $\mathsf{splice} : \mathbb{N}^4 \to \mathbb{N}$ by primitive recursion:

$$\begin{aligned} \mathsf{splice}(x,y,w,0) &= y \\ \mathsf{splice}(x,y,w,z+1) &= \mathsf{pair}(\mathsf{select}(x, w \dot{-} z \dot{-} 1), \mathsf{splice}(x,y,z,w)) + 1 \end{aligned}$$

It is now easy to implement the remaining list operations:

$$\begin{aligned} \mathsf{replace}(x,y,z) &= \mathsf{splice}(x,y,z,z) \\ \mathsf{insert}(x,y) &= \mathsf{replace}(x, \mathsf{pair}(y,0)+1, \mathsf{length}(x)) \\ \mathsf{slice}(x,y,z) &= \mathsf{tail}(\mathsf{replace}(x,0,z),y) \end{aligned}$$ ■

We are now able to implement **course-of-values recursion** using primitive recursive functions:

Lemma 1.3.7. *Let* $h : \mathbb{N}^{k+2} \to \mathbb{N}$ *be a primitive recursive function. If the function* $f : \mathbb{N}^{k+1} \to \mathbb{N}$ *satisfies*

$$\begin{aligned} f(\vec{x},0) &= h(\vec{x},0,\mathsf{list}_0(\,)) \\ f(\vec{x},y+1) &= h(\vec{x},y+1,\mathsf{list}_{y+1}(f(\vec{x},0),\dots,f(\vec{x},y))) \end{aligned}$$

then f *is a primitive recursive function.*

Proof. We define an auxiliary function $F : \mathbb{N}^{k+1} \to \mathbb{N}$ by primitive recursion:

$$\begin{aligned} F(\vec{x},0) &= \mathsf{list}_1(h(\vec{x},0,\mathsf{list}_0(\,))) \\ F(\vec{x},y+1) &= \mathsf{insert}(F(\vec{x},y), h(\vec{x},y+1,F(\vec{x},y))) \end{aligned}$$

The desired function can then be obtained by composition:

$$f(\vec{x},y) = \mathsf{select}(F(\vec{x},y),y)$$ ■

Example 1.3.8. Using the sieve of Eratosthenes, the total function $P : \mathbb{N} \to \mathbb{N}$ given by

$$\begin{aligned} P(0) &= 2 \\ P(x+1) &= \text{the least prime number greater than } P(x) \end{aligned}$$

can be implemented by course-of-values recursion, hence, is primitive recursive.

However, are primitive recursive functions rich enough to capture all effectively computable total functions?

4 The Ackermann function and μ-recursive functions

Definition 1.4.1. The **Ackermann function** is the total function $A : \mathbb{N}^2 \to \mathbb{N}$ satisfying the following equations:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m+1, 0) &= A(m, 1) \\ A(m+1, n+1) &= A(m, A(m+1, n)) \end{aligned}$$

Proposition 1.4.2. *The Ackermann function is a well-defined total function.*

Proof. Lexicographically order $\mathbb{N}^2$: so $(m, n) < (m', n')$ if and only if $m < m'$, or $m = m'$ and $n < n'$. The definition above is then seen to be an instance of well-founded recursion, and there is a unique total function satisfying those equations. ■

The Ackermann function is manifestly effectively computable, but the definition above is also not obviously primitive recursive. In fact, the Ackermann function is not primitive recursive at all, as we shall soon see.

Lemma 1.4.3. *For all natural numbers* m *and* n*,*

$$\begin{aligned} m + n &< A(m, n) && (1) \\ A(m, n) &< A(m, n+1) && (2) \\ A(m, n) &< A(m+1, n) && (3) \\ A(m, n+1) &\leqslant A(m+1, n) && (4) \\ A(m, 2n) &< A(m+2, n) && (5) \end{aligned}$$

Proof. Clearly, $A(0, -)$ is strictly increasing and $A(0, n) > n$. Notice that if $A(m, -)$ is strictly increasing and $A(m, 0) > 0$, then $A(m, n) > n$. We continue by induction: suppose $A(m, -)$ is strictly increasing and $A(m, n) > m + n$. Then, $A(m+1, 0) = A(m, 1) > A(m, 0) \geqslant m + 1$, and

$$A(m+1, n+1) = A(m, A(m+1, n)) > A(m+1, n) > m + n + 1$$

as required. This completes the proof of (1) and (2). Since $A(m+1,n) \geqslant n+1$, we have

$$A(m+1,n+1) = A(m, A(m+1,n)) \geqslant A(m,n+1)$$

and $A(m+1,0) = A(m,1) > A(m,0)$, so we have (3). Also, if we assume $A(m+1,n) \geqslant A(m,n+1)$, then

$$A(m+1,n+1) = A(m, A(m+1,n)) \geqslant A(m, A(m,n+1))$$

but by (1) we know $A(m,n+1) \geqslant n+2$, so (4) follows by induction on n from the above and (2). Finally, note that (3) implies $A(m+2,0) > A(m,0)$; taking $A(m+2,n) > A(m,n)$ as our induction hypothesis, we find

$$\begin{aligned} A(m+2,n+1) &= A(m+1, A(m+2,n)) \\ &> A(m+1, A(m,2n)) && \text{by induction hypothesis and (2)} \\ &\geqslant A(m, A(m,2n)+1) && \text{by (4)} \\ &> A(m, 2n+1+1) && \text{by (1) and (2)} \end{aligned}$$

as required. ■

Lemma 1.4.4. *Let* $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ *be a partial function. The following are equivalent:*

(i) *There is a constant* c *such that*

$$\forall \vec{x} \in \operatorname{dom} f.\, f(\vec{x}) < A(c, \max \vec{x})$$

(ii) *There is a constant* c' *such that*

$$\forall \vec{x} \in \operatorname{dom} f.\, f(\vec{x}) < A(c', \|\vec{x}\|)$$

where $\|\vec{x}\| = x_1 + \cdots + x_k$.

Proof. (i) ⇒ (ii). Because $\max \vec{x} \leqslant \|\vec{x}\|$, we have

$$f(\vec{x}) < A(c, \max \vec{x}) \leqslant A(x, \|\vec{x}\|)$$

by inequality (2) of the previous lemma.

(ii) ⇒ (i). Let ℓ be the smallest natural number such that $k \leqslant 2^\ell$. It is clear that $\|\vec{x}\| \leqslant k \max \vec{x} \leqslant 2^\ell \max \vec{x}$, so

$$f(\vec{x}) < A(c, \|\vec{x}\|) \leqslant A\big(c, 2^\ell \max \vec{x}\big) < A(c + 2\ell, \max \vec{x})$$

by inequalities (2) and (5) of the previous lemma. ■

Theorem 1.4.5. *Let $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be the Ackermann function.*

(i) *If $f : \mathbb{N}^k \to \mathbb{N}$ is a primitive recursive function, then there are constants c_f and c'_f such that*

$$\forall \vec{x} \in \mathbb{N}^k.\, f(\vec{x}) < A(c_f, \max \vec{x})$$

$$\forall \vec{x} \in \mathbb{N}^k.\, f(\vec{x}) < A(c'_f, \|\vec{x}\|)$$

(ii) *In particular, the Ackermann function is not primitive recursive.*

Proof. It is clear that inequality (3) of lemma 1.4.3 and (i) together imply (ii), so it is enough to prove (i). By lemma 1.4.4, it is enough to show the existence of either c_f or c'_f. Inequality (1) implies there is such a c_f if f is a constant function or a projection function. We proceed by structural induction on primitive recursive functions.

Suppose $h : \mathbb{N}^\ell \to \mathbb{N}$ and $g_1, \dots, g_\ell : \mathbb{N}^k \to \mathbb{N}$ are primitive recursive functions such that

$$f(\vec{x}) = h(g_1(\vec{x}), \dots, g_\ell(\vec{x}))$$

and we already have constants $c_h, c_{g_1}, \dots, c_{g_\ell}$. Let $m = \max\{c_h, c_{g_1}, \dots, c_{g_\ell}\}$. Then,

$$\begin{aligned}
f(\vec{x}) &= h(g_1(\vec{x}), \dots, g_\ell(\vec{x})) \\
&< A(c_h, \max\{g_1(\vec{x}), \dots, g_\ell(\vec{x})\}) \\
&< A(c_h, \max\{A(c_{g_1}, \max \vec{x}), \dots, A(c_{g_\ell}, \max \vec{x})\}) \\
&\leqslant A(c_h, A(m, \max \vec{x})) \\
&\leqslant A(m, A(m, \max \vec{x})) \\
&< A(m, A(m+1, \max \vec{x})) \\
&= A(m+1, \max \vec{x} + 1) \\
&\leqslant A(m+2, \max \vec{x})
\end{aligned}$$

so $c_f = m + 2$ works.

Now, let $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ be primitive recursive functions, and suppose $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is the primitive recursive function such that

$$\begin{aligned}
f(\vec{x}, 0) &= g(\vec{x}) \\
f(\vec{x}, y+1) &= h(\vec{x}, y, f(\vec{x}, y))
\end{aligned}$$

For induction, we assume there are constants c'_g and c_h such that

$$g(\vec{x}) < A\big(c'_g, \|\vec{x}\|\big)$$

$$h(\vec{x}, y, z) < A(c_h, \max\{x_1, \dots, x_k, y, z\})$$

for all $\vec{x}$. Let $x = \max \vec{x}$ and $m = \max\left\{c'_g, c_h\right\}$. We have

$$f(\vec{x}, 0) = g(x) < A\big(c'_g, \|\vec{x}\|\big) \leqslant A(m+1, \|\vec{x}\| + 0)$$

and we proceed by induction on y: suppose $f(\vec{x}, y) < A(m+1, \|\vec{x}\| + y)$. Then,

$$\begin{aligned} f(\vec{x}, y+1) &= h(\vec{x}, y, f(\vec{x}, y)) \\ &< A(c_h, \max\{x, y, f(\vec{x}, y)\}) \\ &< A(c_h, \max\{x, y, A(m+1, \|\vec{x}\| + y)\}) \end{aligned}$$

But $A(m+1, \|\vec{x}\| + y) > \|\vec{x}\| + y \geqslant \max\{x, y\}$, so

$$\begin{aligned} f(\vec{x}, y+1) &< A(c_h, A(m+1, \|\vec{x}\| + y)) \\ &\leqslant A(m, A(m+1, \|\vec{x}\| + y)) \\ &= A(m+1, \|\vec{x}\| + y + 1) \end{aligned}$$

thus $c'_f = m + 1$ works. ■

Definition 1.4.6. Let $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ be a partial function. A partial function $g : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ is obtained by **unbounded minimisation** from f just when g has the following property: for all $x_1, \dots, x_k$, $g(x_1, \dots, x_k, z)\downarrow$ if and only if there is a y such that $f(x_1, \dots, x_k, y) = z$, and $g(x_1, \dots, x_k, z)$ is the least such y when it exists. We write $g(\vec{x}, z) = \mu_y[f(\vec{x}, y) = z]$ or simply $g = \mu[f]$ when this holds.

Definition 1.4.7. A **μ-recursive function** is any one of the following:

- A constant function $\mathbb{N}^k \to \mathbb{N}$.
- A projection function $\mathrm{pr}^k_i : \mathbb{N}^k \to \mathbb{N}$.
- The successor function $\mathrm{suc} : \mathbb{N} \to \mathbb{N}$.
- A composite $h \circ (g_1, \dots, g_\ell) : \mathbb{N}^k \rightharpoonup \mathbb{N}$ of some μ-recursive functions $g_1, \dots, g_\ell : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^\ell \rightharpoonup \mathbb{N}$.

- A function $\rho[h; g] : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ obtained by primitive recursion on μ-recursive functions $h : \mathbb{N}^{k+2} \rightharpoonup \mathbb{N}$ and $g : \mathbb{N}^k \rightharpoonup \mathbb{N}$.
- A function $\mu[f] : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ obtained by unbounded minimisation from a μ-recursive *total* function $f : \mathbb{N}^{k+1} \to \mathbb{N}$.

The μ-recursive functions are still reasonably computable, though the use of unbounded minimisation means we have to admit the possibility of non-halting computations: after all, there is general mechanical way of *deciding* whether or not there exists y such that $f(\vec{x}, y) = z$. (We will prove a precise form of this claim later.)

Definition 1.4.8. A **μ-recursive predicate** is a logical predicate φ such that the function $[\varphi]$ is a μ-recursive *total* function.

We have just seen that the Ackermann function is not primitive recursive, even though it is an effectively computable total function. Nevertheless, might it be μ-recursive?

Proposition 1.4.9. *The Ackermann function is a μ-recursive total function.*

Proof. The idea is simple enough: since the Ackermann function is defined by well-founded recursion, any computation of $A(m, n)$ must depend on only finitely many other computations of $A(m', n')$ for $(m', n') < (m, n)$ (in the lexicographic ordering of $\mathbb{N}^2$). By using the list operations of theorem 1.3.6, we can recognise a valid course-of-computation of $A(m, n)$, and by using unbounded minimisation, we can compute $A(m, n)$ by finding such a valid course-of-computation.

Let $p : \mathbb{N}^5 \to \mathbb{N}$ be the function defined as follows. We set $p(x, y, z, w, t) = 1$ if any one of the conditions below holds:

- If $y = 0$ and $w = z + 1$.
- If $y > 0$ and $z = 0$ and there is $u < t$ such that

$$\mathsf{left}(\mathsf{select}(x, u)) = \mathsf{pair}(y \dot{-} 1, 1)$$

and $w = \mathsf{right}(\mathsf{select}(x, u))$ for the least such u.

- If $y > 0$ and $z > 0$ and there is $u < t$ such that
$$\mathsf{left}(\mathsf{select}(x, u)) = \mathsf{pair}(y, w \mathbin{\dot{-}} 1)$$
and $v < t$ such that
$$\mathsf{left}(\mathsf{select}(x, v)) = \mathsf{pair}(y \mathbin{\dot{-}} 1, \mathsf{right}(\mathsf{select}(x, u)))$$
for the least such u, and $w = \mathsf{right}(\mathsf{select}(x, v))$ for the least such v.

Otherwise, we set $p(x, y, z, w) = 0$. It is clear that p so defined is a primitive recursive function, since bounded minimisation is primitive recursive by theorem 1.3.2.

Now define $q : \mathbb{N}^2 \to \mathbb{N}$ by the following primitive recursion: set $q(x, 0) = 1$ and set $q(x, t + 1) = 1$ if $\mathsf{length}(x) \geqslant t+1$ and $q(x, t) = 1$ and $p(x, y, z, w, t) = 1$ where $\mathsf{select}(x, t) = \mathsf{pair}(\mathsf{pair}(y, z), w)$. By the construction of p and q, we have $q(x, t) = 1$ if and only if x codes a course-of-computation of the Ackermann function valid up to (but not including) step t. Thus, we may define the Ackermann function using unbounded minimisation as follows: $A(m, n) = w$ where $\mathsf{select}(x, t) = \mathsf{pair}(\mathsf{pair}(m, n), w)$, $t + 1 = \mathsf{length}(x)$, and x is the least number such that $q(x, t + 1) = 1$. ∎

A similar technique may used to show that any total function defined by well-founded recursion on a primitive recursive function (or even a μ-recursive total function) is μ-recursive. Thus, it appears that μ-recursive functions capture a much stronger notion of effective computability than primitive recursive functions.

Finally, we should note that we may assume that a declaration of a μ-recursive function only uses the minimisation operator once:

Theorem 1.4.10. *Let $f : \mathbb{N}^k \to \mathbb{N}$ be a μ-recursive function. Then, there is a primitive recursive function $g : \mathbb{N}^{k+1} \to \mathbb{N}$ such that*
$$f(\vec{x}) = \mathsf{right}(\mu_y[g(\vec{x}, y) = 1])$$

Proof. The idea is that a μ-recursive declaration of f can be translated into a primitive recursive predicate. See [Cohen, 1987, Thm 4.2 and Thm 9.17]. ∎

We will avoid using the above fact where possible.

— II —

Lambda calculus

Suppose we were proscribed the use of natural numbers, and only allowed to manipulate finite strings of symbols. Could we still do arithmetic?

1 λ-terms

Fix an alphabet: $\mathtt{x}, \mathtt{y}, \mathtt{z}, \ldots$. We will think of these as variable names.

Definition 2.1.1. A **λ-term** is a non-empty finite string of symbols of one of the following forms:

- A single letter, e.g. $\mathtt{x}$. Such a λ-term is called an **atomic λ-term**.
- A parenthesised concatenation of any two λ-terms, e.g. $(\mathtt{xy})$.
- A string of the form $[\boldsymbol{\lambda}\mathtt{x}.\, \mathsf{T}]$, where $\mathtt{x}$ is any single letter and T is a λ-term.

A λ-term will usually be denoted by an uppercase letter or a boldface roman word; we reserve typewriter letters for atomic λ-terms. We use $\mathsf{S} \equiv \mathsf{T}$ to mean that that the λ-terms denoted by S and T are equal as strings.

Example 2.1.2. The $\mathbb{K}$ **combinator** is the λ-term below:

$$\mathbb{K} \equiv [\boldsymbol{\lambda}\mathtt{x}.\, [\boldsymbol{\lambda}\mathtt{y}.\, \mathtt{x}]]$$

We also define the $\mathbb{S}$ **combinator** as a λ-term:

$$\mathbb{S} \equiv [\boldsymbol{\lambda}\mathtt{x}.\, [\boldsymbol{\lambda}\mathtt{y}.\, [\boldsymbol{\lambda}\mathtt{z}.\, ((\mathtt{xz})\,(\mathtt{yz}))]]]$$

By convention, the concatenation of λ-terms is assumed left associative, so we may write $xz\,(yz)$ instead of $((xz)\,(yz))$, and to reduce the proliferation of brackets and **λ** symbols, we further abbreviate the definition $\mathbb{S}$ combinator as follows:

$$\mathbb{S} \equiv [\lambda xyz.\, xz\,(yz)]$$

Definition 2.1.3. Let T be a λ-term. The set $\mathrm{FV}(T)$ of **free variables** of T and the set $\mathrm{BV}(T)$ of **bound variables** of T are defined by structural recursion:

- If $T \equiv x$, then $\mathrm{FV}(T) = \{x\}$ and $\mathrm{BV}(T) = \varnothing$.
- If $T \equiv (UV)$, then $\mathrm{FV}(T) = \mathrm{FV}(U) \cup \mathrm{FV}(V)$ and $\mathrm{BV}(T) = \mathrm{BV}(U) \cup \mathrm{BV}(V)$.
- If $T \equiv [\lambda x.\, U]$, then $\mathrm{FV}(T) = \mathrm{FV}(U) \setminus \{x\}$ and $\mathrm{BV}(T) = \mathrm{BV}(U) \cup \{x\}$.

Note that $\mathrm{FV}(T)$ and $\mathrm{BV}(T)$ need not be disjoint!

Example 2.1.4. In the λ-term $([\lambda y.\, x]\, y)$, the variable y is both free and bound. We should treat λ-terms whose free and bound variables overlap with some caution.

Definition 2.1.5. A **closed λ-term** is a λ-term T such that $\mathrm{FV}(T) = \varnothing$.

Example 2.1.6. The combinators $\mathbb{K}$ and $\mathbb{S}$ are closed λ-terms.

Definition 2.1.7. Let S and T be λ-terms, and let x be any single letter. The λ-term $(T \mid x \mapsto S)$ (usually written $T[S \,/\, x]$) obtained by **substituting** S for x in T is defined recursively as follows:

- If $T \equiv x$, then $(T \mid x \mapsto S) \equiv S$; if T is a single letter but $T \not\equiv x$, then $(T \mid x \mapsto S) \equiv T$.
- If $T \equiv (UV)$, then $(T \mid x \mapsto S) \equiv ((U \mid x \mapsto S)\,(V \mid x \mapsto S))$.
- If $T \equiv [\lambda x.\, U]$, then $(T \mid x \mapsto S) = T$; if $y \not\equiv x$ and $T \equiv [\lambda y.\, U]$, then $(T \mid x \mapsto S) \equiv [\lambda y.\, (U \mid x \mapsto S)]$.

Remark 2.1.8. A poorly-chosen substitution may change the meaning of a λ-term. For example, $[\lambda y.\, x]$ is supposed to be interpreted as a constant function taking the value x, but $([\lambda y.\, x] \mid x \mapsto y) \equiv [\lambda y.\, y]$ is interpreted as the identity function! A **safe substitution** is one of the form $(T \mid x \mapsto S)$ where $\mathrm{FV}(S) \cap \mathrm{BV}(T) = \varnothing$.

2 Conversions, reductions, and equality

Definition 2.2.1. A λ-term V is an **α-conversion** of a λ-term T if $\mathsf{T} \equiv [\lambda \mathsf{x}.\, \mathsf{U}]$, y does not occur either free or bound in U, and $\mathsf{V} \equiv [\lambda \mathsf{y}.\, (\mathsf{U} \mid \mathsf{x} \mapsto \mathsf{y})]$. We write $\mathsf{T} \rightsquigarrow_\alpha \mathsf{V}$ when this holds.

Example 2.2.2.

$$\mathbb{K} \rightsquigarrow_\alpha [\lambda \mathsf{zy}.\, \mathsf{z}]$$

Remark 2.2.3. Note that $\rightsquigarrow_\alpha$ is an equivalence relation on the set of λ-terms. We will usually treat λ-terms which are α-conversions of each other as being equal, since they are literally the same after renaming variables. By replacing subterms of T by suitable α-conversions, we may always make $(\mathsf{T} \mid \mathsf{x} \mapsto \mathsf{S})$ into a safe substitution.

Definition 2.2.4. A λ-term V is a **β-conversion** of a λ-term T if $\mathsf{T} \equiv ([\lambda \mathsf{x}.\, \mathsf{U}]\, \mathsf{S})$ and $\mathsf{V} \equiv (\mathsf{U} \mid \mathsf{x} \mapsto \mathsf{S})$ is a *safe* substitution of S for x in U. We write $\mathsf{T} \rightsquigarrow_\beta \mathsf{V}$ when this holds.

Remark 2.2.5. The β-conversion of $[\lambda \mathsf{x}.\, \mathsf{U}]\, \mathsf{S}$ should be interpreted as the result of applying the function $[\lambda \mathsf{x}.\, \mathsf{U}]$ to the term S.

Example 2.2.6. We can use β-reduction to peel off layers of $[\lambda \cdots . \cdots]$:

$$\begin{aligned} \mathbb{S}\mathsf{x} &\rightsquigarrow_\beta [\lambda \mathsf{yz}.\, \mathsf{xz}\,(\mathsf{yz})] \\ [\lambda \mathsf{yz}.\, \mathsf{xz}\,(\mathsf{yz})]\, \mathsf{y} &\rightsquigarrow_\beta [\lambda \mathsf{z}.\, \mathsf{xz}\,(\mathsf{yz})] \\ [\lambda \mathsf{z}.\, \mathsf{xz}\,(\mathsf{yz})]\, \mathsf{z} &\rightsquigarrow_\beta \mathsf{xz}\,(\mathsf{yz}) \end{aligned}$$

Example 2.2.7. We should think of $\mathbb{K}\mathsf{T}$ as being the constant function which evaluates to T:

$$\mathbb{K}\mathsf{T} \rightsquigarrow_\beta [\lambda \mathsf{y}.\, \mathsf{T}]$$

Of course, this only makes sense when y is not free in T.

Definition 2.2.8. A λ-term U is a **η-conversion** of a λ-term T if $\mathsf{T} \equiv [\lambda \mathsf{x}.\, \mathsf{Ux}]$ and x does not occur free in U. We write $\mathsf{T} \rightsquigarrow_\eta \mathsf{U}$ when this holds.

Remark 2.2.9. If $\mathsf{T} \rightsquigarrow_\eta \mathsf{U}$, then we should think of T and U as being extensionally equal λ-terms.

Definition 2.2.10. A λ-term T **reduces to** a λ-term T' when any of the following conditions holds:

- $T \rightsquigarrow_\beta T'$ or $T \rightsquigarrow_\eta T'$.
- $T \equiv [\lambda x.\, U]$, $T' \equiv [\lambda x.\, U']$, and U reduces to U'.
- $T \equiv (UV)$, $T' \equiv (U'V)$, and U reduces to U'.
- $T \equiv (UV)$, $T' \equiv (UV')$, and V reduces to V'.

We write $T \rightsquigarrow T'$ when T reduces to T'. If there is a finite sequence of λ-terms $T_1, \dots, T_n$ such that $T \rightsquigarrow_\alpha T_1$, $T_1 \rightsquigarrow T_2, \dots, T_{n-1} \rightsquigarrow T_n$, $T_n \rightsquigarrow_\alpha T'$, then we write $T \twoheadrightarrow T'$. We allow $n = 1$ here, so that $T \twoheadrightarrow T$.

Example 2.2.11. Recalling example 2.2.6,

$$\mathbb{S}xyz \rightsquigarrow [\lambda yz.\, xz\,(yz)]\, yz \rightsquigarrow [\lambda z.\, xz\,(yz)]\, z \rightsquigarrow xz\,(yz)$$

Definition 2.2.12. A λ-term T is in **normal form** when no non-trivial reductions of T are possible. A **normalisable λ-term** is a λ-term T such that there is a λ-term T' in normal form and $T \twoheadrightarrow T'$.

Example 2.2.13. Let $\boldsymbol{\Omega} \equiv ([\lambda x.\, xx]\, [\lambda x.\, xx])$. Since $\boldsymbol{\Omega} \rightsquigarrow_\beta \boldsymbol{\Omega}$, $\boldsymbol{\Omega}$ is *not* in normal form. In fact, it is not even normalisable.

Example 2.2.14. The $\mathbb{I}$ **combinator** is defined as

$$\mathbb{I} \equiv [\lambda x.\, x]$$

but we have the reductions

$$\mathbb{S}\mathbb{K}\mathbb{K} \rightsquigarrow [\lambda yz.\, \mathbb{K}z\,(yz)]\, \mathbb{K} \rightsquigarrow [\lambda z.\, \mathbb{K}z\,(\mathbb{K}z)] \rightsquigarrow [\lambda z.\, z] \rightsquigarrow_\alpha \mathbb{I}$$

Thus, $\mathbb{S}\mathbb{K}\mathbb{K} \twoheadrightarrow \mathbb{I}$.

Despite the name, reduction can sometimes lead to less simple terms:

Example 2.2.15. The $\mathbb{Y}$ **combinator** is defined as

$$\mathbb{Y} \equiv [\lambda h.\, ([\lambda x.\, h\,(xx)]\, [\lambda x.\, h\,(xx)])]$$

and observe that

$$\mathbb{Y}H \rightsquigarrow_\beta ([\lambda x.\, H\,(xx)]\, [\lambda x.\, H\,(xx)]) \rightsquigarrow_\beta H\,([\lambda x.\, H\,(xx)]\, x]\, [\lambda x.\, H\,(xx)])$$

where the middle λ-term occurs as a strict subterm of the RHS!

Definition 2.2.16. Two λ-terms T and T' are **equal** just if there is a finite sequence of λ-terms $T_1, \ldots, T_{2n-1}$ such that

$$T \twoheadrightarrow T_1 \twoheadleftarrow T_2 \twoheadrightarrow \cdots \twoheadleftarrow T_{2n-2} \twoheadrightarrow T_{2n-1} \twoheadleftarrow T'$$

and we write $T = T'$ when this holds.

Proposition 2.2.17. *Equality of λ-terms as defined above is an equivalence relation and respects the formation of λ-terms, i.e.*

1. *If* $U = U'$*, then* $[\lambda x.\, U] = [\lambda x.\, U']$.
2. *If* $U = U'$*, then* $(UV) = (U'V)$.
3. *If* $V = V'$*, then* $(UV) = (UV')$.

Proof. It is clear that equality is reflexive, symmetric, and transitive. The rest follows from the fact that $\rightsquigarrow$ respects the formation of λ-terms. ■

Actually, in the definition, it suffices to take $n = 1$:

Theorem 2.2.18 (Church–Rosser). *If* $T = T'$*, then there is a λ-term* T'' *such that* $T \twoheadrightarrow T''$ *and* $T' \twoheadrightarrow T''$*. In particular, if* T *is normalisable, then its normal form is unique up to α-conversion.*

Proof. Omitted: see [Church and Rosser, 1936]. □

Notice that the atomic λ-terms x and y are already in normal form, so the Church–Rosser theorem implies that $x \neq y$ (as λ-terms, of course). Thus, not every equation in λ-calculus is valid, which is a relief: otherwise we would not be able to do anything useful with it!

3 Arithmetic and recursion

In order to do arithmetic in λ-calculus, we must first fix an encoding for numbers in λ-terms.

Definition 2.3.1. The **Church numerals** are defined as follows:

$$\begin{aligned}
\underline{0} &\equiv [\lambda \mathtt{fx.\,x}] \\
\underline{1} &\equiv [\lambda \mathtt{fx.\,fx}] \\
&\vdots \\
\underline{n} &\equiv [\lambda \mathtt{fx.}\, \underbrace{\mathtt{f}\,(\cdots(\mathtt{f}}_{n \text{ times}}\ \mathtt{x})\cdots)] \\
&\vdots
\end{aligned}$$

Notice that each $\underline{n}$ is already in normal form and $\underline{n} = \underline{m}$ if and only if $n = m$.

The Church numeral $\underline{n}$ should be interpreted as the operator which iterates a function n times.

Proposition 2.3.2. *There exist λ-terms* **add**, **mul**, *and* **pow** *such that*

$$\begin{aligned}
\mathbf{suc}\,\underline{n} &\twoheadrightarrow \underline{n+1} \\
\mathbf{add}\,\underline{n}\,\underline{m} &\twoheadrightarrow \underline{n+m} \\
\mathbf{mul}\,\underline{n}\,\underline{m} &\twoheadrightarrow \underline{n\times m} \\
\mathbf{pow}\,\underline{n}\,\underline{m} &\twoheadrightarrow \underline{n^m}
\end{aligned}$$

for all Church numerals $\underline{n}$ and $\underline{m}$.

Proof. One simply verifies inductively that these work:

$$\begin{aligned}
\mathbf{suc} &\equiv [\lambda \mathtt{nfx.\,f\,(nfx)}] \\
\mathbf{add} &\equiv [\lambda \mathtt{nmfx.\,mf\,(nfx)}] \\
\mathbf{mul} &\equiv [\lambda \mathtt{nmf.\,m\,(nf)}] \\
\mathbf{pow} &\equiv [\lambda \mathtt{nm.\,mn}]
\end{aligned}$$

■

Remark 2.3.3. The λ-calculus under discussion is untyped, so λ-terms such as $(\mathbf{add}\,UV)$ are legal even when U and V are not equal to Church numerals. However, the behaviour of **add** is left unspecified in that case and we should regard such λ-terms as being devoid of meaning.

We also need to encode the two truth values so that we can talk about λ-terms representing predicates:

Proposition 2.3.4. *There exist λ-terms* $\underline{\top}$, $\underline{\bot}$, *and* **if-then-else** *such that*

$$\begin{aligned}\textbf{if-then-else}\,\underline{\top}\,UV &\twoheadrightarrow U\\ \textbf{if-then-else}\,\underline{\bot}\,UV &\twoheadrightarrow V\end{aligned}$$

for all λ-terms U *and* V.

Proof. First, we define $\underline{\top}$ and $\underline{\bot}$:

$$\begin{aligned}\underline{\top} &\equiv [\lambda xy.\, x]\\ \underline{\bot} &\equiv [\lambda xy.\, y]\end{aligned}$$

These definitions of $\underline{\top}$ and $\underline{\bot}$ make it easy to define **if-then-else**:

$$\textbf{if-then-else} \equiv \mathbb{I}$$ ■

Corollary 2.3.5. *There exist λ-terms* **not**, **and**, *and* **or** *such that*

$$\begin{array}{ccc} & \textbf{and}\,\underline{\top}\,\underline{\top} \twoheadrightarrow \underline{\top} & \textbf{or}\,\underline{\top}\,\underline{\top} \twoheadrightarrow \underline{\top}\\ \textbf{not}\,\underline{\top} \twoheadrightarrow \underline{\bot} & \textbf{and}\,\underline{\top}\,\underline{\bot} \twoheadrightarrow \underline{\bot} & \textbf{or}\,\underline{\top}\,\underline{\bot} \twoheadrightarrow \underline{\top}\\ \textbf{not}\,\underline{\bot} \twoheadrightarrow \underline{\top} & \textbf{and}\,\underline{\bot}\,\underline{\top} \twoheadrightarrow \underline{\bot} & \textbf{or}\,\underline{\bot}\,\underline{\top} \twoheadrightarrow \underline{\top}\\ & \textbf{and}\,\underline{\bot}\,\underline{\bot} \twoheadrightarrow \underline{\bot} & \textbf{or}\,\underline{\bot}\,\underline{\bot} \twoheadrightarrow \underline{\bot}\end{array}$$

Proof. Use **if-then-else**. ■

Lemma 2.3.6. *There exists a λ-term* **is-zero** *such that*

$$\textbf{is-zero}\,\underline{n} \twoheadrightarrow \begin{cases}\underline{\top} & \textit{if}\, n = 0\\ \underline{\bot} & \textit{if}\, n \neq 0\end{cases}$$

for all Church numerals $\underline{n}$.

Proof. Notice that $(\mathbb{K}\,\underline{\bot})$ has the property that

$$(\mathbb{K}\,\underline{\bot}) \cdots (\mathbb{K}\,\underline{\bot})\,\underline{\top} \twoheadrightarrow (\mathbb{K}\,\underline{\bot})\,\underline{\top} \twoheadrightarrow \underline{\bot}$$

so the definition below works:

$$\textbf{is-zero} \equiv [\lambda n.\, n\,(\mathbb{K}\,\underline{\bot})\,\underline{\top}]$$ ■

We previously argued that μ-recursive functions capture a good notion of effective computability, so we should show that μ-recursive functions can be encoded as λ-terms. Notice that a declaration such as

$$\mathsf{F} \equiv [\lambda \mathtt{n}.\, \textbf{if-then-else}\, (\textbf{is-zero}\, \mathtt{n})\, \mathsf{G}\, (\mathsf{F}\, (\textbf{pre}\, \mathtt{n}))]$$

is illegal because F appears in both the LHS and the RHS: recall that λ-terms are finite by definition and so we cannot substitute F on the RHS infinitely many times to get rid of it. However, if we weaken $\equiv$ to $=$ then we *can* construct such a λ-term F using the $\mathbb{Y}$ combinator from example 2.2.15, since $\mathbb{Y}$ satisfies the following equation:

$$\mathbb{Y}\mathsf{H} = \mathsf{H}\, (\mathbb{Y}\mathsf{H})$$

Therefore, if we set

$$\mathsf{H} \equiv [\lambda \mathtt{f}\mathtt{n}.\, \textbf{if-then-else}\, (\textbf{is-zero}\, \mathtt{n})\, \mathsf{G}\, (\mathtt{f}\, (\textbf{pre}\, \mathtt{n}))]$$

then $\mathsf{F} \equiv \mathbb{Y}\mathsf{H}$ has the desired property:

$$\mathsf{F} = \mathsf{H}\mathsf{F} \rightsquigarrow_\beta [\lambda \mathtt{n}.\, \textbf{if-then-else}\, (\textbf{is-zero}\, \mathtt{n})\, \mathsf{G}\, (\mathsf{F}\, (\textbf{pre}\, \mathtt{n}))]$$

Thus we can use the $\mathbb{Y}$ combinator to implement recursive definitions—even non-wellfounded ones! However, to implement primitive recursion we must first implement **pre**, and it turns out we need to first implement pairs using λ-terms.

Proposition 2.3.7. *There exist λ-terms* **pair**, **left**, *and* **right**, *such that the following reductions hold for all λ-terms* U *and* V*:*

$$\begin{aligned} \textbf{left}\, (\textbf{pair}\, \mathsf{U}\mathsf{V}) &\rightsquigarrow \mathsf{U} \\ \textbf{right}\, (\textbf{pair}\, \mathsf{U}\mathsf{V}) &\rightsquigarrow \mathsf{V} \end{aligned}$$

Proof. Using proposition 2.3.4, one may check that these work:

$$\begin{aligned} \textbf{pair} &\equiv [\lambda \mathtt{x}\mathtt{y}\mathtt{t}.\, \textbf{if-then-else}\, \mathtt{t}\mathtt{x}\mathtt{y}] \\ \textbf{left} &\equiv [\lambda \mathtt{z}.\, \mathtt{z}\, \underline{\top}] \\ \textbf{right} &\equiv [\lambda \mathtt{z}.\, \mathtt{z}\, \underline{\bot}] \end{aligned}$$

■

Proposition 2.3.8. *There is a λ-term* **pre** *such that*

$$\begin{aligned} \textbf{pre}\, \underline{0} &\twoheadrightarrow 0 \\ \textbf{pre}\, \underline{n+1} &\twoheadrightarrow \underline{n} \end{aligned}$$

for all Church numerals $\underline{n}$.

Proof. First, define an auxiliary λ-term Γ as below:

$$\Gamma \equiv [\lambda \mathtt{fz}.\, \mathbf{pair}\, (\mathbf{right}\, \mathtt{z})\, (\mathtt{f}\, (\mathbf{right}\, \mathtt{z}))]$$

Thus, for any λ-terms F, U, and V,

$$\Gamma F\, (\mathbf{pair}\, U V) \twoheadrightarrow (\mathbf{pair}\, V\, (F V))$$

and so for all Church numerals $\underline{n}$,

$$(\underline{n+1}\, (\Gamma F))\, (\mathbf{pair}\, U V) \twoheadrightarrow (\mathbf{pair}\, (\underline{n}\, F\, V)\, (\underline{n+1}\, F\, V))$$

by induction on n. One may then check that

$$\mathbf{pre} \equiv [\lambda \mathtt{nfx}.\, \mathbf{left}\, (\mathtt{n}\, (\Gamma \mathtt{f})\, (\mathbf{pair}\, \underline{0}\, \mathtt{x}))]$$

has the required property. ∎

Corollary 2.3.9. (i) *There exists a λ-term* **sub** *such that*

$$\mathbf{sub}\, \underline{n}\, \underline{m} = \begin{cases} \underline{0} & \textit{if}\, m > n \\ \underline{n-m} & \textit{if}\, m \leqslant n \end{cases}$$

for all Church numerals $\underline{n}$ *and* $\underline{m}$.

(ii) *There exists a λ-term* **is-equal** *such that*

$$\mathbf{is\text{-}equal}\, \underline{n}\, \underline{m} = \begin{cases} \underline{\top} & \textit{if}\, n = m \\ \underline{\bot} & \textit{if}\, n \neq m \end{cases}$$

for all Church numerals $\underline{n}$ *and* $\underline{m}$.

Proof. (i). We iterate **pre** to obtain **sub**:

$$\mathbf{sub} \equiv [\lambda \mathtt{nm}.\, \mathtt{m}\, \mathbf{pre}\, \mathtt{n}]$$

(ii). We use **sub**, **is-zero**, and **and**:

$$\mathbf{is\text{-}equal} \equiv [\lambda \mathtt{nm}.\, \mathbf{and}\, (\mathbf{is\text{-}zero}\, (\mathbf{sub}\, \mathtt{nm}))\, (\mathbf{is\text{-}zero}\, (\mathbf{sub}\, \mathtt{mn}))]$$ ∎

When computing using λ-terms, one must be wary of non-terminating reduction strategies:

Example 2.3.10. Recall the λ-term $\mathbf{\Omega} \equiv ([\boldsymbol{\lambda} x.\, xx]\, [\boldsymbol{\lambda} x.\, xx])$, and consider the λ-term $T \equiv ([\boldsymbol{\lambda} y.\, U]\, \mathbf{\Omega})$, where the variable y does not occur free in U. If we β-convert the outermost λ-term first, then

$$T \rightsquigarrow_\beta U$$

but if we try to β-convert the inner λ-terms first, we end up in a loop:

$$T \rightsquigarrow T \rightsquigarrow T \cdots$$

because $\mathbf{\Omega} \rightsquigarrow_\beta \mathbf{\Omega}$!

However, it is a fact that β-converting λ-subterms of a *normalisable* λ-term T from the leftmost outermost to the innermost will always terminate in a finite number of steps, and then η-converting the result yields the normal form of T.

Definition 2.3.11. Let $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ be a partial function. A **λ-representation** of f is a λ-term F such that the following holds:

- For all $x_1, \ldots, x_k$ such that $f(x_1, \ldots, x_k)\downarrow$, the λ-term $\left(F\, \underline{x_1} \cdots \underline{x_k}\right)$ is normalisable and
$$\left(F\, \underline{x_1} \cdots \underline{x_k}\right) \twoheadrightarrow \underline{f(x_1, \ldots, x_k)}$$
- Otherwise, $\left(F\, \underline{x_1} \cdots \underline{x_k}\right)$ is not normalisable.[1]

A **λ-representable function** is a partial function that admits a λ-representation.

Theorem 2.3.12.

(i) *The set of λ-representable functions is closed under primitive recursion.*

(ii) *If $g : \mathbb{N}^{k+1} \to \mathbb{N}$ is a λ-representable* total *function, then the partial function $\mu[g] : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ is λ-representable.*

(iii) *Every primitive recursive function admits a λ-representation.*

(iv) *Every μ-recursive function admits a λ-representation.*

[1] This requirement has its disadvantages: see [Barendregt, 1984, Ch. 2, § 2].

Proof. (i). Let $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ be a partial function of the form $f = \rho[h; g]$ for some λ-representable functions $g : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^{k+2} \rightharpoonup \mathbb{N}$. Let G and H be λ-representations of g and h, respectively. Define an auxiliary λ-term as below:

$$Z \equiv (H\, x_1 \, \ldots \, x_k \, (\mathbf{pre}\, y)\, (f\, x_1 \, \ldots \, x_k \, (\mathbf{pre}\, y)))$$

We may represent f by the λ-term F below:

$$F \equiv \mathbb{Y}\, [\lambda f\, x_1 \, \ldots \, x_k \, y.\, \textbf{if-then-else}\, (\textbf{is-zero}\, y)\, (G\, x_1 \, \ldots \, x_k)\, Z]$$

Indeed, one may check by induction on y that $\left(F\, \underline{x_1} \cdots \underline{x_k}\, \underline{y}\right) \twoheadrightarrow \underline{f(x_1, \ldots, x_k, y)}$ as required.

(ii). Let $f : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ be a partial function of the form $f = \mu[g]$ for some λ-representable total function $g : \mathbb{N}^{k+1} \to \mathbb{N}$. Let G be a λ-representation of g. Consider the λ-terms below:

$$Y \equiv (G\, x_1 \, \ldots \, x_k \, z)$$
$$T \equiv \mathbb{Y}\, [\lambda t\, z\, x_1 \, \ldots \, x_k \, y.\, \textbf{if-then-else}\, (\textbf{is-equal}\, Y y)\, z\, (t\, (\mathbf{suc}\, z)\, x_1 \, \ldots \, x_k \, y)]$$

Let $F \equiv T\, \underline{0}$. Since $f(x_1, \ldots, x_k, y) = \mu_z[g(x_1, \ldots, x_k, z) = y]$, it is clear that $f(x_1, \ldots, x_k, y)\downarrow$ implies $\left(F\, \underline{x_1} \ldots \underline{x_k}\, \underline{y}\right) \twoheadrightarrow \underline{f(x_1, \ldots, x_k, y)}$, and otherwise the λ-term $\left(F\, \underline{x_1} \ldots \underline{x_k}\, \underline{y}\right)$ is not normalisable.[2] Thus, F is a λ-representation of f.

(iii). Clearly, the constant functions and projection functions admit λ-representations, and the composition of λ-representable functions is certainly λ-representable. The conclusion follows from (i).

(iv). This follows from (ii) and (iii). ■

Remark 2.3.13. By modifying some of the λ-terms appearing in the above proof, one may find λ-terms expressing the λ-calculus versions of the operators $\rho[-;-]$ and $\mu[-]$. In particular, we may define $\rho[H; G]$ and $\mu[F]$ even when F, G, H accept arguments which are not Church numerals.

4 Miscellaneous topics

Although the λ-representability of primitive recursive functions implies that we can encode lists of natural numbers as λ-terms (via Church numerals), we should

[2] This is actually somewhat subtle: see [Barendregt, 1984, Thm 8.3.14, Lem. 8.4.11].

define a "native" encoding which allows us to encode lists of arbitrary λ-terms. A significant advantage of this method is the ability to treat finite and infinite lists on the same basis.

Proposition 2.4.1. *There is an encoding of infinite sequences of λ-terms with the following properties:*

- *There exists a λ-term* **stream**, *such that the λ-term* $(\mathbf{stream}\, F)$ *codes the sequence* $(F\,\underline{0}, F\,\underline{1}, F\,\underline{2}, \ldots)$ *(up to equality of λ-terms).*
- *There exist λ-terms* **is-empty**, **nil**, **cons**, **head**, **tail**,[1] **select**, *and* **replace** *satisfying the specification below.*

Let U and V be λ-terms, and let S code the sequence $(A_0, A_1, A_2, \ldots)$.

(i) *The λ-term* $(\mathbf{is\text{-}empty}\,\mathbf{nil})$ *reduces to* $\underline{\top}$, *while* $(\mathbf{is\text{-}empty}\,(\mathbf{cons}\,UV))$ *and* $(\mathbf{is\text{-}empty}\,S)$ *both reduce to* $\underline{\bot}$.

(ii) *The λ-term* $(\mathbf{cons}\,US)$ *codes the sequence* $(U, A_0, A_1, A_2, \ldots)$.

(iii) *The λ-term* $(\mathbf{head}\,\mathbf{nil})$ *reduces to* **nil**, *while* $(\mathbf{head}\,(\mathbf{cons}\,UV)) \twoheadrightarrow U$ *and* $(\mathbf{head}\,S) \twoheadrightarrow A_0$.

(iv) *The λ-term* $(\mathbf{tail}\,\mathbf{nil})$ *reduces to* **nil**, *while* $(\mathbf{tail}\,(\mathbf{cons}\,UV)) \twoheadrightarrow V$ *and* $(\mathbf{tail}\,S)$ *codes the sequence* $(A_1, A_2, A_3, \ldots)$.

(v) *Given a natural number m,* $(\mathbf{select}\,S\,\underline{m}) \twoheadrightarrow A_m$.

(vi) *Given a natural number m and a λ-term T coding $(B_0, B_1, B_2, \ldots)$, the λ-term* $(\mathbf{replace}\,ST\,\underline{m})$ *codes the sequence* $(A_0, \ldots, A_{m-1}, B_0, B_1, B_2, \ldots)$.

Proof. Like before, we use linked lists. First, make the following definitions of **cons**, **nil** and **is-empty**:[2]

$$\begin{aligned} \mathbf{cons} &\equiv \mathbf{pair} \\ \mathbf{nil} &\equiv [\lambda x.\, \underline{\top}] \\ \mathbf{is\text{-}empty} &\equiv [\lambda x.\, x\,[\lambda yz.\, \underline{\bot}]] \end{aligned}$$

[1] In some circles, **head** and **tail** are traditionally known as **car** and **cdr**, respectively. This **tail** should *not* be interpreted as the analogue of tail defined in theorem 1.3.6.

[2] Caution: This depends heavily on the implementation details of **pair** in proposition 2.3.7.

Now, define λ-terms **head** and **tail** as follows:

$$\begin{aligned}\textbf{head} &\equiv [\lambda \mathtt{x}.\, \textbf{if-then-else}\, (\textbf{is-empty}\, \mathtt{x})\, \textbf{nil}\, (\textbf{left}\, \mathtt{x})] \\ \textbf{tail} &\equiv [\lambda \mathtt{x}.\, \textbf{if-then-else}\, (\textbf{is-empty}\, \mathtt{x})\, \textbf{nil}\, (\textbf{right}\, \mathtt{x})]\end{aligned}$$

The λ-term **select** can be defined using iteration:

$$\textbf{select} \equiv [\lambda \mathtt{xm}.\, \textbf{head}\, (\mathtt{m}\, \textbf{tail}\, \mathtt{x})]$$

On the other hand, the λ-term **replace** is defined by double recursion:

$$\begin{aligned}R &\equiv (\textbf{cons}\, (\textbf{head}\, \mathtt{x})\, (\mathtt{f}\, (\textbf{tail}\, \mathtt{x})\, y\, (\textbf{pre}\, \mathtt{m}))) \\ \textbf{replace} &\equiv \mathbb{Y}\, [\lambda \mathtt{fxym}.\, \textbf{if-then-else}\, (\textbf{is-zero}\, \mathtt{r})\, \mathtt{y}R]\end{aligned}$$

One may then check that the following definition of **stream** works:

$$\textbf{stream} \equiv (\mathbb{Y}\, [\lambda \mathtt{smf}.\, \textbf{cons}\, (\mathtt{fm})\, (\mathtt{s}\, (\textbf{suc}\, \mathtt{m})\, \mathtt{f})]\, \underline{0}) \qquad \blacksquare$$

Theorem 2.4.2. *There is an encoding of finite lists of λ-terms with the following properties:*

- *There exists a λ-term* $\textbf{list}_n$ *for each natural number* n*, such that* $\textbf{list}_0 \equiv \textbf{nil}$ *and the λ-term* $(\textbf{list}_n\, A_0 \ldots A_{n-1})$ *codes the list* $(A_0, \ldots, A_{n-1})$*.*
- *If a λ-term* L *codes the finite list* $(A_0, \ldots, A_{n-1})$*, then* L *also codes the sequence* $(A_0, \ldots, A_{n-1}, \textbf{nil}, \textbf{nil}, \textbf{nil}, \ldots)$*.*
- *There exist λ-terms* **length**, **insert**, *and* **slice** *satisfying the specification below.*

Let $L \equiv (\textbf{list}_n\, A_0 \ldots A_{n-1})$*, and let* S *code the sequence* $(B_0, B_1, B_2, \ldots)$*.*

(i) *The λ-term* $(\textbf{is-empty}\, L)$ *reduces to* $\underline{\perp}$ *if* $n > 0$ *and* $(\textbf{is-empty}\, L)$ *reduces to* $\underline{\top}$ *if* $n = 0$*.*

(ii) *The λ-term* $(\textbf{length}\, L)$ *reduces to* $\underline{n}$*.*

(iii) *If* $m < n$ *then* $(\textbf{select}\, L\, \underline{m}) \twoheadrightarrow A_m$*, and otherwise* $(\textbf{select}\, L\, \underline{m}) \twoheadrightarrow \textbf{nil}$*.*

(iv) *The λ-term* $(\textbf{insert}\, LU)$ *codes the list* $(A_0, \ldots, A_{n-1}, U)$*.*

(v) *Given a natural number* m*, the λ-term* $(\textbf{replace}\, SL\, \underline{m})$ *codes the finite list* $(B_0, \ldots, B_{m-1}, A_0, \ldots, A_{n-1})$*.*

(vi) *If* $\mathtt{r}$ *and* $\mathtt{s}$ *are natural numbers and* $\mathtt{r} < \mathtt{s}$, *then* $(\mathbf{slice}\,\mathtt{S}\,\underline{\mathtt{r}}\,\underline{\mathtt{s}})$ *codes the list* $(\mathtt{B}_{\mathtt{r}}, \ldots, \mathtt{B}_{\mathtt{s}-1})$, *and otherwise* $(\mathbf{slice}\,\underline{\mathtt{r}}\,\underline{\mathtt{s}}) \twoheadrightarrow \mathbf{nil}$.

Proof. We use the λ-terms defined in the previous proposition to define $\mathbf{list}_n$ for all natural numbers n as below:

$$\begin{aligned}
\mathbf{list}_0 &\equiv \mathbf{nil}\\
\mathbf{list}_1 &\equiv [\lambda \mathtt{a}_0.\, \mathbf{cons}\,\mathtt{a}_0\,\mathbf{nil}]\\
&\ \vdots\\
\mathbf{list}_n &\equiv [\lambda \mathtt{a}_0\ \ldots\ \mathtt{a}_{n-1}.\, \mathbf{cons}\,\mathtt{a}_0\,(\mathbf{cons}\cdots(\mathbf{cons}\,\mathtt{a}_{n-1}\,\mathbf{nil})\cdots)]\\
&\ \vdots
\end{aligned}$$

By the specification of **is-empty**, **nil**, and **cons**, we see that (i) holds, and by examining the details of how **select** and **replace** were defined in the proof above, we see that (iii) and (v) are satisfied.

We use structural recursion on lists and an accumulator to define **length**:

$$\mathbf{length} \equiv (\mathbb{Y}\,[\lambda \mathtt{crx}.\, \mathbf{if\text{-}then\text{-}else}\,(\mathbf{is\text{-}empty}\,\mathtt{x})\,\mathtt{r}\,(\mathtt{c}\,(\mathbf{suc}\,\mathtt{r})\,(\mathbf{tail}\,\mathtt{x}))]\,\underline{0})$$

We also define the λ-term **insert** by structural recursion on lists:

$$\begin{aligned}
\mathrm{I} &\equiv (\mathbf{cons}\,(\mathbf{head}\,\mathtt{x})\,(\mathtt{f}\,(\mathbf{tail}\,\mathtt{x})\,\mathtt{b}))\\
\mathbf{insert} &\equiv \mathbb{Y}\,[\lambda \mathtt{fxb}.\, \mathbf{if\text{-}then\text{-}else}\,(\mathbf{is\text{-}empty}\,\mathtt{x})\,(\mathbf{cons}\,\mathtt{b}\,\mathbf{nil})\,\mathrm{I}]
\end{aligned}$$

Finally, we use **replace** and iterate **tail** to define **slice**:

$$\mathbf{slice} \equiv [\lambda \mathtt{xrs}.\, \mathtt{r}\,\mathbf{tail}\,(\mathbf{replace}\,\mathtt{x}\,\mathbf{nil}\,\mathtt{s})]$$ ■

Recall the combinators $\mathbb{S}$, $\mathbb{K}$, and $\mathbb{I}$:

$$\mathbb{S} \equiv [\lambda \mathtt{xyz}.\, \mathtt{xz}\,(\mathtt{yz})] \qquad \mathbb{K} \equiv [\lambda \mathtt{xy}.\, \mathtt{x}] \qquad \mathbb{I} \equiv [\lambda \mathtt{x}.\, \mathtt{x}]$$

Theorem 2.4.3. *If* $\mathtt{T}$ *is a λ-term, then there is a λ-term* $\mathtt{T}'$ *constructed using only the free variables of* $\mathtt{T}$ *and the combinators* $\mathbb{S}, \mathbb{K}, \mathbb{I}$ *and no other instances of λ-abstraction such that* $\mathtt{T}' \twoheadrightarrow \mathtt{T}$. *We call such a λ-term an* **abstraction elimination** *of* $\mathtt{T}$.

Proof. If $\mathsf{T} \equiv \mathsf{x}$, then T is already in the required form, so we may take $\mathsf{T}' \equiv \mathsf{T}$ in this case. If $\mathsf{T} \equiv (\mathsf{UV})$ and U' and V' are abstraction eliminations of U and V, respectively, then

$$\mathsf{T}' \equiv (\mathsf{U}'\mathsf{V}') \twoheadrightarrow (\mathsf{U}\mathsf{V}') \twoheadrightarrow (\mathsf{UV}) \equiv \mathsf{T}$$

as required. If $\mathsf{T} \equiv [\lambda \mathsf{x}.\, W]$ and $\mathsf{x} \notin \mathrm{FV}(W)$, then set $\mathsf{T}' \equiv (\mathbb{K}W')$, where W' is an abstraction elimination of W; otherwise we proceed by structural induction:

- If $W \equiv \mathsf{x}$, then set $\mathsf{T}' \equiv \mathbb{I}$.
- If $\mathsf{x} \in \mathrm{FV}(W)$ and $W \equiv (\mathsf{UV})$, then set $\mathsf{T}' \equiv ((\mathbb{S}\mathsf{U}')\,\mathsf{V}')$, where U' and V' are the abstraction eliminations of $[\lambda \mathsf{x}.\, \mathsf{U}]$ and $[\lambda \mathsf{x}.\, \mathsf{V}]$, respectively.
- If $\mathsf{x} \in \mathrm{FV}(W)$, $W \equiv [\lambda \mathsf{y}.\cdots]$, and W' is an abstraction elimination of W, then $\mathrm{BV}(W') = \varnothing$ and so we can run the above recursion on $[\lambda \mathsf{x}.\, W']$ to obtain T' without encountering this case again. ■

Recalling that $(\mathbb{SKK}) \twoheadrightarrow \mathbb{I}$, we obtain the following easy corollary:

Corollary 2.4.4. *Any λ-term admits an abstraction elimination using only the combinators $\mathbb{S}$ and $\mathbb{K}$.* ■

Note, however, the abstraction eliminations are *not* unique. For example, one may verify that $(\mathbb{SKS}) \twoheadrightarrow \mathbb{I}$, and indeed, for any λ-term T whatsoever, $(\mathbb{SK}\mathsf{T}) \twoheadrightarrow \mathbb{I}$.

Example 2.4.5. Recall that **suc** $= [\lambda \mathtt{nfx}.\, \mathtt{f}\,(\mathtt{nfx})]$. Using the recursion described above, we obtain the following abstraction eliminations:

$$\begin{gathered}
(\mathbb{S}\,(\mathbb{K}\mathtt{f})\,(\mathbb{K}\,(\mathtt{nf}))) \twoheadrightarrow [\lambda \mathtt{x}.\, \mathtt{f}\,(\mathtt{nfx})] \\
(\mathbb{S}\,(\mathbb{KS})\,\mathbb{K}) \twoheadrightarrow [\lambda \mathtt{f}.\, \mathbb{S}\,(\mathbb{K}\mathtt{f})] \\
(\mathbb{S}\,(\mathbb{KK})\,\mathtt{n}) \twoheadrightarrow [\lambda \mathtt{f}.\, \mathbb{K}\,(\mathtt{nf})] \\
(\mathbb{S}\,(\mathbb{S}\,(\mathbb{KS})\,\mathbb{K})\,(\mathbb{S}\,(\mathbb{KK})\,\mathtt{n})) \twoheadrightarrow [\lambda \mathtt{f}.\, \mathbb{S}\,(\mathbb{K}\mathtt{f})\,(\mathbb{K}\,(\mathtt{nf}))] \\
(\mathbb{S}\,(\mathbb{K}\,(\mathbb{S}\,(\mathbb{S}\,(\mathbb{KS})\,\mathbb{K})))\,(\mathbb{S}\,(\mathbb{KK}))) \twoheadrightarrow [\lambda \mathtt{n}.\, \mathbb{S}\,(\mathbb{S}\,(\mathbb{KS})\,\mathbb{K})\,(\mathbb{S}\,(\mathbb{KK})\,\mathtt{n})] \twoheadrightarrow \mathbf{suc}
\end{gathered}$$

Remark 2.4.6. There are other choices of combinators which could be used to achieve abstraction elimination. For example, one could use the $\mathbb{BCKW}$ system of Curry [1930a,b]:

$$\begin{aligned}
\mathbb{B} &\equiv [\lambda \mathsf{xyz}.\, \mathsf{x}\,(\mathsf{yz})] & \qquad \mathbb{C} &\equiv [\lambda \mathsf{xyz}.\, \mathsf{xzy}] \\
\mathbb{K} &\equiv [\lambda \mathsf{xy}.\, \mathsf{x}] & \qquad \mathbb{W} &\equiv [\lambda \mathsf{xy}.\, \mathsf{xyy}]
\end{aligned}$$

— III —

Machines

Computation is supposed to be a mechanical process, yet so far our two proposed formalisations of effective computability do not explicitly invoke the notion of a machine. There are many different definitions of abstract machine but, so far, all the proposed definitions of abstract *finitistic* machines turn out to have exactly the same power as μ-recursive functions and λ-calculus. The **Church–Turing thesis** asserts that this will be the case for *any* attempt to formalise effective computability.[1]

1 Turing machines

Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

Alan Perlis, *Epigrams on Programming*

Definition 3.1.1. A **Turing machine** $\mathfrak{M}$ comprises the following data:

- A non-empty finite set Q, called the **states** of $\mathfrak{M}$.
- A non-empty finite set Γ, called the **alphabet** of $\mathfrak{M}$.
- A distinguished symbol $\mathfrak{b}$ in Γ, called the **blank symbol**.

[1] The Church–Turing thesis is an informal claim and cannot be formally proven *per se*.

- A distinguished state in Q: the **initial state** q_0.
- A subset of Q: the **accepting states** F.
- A partial function $\delta : (Q \setminus F) \times \Gamma \rightharpoonup Q \times \Gamma \times \{-1, 0, +1\}$, called the **transition function**.

Strictly speaking, we require Q and Γ to be the smallest sets such that $b \in \Gamma$ and the type declaration for δ is valid, i.e.

- For each q in Q, there is some a in Γ such that either $\delta(q, a)\downarrow$, or there are some q' in Q and a' in Γ such that $\delta(q', a')\downarrow$ and $\delta(q', a') = (q, a)$.
- For each a in Γ, either $a = b$, or there is some q in Q such that $\delta(q, a)\downarrow$, or there are q and q' in Q and a' in γ such that $\delta(q', a')\downarrow$ and $\delta(q', a') = (q, a)$.

Definition 3.1.2. A **tape** $\bar{a}$ for a Turing machine $\mathfrak{M}$ is an integer-indexed list $(a_i \mid i \in \mathbb{Z})$ such that

- each a_i is in the alphabet Γ of $\mathfrak{M}$, and
- the set $\{i \in \mathbb{Z} \mid a_i \neq b\}$ is finite, where b is the blank symbol.

Let $\mathcal{L}$ be the set of all tapes for a Turing machine $\mathfrak{M}$. The transition function δ induces a partial function $\sigma : Q \times \mathcal{L} \rightharpoonup Q \times \mathcal{L}$ as follows:

- If $q \in F$, then $\sigma(q, \bar{a}) = (q, \bar{a})$.
- If $\delta(q, a_0)\uparrow$, then $\sigma(q, \bar{a})\uparrow$ as well.
- Otherwise, if $\delta(q, a_0) = (q', a', u)$, then $\sigma(q, \bar{a}) = (q', \bar{a}')$ where

$$a'_i = \begin{cases} a' & \text{if } i = -u \\ a_{i+u} & \text{if } i \neq -u \end{cases}$$

It is clear that we can implement σ as a *primitive* recursive function for any fixed Turing machine $\mathfrak{M}$, at least once we have fixed an encoding of Q and $\mathcal{L}$ and turned σ into a *total* function by adjoining a distinguished element $\perp$ to the domain and codomain to represent undefined function values. Now, define a partial function $\sigma^* : \mathcal{L} \times \mathbb{N} \rightharpoonup Q \times \mathcal{L}$ by primitive recursion:

$$\sigma^*(\bar{a}, 0) = (q_0, \bar{a})$$
$$\sigma^*(\bar{a}, n+1) = \sigma(\sigma^*(\bar{a}, n))$$

(As usual, what this means is that the LHS is defined and equal to the RHS whenever the RHS is defined, and the LHS is undefined when the RHS is undefined.) Again, σ^* is primitive recursive once it has been appropriately modified. Thus, we obtain a μ-recursive partial function $\sigma^\infty : \mathcal{L} \rightharpoonup Q \times \mathcal{L}$ by unbounded minimisation:

$$\sigma^\infty(\bar{a}) = \sigma^*(\bar{a}, \mu_n[\sigma^*(\bar{a}, n)\downarrow \text{ and } \sigma^*(\bar{a}, n) \in F \times \mathcal{L}])$$

What is remarkable is that we can do all this in a uniform way *even when $\mathfrak{M}$ is allowed to vary*. Indeed, without loss of generality, we may demand that the following hold for every Turing machine $\mathfrak{M}$:

- Both the state space $Q_\mathfrak{M}$ and the alphabet $\Gamma_\mathfrak{M}$ are finite subsets of $\mathbb{N}$.
- The blank symbol $b_\mathfrak{M}$ is 0.

Under this assumption, the set of all Turing machines is countable, and we may uniformly encode tapes as finite lists of natural numbers indexed by integers.

Theorem 3.1.3 (Kleene). *There exists an enumeration of the set of all Turing machines and a primitive recursive function $E : \mathbb{N}^3 \to \mathbb{N}$ with the following properties:*

- *The number $E(m, s, t)$ codes a finite list of length less than t.*
- *If m codes the Turing machine $\mathfrak{M}$ and s codes the tape $\bar{a}$, then*

  $$E(m, s, t) \textit{ codes the list } (\sigma^*_\mathfrak{M}(\bar{a}, 0), \ldots, \sigma^*_\mathfrak{M}(\bar{a}, \ell))$$

  where ℓ is the greatest number less than t such that $\sigma^*_\mathfrak{M}(\bar{a}, \ell)\downarrow$. (If s codes a tape $\bar{a}$ which contains symbols not in the alphabet $\Gamma_\mathfrak{M}$, then $E(m, s, t)$ codes the empty list.)

In particular, the function $T : \mathbb{N}^3 \to \mathbb{N}$ defined below is primitive recursive:

$$T(m, s, h) = \begin{cases} 1 & \textit{if } h = E(m, s, \mathsf{length}(h) + 1) \\ 0 & \textit{otherwise} \end{cases}$$

The predicate represented by T is called **Kleene's T-predicate**.

Proof. This is clear, since everything in sight is finite and easily bounded. ∎

By theorem 2.3.12, Kleene's T-predicate is λ-representable, but it is not hard to directly prove a version of the above theorem using the "native" λ-calculus encoding of sequences given in proposition 2.4.1. We also have a converse:

Theorem 3.1.4 (Turing). *There exists a Turing machine $\mathfrak{M}$ and an encoding of λ-terms with variables drawn from a fixed countably infinite alphabet Δ as tapes for $\mathfrak{M}$ with the following properties:*

- *$\mathfrak{M}$ has a unique accepting state $\top$.*
- *If the tape $\bar{a}$ codes a λ-term T and every λ-subterm U of T has the property $\mathrm{BV}(U) \cap \mathrm{FV}(U) = \varnothing$, then $\sigma^\infty(\bar{a})\downarrow$ if and only if T is normalisable, and in that case $\sigma^\infty(\bar{a}) = (\top, \bar{a}')$ where $\bar{a}'$ codes the normal form of T (up to α-conversion).*

Proof. We fix an enumeration of Δ, so that $\Delta = \{x_1, x_2, x_3, \ldots\}$. The alphabet Γ of our Turing machine $\mathfrak{M}$ consists of the following symbols:

$$b \quad x \quad ' \quad \beta \quad \backslash$$

Additional symbols may be used to record working state on the tape. For clarity, we will enclose strings with guillemets, e.g. $\langle\!\langle \backslash x' x' \rangle\!\rangle$. We encode λ-terms using prefix notation:

- If $T \equiv x_n$, then we encode T as a string of the form $\langle\!\langle x'^{\cdots\prime} \rangle\!\rangle$ with n prime symbols.
- If $T \equiv (UV)$, then we encode T as $\langle\!\langle \beta \rangle\!\rangle$ followed by the encoding of U and the encoding of V.
- If $T \equiv [\lambda x_n . U]$, then we encode T as $\langle\!\langle \backslash \rangle\!\rangle$ followed by the encoding of x_n and the encoding of U.

For example, the $\mathbb{S}$ combinator is encoded as $\langle\!\langle \backslash x' \backslash x'' \backslash x''' \beta \beta x' x''' \beta x'' x''' \rangle\!\rangle$. It is easy to check that this encodes λ-terms unambiguously and has the advantage of not requiring brackets (but also the disadvantage of being hard to read).

It was earlier stated that a normalisable λ-term can always be reduced to normal form by iteratively β-reducing the leftmost outermost λ-subterms of the form (UV) and then iteratively η-reducing the result.

This is doable by a Turing machine: we use the negatively-indexed section of the tape as working memory to record the name of the variable we are replacing and the λ-term we are substituting, and we use the section of the tape after the end of the input to record the partial output. First, we do the β-reductions:

1. Scan the input for the leftmost occurrence of $\langle\!\langle \beta\backslash x \rangle\!\rangle$. If there are no occurrences, return to the beginning of the input and proceed to the η-reduction step.

2. Copy the name of the variable to be replaced to working memory.

3. Copy the substituent term to working memory, and replace the substituent in the input with $\langle\!\langle ?\ldots? \rangle\!\rangle$. Note that we must count β symbols in order to correctly determine the start and end of the substituent. Since we are not yet outputting anything, we may use the section after the tape to count β symbols.

4. Copy to output the input up to the leftmost occurrence of $\langle\!\langle \beta\backslash x \rangle\!\rangle$, and skip the $\langle\!\langle \beta\backslash x^{\prime\cdots\prime} \rangle\!\rangle$.

5. Copy to output the input up to the leftmost occurrence of $\langle\!\langle ? \rangle\!\rangle$, taking care to replace each occurrence of the variable to be replaced in the input with the substituent in the output.

6. Skip the $\langle\!\langle ?\ldots? \rangle\!\rangle$, and copy the remainder of the input to output.

7. Erase the working memory and input, and go to the beginning of the output.

8. Repeat, regarding the current output as the new input.

Because we are assuming that every λ-subterm U of the input T has the property $BV(U) \cap FV(U) = \varnothing$, this correctly computes the β-reduction. Now, we do the η-reductions:

1. Scan the input for the leftmost occurrence of $\langle\!\langle \backslash x^{\prime\cdots\prime}\beta U x^{\prime\cdots\prime} \rangle\!\rangle$, where U denote a well-formed subterm. Note that we must count both β symbols and prime symbols. If there are no occurrences, return to the beginning of the input and halt in an accepting state.

2. Replace the substrings $\langle\!\langle \backslash x^{\prime\cdots\prime}\beta \rangle\!\rangle$ and $\langle\!\langle x^{\prime\cdots\prime} \rangle\!\rangle$ found above with $\langle\!\langle ?\ldots? \rangle\!\rangle$.

3. Copy the input to output, skipping the occurrences of $\langle\!\langle ? \ldots ? \rangle\!\rangle$.

4. Erase the working memory and input, and go to the beginning of the output.

5. Repeat, regarding the current output as the new input.

One may verify that this algorithm can in principle be implemented as a Turing machine. ∎

Now, let $\mathcal{L}$ be the set of all tapes written in a fixed alphabet Γ. For each natural number k, choose an injective total function $\mathsf{in}_k : \mathbb{N}^k \to \mathcal{L}$, and also choose an injective total function $\mathsf{out} : \mathbb{N} \to \mathcal{L}$.

Definition 3.1.5. Let $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ be a partial function. A Turing machine $\mathfrak{M}$ **computes** f just if $\mathfrak{M}$ satisfies the following conditions:

- Γ is a subset of the alphabet of $\mathfrak{M}$.
- $\mathfrak{M}$ has a unique accepting state $\top$.
- For all $\vec{x}$ in $\mathbb{N}^k$, $\sigma^\infty(\mathsf{in}_k(\vec{x}))\downarrow$ if and only if $f(\vec{x})\downarrow$, and when $f(\vec{x})\downarrow$, we have $\sigma^\infty(\mathsf{in}_k(\vec{x})) = (\top, \mathsf{out}(f(\vec{x})))$.

A **Turing-computable function** is a partial function f which admits such a Turing machine $\mathfrak{M}$.

Theorem 3.1.6 (Church–Turing). *Let $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ be a partial function. The following are equivalent:*

(i) *There exists a μ-recursive declaration of f.*

(ii) *There exists a λ-representation of f.*

(iii) *There exists a Turing machine that computes f.*

Proof. (i) ⇒ (ii). This is theorem 2.3.12.
(ii) ⇒ (iii). This is Turing's theorem (3.1.4).
(iii) ⇒ (i). This is a corollary of Kleene's theorem (3.1.3). ∎

We obtain the following strengthening of theorem 1.4.10 as a corollary:

Corollary 3.1.7 (Kleene normal form). *There exists a primitive recursive function* $V : \mathbb{N}^3 \to \mathbb{N}$ *with the following property: for every μ-recursive function* $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$, *there exists a natural number* m *such that*

$$f(\vec{x}) = \mathsf{right}(\boldsymbol{\mu}_z[V(m, \mathsf{list}_k(\vec{x}), z) = 1])$$

for all $\vec{x}$ *in* $\operatorname{dom} f$, *and* $\boldsymbol{\mu}_z[V(m, \mathsf{list}_k(\vec{x}), z) = 1]\uparrow$ *if and only if* $f(\vec{x})\uparrow$.

Proof. One simply modifies Kleene's T-predicate to obtain the desired primitive recursive function V. ■

Corollary 3.1.8 (Universal Turing machine). *There is a Turing machine that can simulate any Turing machine.* ■

Corollary 3.1.9. *Every μ-recursive partial function* $f : \mathbb{N} \rightharpoonup \mathbb{N}$ *admits a μ-recursive partial right inverse, i.e. a partial function* $g : \mathbb{N} \rightharpoonup \mathbb{N}$ *such that the following holds:*

$$y \in \operatorname{im} f \textit{ implies } g(y)\downarrow \textit{ and } f(g(y)) = y$$

Proof. The obvious method using $\boldsymbol{\mu}_x[f(x) = y]$ does not work, since f is not necessarily a total function. Instead, we must use Kleene's T-predicate to construct a primitive recursive predicate $\varphi(x, y, z)$ with free variables x, y, z such that

$$\exists z.\ \varphi(x, y, z) \text{ holds if and only if } f(x) = y$$

This can then be used to define g:

$$g(y) = \mathsf{right}(\boldsymbol{\mu}_z[\varphi(\mathsf{right}(z), y, \mathsf{left}(z)) = 1])$$ ■

Recall the primitive recursive function $V : \mathbb{N}^3 \to \mathbb{N}$ of corollary 3.1.7. Let $F_k : \mathbb{N}^{1+k} \rightharpoonup \mathbb{N}$ be the μ-recursive partial function given below:

$$F(m, \vec{x}) = \mathsf{right}(\boldsymbol{\mu}_z[V(m, \mathsf{list}_k(\vec{x}), z) = 1])$$

Then, for every μ-recursive partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ there is a natural number r such that $f = F_k(r, -)$, so we may call F_k the **universal** k**-ary μ-recursive function**.

Theorem 3.1.10 (Kleene's $s_{m,n}$ theorem). *There is a primitive recursive function* $s_{m,n} : \mathbb{N}^{1+m} \to \mathbb{N}$ *with the following property: for all natural numbers* r *and all* m*-tuples* $\vec{x}$,

$$F_{m+n}(r, \vec{x}, -) = F_n(s_{m,n}(r, \vec{x}), -)$$

We call $s_{m,n}$ *a* **partial evaluation function**.

Proof. Strictly speaking this depends on how we define the enumeration of k-ary μ-recursive functions, but in principle any enumeration that goes via λ-calculus (as ours does) will work. Indeed, under any reasonable enumeration $\#$ of λ-terms (such as the one in theorem 3.1.4), the function

$$(\#(T), x_1, \ldots, x_m) \mapsto \#(T\, \underline{x_m} \ldots \underline{x_n})$$

will be primitive recursive, and this is exactly what we need. ∎

Remark. The fact that there is an effectively computable partial evaluation function is not really a surprise when phrased in terms of programming: all it says is that, given the source code of a program which computes a function f of $m+n$ variables and instances $x_1, \ldots, x_m$ of the first m variables, there is an automatic process by which one can obtain the source code of a program which computes the function $f(x_1, \ldots, x_m, -, \ldots, -)$; but this is entirely obvious: just search and replace!

What would be interesting, however, is an *optimising* partial evaluator, i.e. a partial evaluation function which returns the source code of a *more efficient* program that computes $f(\vec{x}, -)$. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two programming languages,[1] and let r_0 be the $\mathcal{L}_1$-code of an interpreter for $\mathcal{L}_2$.[2] Given an optimising partial evaluator $s_{1,1}$ for $\mathcal{L}_1$, we could do the following:

1. Compute $s_{1,1}(r_0, r)$ to obtain a $\mathcal{L}_1$-code for an *efficient* program equivalent to the program described by the $\mathcal{L}_2$-code r, i.e. an executable for r.
2. Compute $s_{1,1}(r_1, r_0)$, where r_1 is a $\mathcal{L}_1$-code for $s_{1,1}$, to obtain a $\mathcal{L}_1$-code that translates $\mathcal{L}_2$-codes into equivalent $\mathcal{L}_1$-codes, i.e. a $\mathcal{L}_1$-compiler for $\mathcal{L}_2$.
3. Compute $s_{1,1}(r_1, r_1)$ to obtain a $\mathcal{L}_1$-code for a program that transforms interpreters into compilers!

These are called the **1st, 2nd, and 3rd Futamura projections**, respectively.

Recall the $\mathbb{Y}$ combinator of λ-calculus:

$$\mathbb{Y} = [\boldsymbol{\lambda} \mathtt{h}.\, ([\boldsymbol{\lambda} \mathtt{x}.\, \mathtt{h}\,(\mathtt{xx})]\, [\boldsymbol{\lambda} \mathtt{x}.\, \mathtt{h}\,(\mathtt{xx})])]$$

[1] i.e. two different effective enumerations of computable functions.
[2] i.e. the universal unary partial computable function F_1 for the second enumeration.

It is sometimes called a **fixed point combinator** because it produces solutions for the following equation of λ-terms:

$$F = (HF)$$

Indeed, one may check that $F \equiv (\mathbb{Y}H)$ has the required property. Using the partial evaluation function $s_{1,1}$, we may do something similar with μ-recursive functions:

Corollary 3.1.11 (Fixed point theorem). *If $h : \mathbb{N} \rightharpoonup \mathbb{N}$ is a μ-recursive partial function, then there is a natural number n solving the functional equation below:*

$$F_1(n, -) = F_1(h(n), -)$$

(Caution: $h(n)$ may be undefined, in which case $F_1(n, x)\uparrow$ for all x.)

Proof. Let $g : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ be the partial function defined by

$$g(x, y) = F_1(h(s_{1,1}(x, x)), y)$$

This is μ-recursive because F_1, $s_{1,1}$, and h are, and so $g = F_2(m, -, -)$ for some natural number m. Let $n = s_{1,1}(m, m)$. Then,

$$\begin{aligned}
F_1(n, -) &= F_1(s_{1,1}(m, m), -) && \text{by definition of } n \\
&= F_2(m, m, -) && \text{by definition of } s_{1,1} \\
&= g(m, -) && \text{by definition of } m \\
&= F_1(h(s_{1,1}(m, m)), -) && \text{by definition of } g \\
&= F_1(h(n), -) && \text{by definition of } n
\end{aligned}$$ ■

Remark 3.1.12. A careful examination of the above proof reveals another fixed point combinator for λ-calculus:

$$\hat{\mathbb{Y}} = [\lambda h.\, ([\lambda xy.\, h\,(xx)\,y]\,[\lambda xy.\, h\,(xx)\,y])]$$

Notice that by η-reducing the innermost λ-abstractions, $\hat{\mathbb{Y}} \twoheadrightarrow \mathbb{Y}$.

2 Decidable and semidecidable sets

We will soon see how useful the machine formalism is.

Definition 3.2.1. A **decidable set** is a subset X of $\mathbb{N}$ such that its characteristic function $\chi_X : \mathbb{N} \to \{0, 1\}$, given by

$$\chi_X(x) = \begin{cases} 0 & \text{if } x \notin X \\ 1 & \text{if } x \in X \end{cases}$$

is μ-recursive (or, equivalently, λ-representable or Turing-computable).

Clearly, every finite set is decidable, and the extension of any μ-recursive predicate is decidable (by definition). What is not so clear is whether, say, the image of a μ-recursive *partial* function is decidable. In fact, it is not decidable in general.

Definition 3.2.2. A **recursively enumerable set** is a *non-empty* subset X of $\mathbb{N}$ such that there is a μ-recursive total function $g : \mathbb{N} \to \mathbb{N}$ whose image is precisely X.

Definition 3.2.3. A **semidecidable set** is a subset X of $\mathbb{N}$ with the following property: there exists a μ-recursive partial function $f : \mathbb{N} \rightharpoondown \mathbb{N}$ such that $f(x){\downarrow}$ *and* $h(x) = 1$ if and only if $x \in X$. (If $x \notin X$, then either $h(x){\uparrow}$ or $f(x) \neq 1$.)

Proposition 3.2.4. *Let* X *be a* non-empty *subset of* $\mathbb{N}$. *The following are equivalent:*

(i) X *is recursively enumerable.*

(ii) $X = \operatorname{im} g$ *for some μ-recursive partial function* $g : \mathbb{N} \rightharpoondown \mathbb{N}$.

(iii) $X = \operatorname{dom} f$ *for some μ-recursive partial function* $f : \mathbb{N} \rightharpoondown \mathbb{N}$.

(iv) X *is semidecidable.*

(v) X *is the image of a primitive recursive function* $g : \mathbb{N} \to \mathbb{N}$.

Proof. (i) $\Rightarrow$ (ii). Obvious.

(ii) $\Rightarrow$ (iii). Let f be any computable right inverse for g: this exists by corollary 3.1.9, and $x \in \operatorname{im} g$ if and only if there is y such that $g(y) = x$, but this happens if and only if $x \in \operatorname{dom} f$.

(iii) $\Rightarrow$ (iv). Let $k : \mathbb{N} \to \mathbb{N}$ be the constant function given by $k(y) = 1$, and let $h = k \circ f$. One may check that h is the function required in the definition of semidecidability.

(iv) $\Rightarrow$ (v). Since X is assumed non-empty, it has a least member x_0. By modifying Kleene's T-predicate (theorem 3.1.3), we may obtain a primitive recursive predicate $\varphi(x, z)$ with two free variables x and z such that

$$\exists z.\, \varphi(x, z) \text{ holds if and only if } h(x)\downarrow \text{ and } h(x) = 1$$

and the function $g : \mathbb{N} \to \mathbb{N}$ defined below using bounded quantification is primitive recursive, by :

$$g(y) = \begin{cases} x & \text{if } \exists z < y.\, \exists x < y.\, \varphi(x, z) \\ x_0 & \text{if } \forall z < y.\, \forall x < y.\, \neg\varphi(x, z) \end{cases}$$

By construction, the image of g is precisely X, as required.

(v) $\Rightarrow$ (i). Obvious. ■

Proposition 3.2.5. *Let* X *be a subset of* $\mathbb{N}$. *The following are equivalent:*

(i) X *is decidable.*

(ii) $\mathbb{N} \setminus X$ *is decidable.*

(iii) X *and* $\mathbb{N} \setminus X$ *are both semidecidable.*

Proof. (i) $\Leftrightarrow$ (ii). Bounded subtraction is primitive recursive, so we can easily turn the characteristic function of X into the characteristic function of $\mathbb{N} \setminus X$ and vice-versa.

(i) and (ii) $\Rightarrow$ (iii). Obvious.

(iii) $\Rightarrow$ (i). Let $h, h' : \mathbb{N} \rightharpoondown \mathbb{N}$ be the partial characteristic functions of X and $\mathbb{N} \setminus X$, respectively. By using two instances of Kleene's T-predicate, we may construct a μ-recursive *total* function $\chi_X : \mathbb{N} \to \mathbb{N}$ such that

$$\chi(x) = \begin{cases} 1 & \text{if } (h(x)\downarrow \wedge h(x) = 1) \vee (h'(x)\downarrow \wedge h'(x) \neq 1) \\ 0 & \text{otherwise} \end{cases}$$

and this is precisely the characteristic function of X. ■

Proposition 3.2.6.

(i) *If* $f : \mathbb{N} \to \mathbb{N}$ *is a μ-recursive total function and* X *is a decidable (resp. semidecidable) set, then* $f^{-1}X$ *is a decidable (resp. semidecidable) set.*

(ii) *If* X *and* Y *are decidable (resp. semidecidable) sets, so both* $X \cap Y$ *and* $X \cup Y$ *are decidable (resp. semidecidable).*

(iii) *If* $f : \mathbb{N} \rightharpoonup \mathbb{N}$ *is a μ-recursive partial function and* X *is a semidecidable set, then the image of* X *under* f *is semidecidable.*

(iv) *If* $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ *is a μ-recursive partial (resp. total) function, then the set* $\Gamma = \{\mathsf{list}_{k+1}(\vec{x}, f(\vec{x})) \mid \vec{x} \in \operatorname{dom} f\}$ *is semidecidable (resp. decidable).*

Proof. (i). The composition of μ-recursive functions is μ-recursive.

(ii). Use proposition 1.2.4.

(iii). If X is empty, then its image is empty and is semidecidable; otherwise, X is recursively enumerable by the previous proposition, so its image is also recursively enumerable.

(iv). The set Γ is the image of a μ-recursive function, so it is certainly semidecidable. If f is known to be total, then the characteristic function χ_Γ admits an evident μ-recursive declaration. ■

More generally, we may talk about decidable or semidecidable subsets of any countable set U once we have fixed a bijection of U with a decidable subset of $\mathbb{N}$. In particular, we can use the primitive recursive functions list_k of theorem 1.3.6 to encode k-tuples of natural numbers as single natural numbers, so we may talk about decidable and semidecidable subsets of $\mathbb{N}^k$ for every positive integer k.

Proposition 3.2.7. *Let* X *be a subset of* $\mathbb{N}^k$*. The following are equivalent:*

1. X *is semidecidable.*
2. *There is a decidable subset* Y *of* $\mathbb{N}^{k+1}$ *such that* X *is the image of* Y *under the projection function* $(a_0, \dots, a_k) \mapsto (a_0, \dots, a_{k-1})$.

Proof. (i) ⇒ (ii). Let $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ be a μ-recursive partial characteristic function for X. Using Kleene's T-predicate (theorem 3.1.3), we may construct a primitive recursive predicate $\varphi(\vec{x}, y)$ with free variables $\vec{x}$ and y such that $\varphi(\vec{x}, y)$ holds if and only if y codes a terminating computation of $f(\vec{x})$ with $f(\vec{x}) = 1$. The extension Y of this predicate is the desired decidable subset of $\mathbb{N}^{k+1}$.

(ii) ⇒ (i). The image of a decidable set is semidecidable. ■

Proposition 3.2.8. *Let* $f : \mathbb{N}^k \to \mathbb{N}$ *be a μ-recursive total function. If* f *is non-decreasing in the sense that* $f(\vec{x}) \geqslant \max \vec{x}$ *for all* k*-tuples* $\vec{x}$*, then the image under* f *of any decidable subset of* $\mathbb{N}^k$ *is decidable.*

Proof. Let $y \in \mathbb{N}$, and let X be a decidable subset of $\mathbb{N}^k$. By hypothesis, if $f(\vec{x}) = y$, then $\max \vec{x} \leqslant y$, so there are only finitely many k-tuples we must check in order to determine whether y is in $\operatorname{im} f|_X$. It is then clear that the characteristic function of $\operatorname{im} f|_X$ is μ-recursive. ■

Example 3.2.9. Let $f : \mathbb{N}^k \to \mathbb{N}$ be a μ-recursive total function. If f is strictly increasing, then there is a decidable set A such that $\operatorname{im} f|_{A^k}$ is $\mathbb{N} \setminus A$. [...]

The set of semidecidable subsets of $\mathbb{N}$ is clearly countable. It can even be encoded in a computable way, in the following sense: there is a μ-recursive partial function $\chi : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ such that $\chi(m, x)\downarrow$ with $\chi(m, x) = 1$ if and only if x is in the m-th semidecidable subset of $\mathbb{N}$. (This enumeration is not assumed bijective, however.)

Proposition 3.2.10. *A semidecidable union of semidecidable subsets of* $\mathbb{N}$ *is itself semidecidable: if* I *is a semidecidable subset of* $\mathbb{N}$*, then the set*

$$X = \{x \in \mathbb{N} \mid \exists m \in I.\, (\chi(m, x)\downarrow \wedge \chi(m, x) = 1)\}$$

is also a semidecidable subset of $\mathbb{N}$.

Proof. One simply notes that X is a projection of the semidecidable set

$$Y = \left\{(x, m) \in \mathbb{N}^2 \,\middle|\, \chi(m, x)\downarrow \wedge \chi(m, x) = 1\right\}$$

and so X is itself semidecidable. ■

3 Undecidability

Since the set of semidecidable subsets of $\mathbb{N}$ is countable, a counting argument shows that there must exist subsets of $\mathbb{N}$ which are not even semidecidable; but are there any *interesting* sets which are not semidecidable? For example, if we could show that a set X is semidecidable but not decidable, then its complement $\mathbb{N} \setminus X$ must be not semidecidable.

Let us write $\boxed{m}$ for the m-th Turing machine, as enumerated by Kleene's T-predicate (theorem 3.1.3). To simplify notation, if $\mathfrak{M}$ is a Turing machine, we also write $\mathfrak{M}(\bar{a})$ for $\sigma^{\infty}_{\mathfrak{M}}(\bar{a})$.

Definition 3.3.1. A Turing machine $\mathfrak{M}$ **halts** on an input $\bar{a}$ if $\mathfrak{M}(\bar{a})\downarrow$. The **halting set** is the following subset of $\mathbb{N}$:

$$K = \left\{ \mathsf{pair}(m, s) \,\middle|\, m \in \mathbb{N} \text{ and } s \in \mathbb{N} \text{ and } s \text{ codes a tape } \bar{a} \text{ for } \boxed{m} \text{ and } \boxed{m}(\bar{a})\downarrow \right\}$$

Theorem 3.3.2 (Halting problem). *The halting set is semidecidable but not decidable.*

Proof. It is clear that K is semidecidable: simply use Kleene's T-predicate (theorem 3.1.3). Let F_1 be the universal unary μ-recursive function, and consider the set X defined below:

$$X = \{n \in \mathbb{N} \mid F_1(n, n)\downarrow\}$$

Since F_1 is μ-recursive, there is a Turing machine $\mathfrak{M}$ which computes F_1, so we could decide membership in X if we could decide membership in K.

Let $\chi_X : \mathbb{N} \to \{0, 1\}$ be the characteristic function of X, and let $f(n) = 1$ if $n \in X$, $f(n)\uparrow$ if $n \notin X$. The partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ so defined is μ-recursive if χ_X is. Suppose χ_X is μ-recursive. Then, there is a natural number n such that $f = F_1(n, -)$, so consider $f(n)$: if $f(n) = 1$, then $F_1(n, n)\uparrow$, so $f(n)\uparrow$; but if $f(n)\uparrow$, then $F_1(n, n)\downarrow$, so $f(n) = 1$—either way, we have a contradiction, so χ_X is not μ-recursive. Hence X and K are not decidable sets. ■

The fixed point theorem can be used to prove a recursion-theoretic generalisation of the undecidability of the halting problem:

Theorem 3.3.3 (Rice). *Let A be a subset of the set of* graphs *of all μ-recursive partial functions $\mathbb{N} \rightharpoonup \mathbb{N}$. The set*

$$X = \{n \in \mathbb{N} \mid \textit{the graph of } F_1(n, -) \textit{ is in } A\}$$

is decidable if and only if $X = \varnothing$ or $X = \mathbb{N}$.

Proof. Suppose we have $a \in X$ and $b \notin X$. If X is a decidable set, then the function $h : \mathbb{N} \to \mathbb{N}$ given below is a μ-recursive total function:

$$h(z) = \begin{cases} b & z \notin X \\ a & z \in X \end{cases}$$

By construction, $h(z) \in X$ if and only if $z \notin X$. Now, by corollary 3.1.11, there is a natural number n such that $F_1(n, -) = F_1(h(n), -)$. But that means $n \in X$ if and only if $n \notin X$—a contradiction. So X is not a decidable set. ■

Proposition 3.3.4. *The set* $X = \{n \in \mathbb{N} \mid F_1(n, -)$ *is a total function*$\}$ *is not semidecidable.*

Proof. It is easy to show that X is not *decidable* using Rice's theorem, but we are claiming something stronger here. The set X is non-empty, so it is recursively enumerable if and only if it is semidecidable, by proposition 3.2.4. Suppose, for a contradiction, that we have a μ-recursive surjection $g : \mathbb{N} \to X$. Consider the function $f : \mathbb{N} \to \mathbb{N}$ given below:

$$f(x) = F_1(g(x), x) + 1$$

This is a μ-recursive total function, so there is some natural number n for which $f = F_1(g(n), -)$. But we have

$$\begin{aligned} f(n) &= F_1(g(n), n) + 1 && \text{by definition of } f \\ f(n) &= F_1(g(n), n) && \text{by definition of } n \end{aligned}$$

which is a contradiction. Hence X cannot be recursively enumerable. ■

Definition 3.3.5. Let X and Y be subsets of $\mathbb{N}$. A total function $f : \mathbb{N} \to \mathbb{N}$ **separates** X from Y just if it has these properties:

- for all x in X, $f(x) = 1$, and
- for all y in Y, $f(y) \neq 1$.

We say X and Y are **recursively separable** just when there is a μ-recursive total function that separates X from Y; otherwise we say X and Y are **recursively inseparable**.

Remark 3.3.6. Clearly, there is μ-recursive total function separating X from Y if and only if there is a μ-recursive total function separating Y from X, so despite appearances recursive separability is indeed a property of *unordered* pairs of subsets of $\mathbb{N}$.

Example 3.3.7. Let $X \subseteq \mathbb{N}$. Then, X and $\mathbb{N} \setminus X$ are recursively separable if and only if X is decidable.

Informally, we may think of recursive separability of X and Y as a relativised version of decidability: it simply says that there is a decision procedure that correctly accepts all members of X and rejects all members of Y.

Lemma 3.3.8. *Let* X *and* Y *be subsets of* $\mathbb{N}$. *The following are equivalent:*

(i) X *and* Y *are recursively separable.*

(ii) *There is a decidable set* Z *such that* $X \subseteq Z$ *and* $Y \cap Z = \varnothing$.

Proof. (i) $\Rightarrow$ (ii). Let $f : \mathbb{N} \to \mathbb{N}$ be a μ-recursive total function that separates X from Y, and take $Z = \{z \in \mathbb{N} \mid f(z) = 1\}$.

(ii) $\Rightarrow$ (i). The characteristic function of Z separates X from Y and is a μ-recursive total function by hypothesis. ■

Example 3.3.9. X and Y as defined below are recursively inseparable:

$$X = \{n \in \mathbb{N} \mid F_1(n, n){\downarrow} \text{ and } F_1(n, n) \neq 1\}$$
$$Y = \{n \in \mathbb{N} \mid F_1(n, n){\downarrow} \text{ and } F_1(n, n) = 1\}$$

Let f be a μ-recursive total function. Then, there is a natural number n such that $f = F_1(n, -)$. Consider $f(n)$: if $f(n) = 1$, then $F_1(n, n){\downarrow}$ and $F_1(n, n) = 1$, so $n \in Y$; if $f(n) \neq 1$, then $F_1(n, n){\downarrow}$ and $F_1(n, n) \neq 1$, so $n \in X$—either way, f cannot separate X from Y. Hence, X and Y must be recursively inseparable.

4 Applications to logic

In this section, we work with theories in a suitable formal system. The precise requirements are left vague; for concreteness, one could take the formal system to be ordinary second-order logic.

Proposition 3.4.1 (Craig). *Let* $\mathcal{T}$ *be a theory in a countable language* $\mathcal{L}$. *Fix an encoding of* $\mathcal{L}$. *If* $\mathcal{T}$ *has a semidecidable axiomatisation in this encoding, then* $\mathcal{T}$ *also has a decidable axiomatisation in this encoding. We say* $\mathcal{T}$ *is a* **recursively axiomatisable theory** *in this case.*

Proof. Let A be a semidecidable axiomatisation of $\mathcal{T}$. If A is empty, then we already have a decidable axiomatisation. Otherwise, A is recursively enumerable, so there is a primitive recursive surjection $g : \mathbb{N} \to A$. Let B be the set of formulae such that

$$g(n) = \varphi \text{ if and only if } (\top \wedge \underbrace{(\varphi \wedge (\cdots \wedge (\varphi}_{n \text{ times}} \wedge \varphi) \cdots))) \in B$$

Clearly, B is a decidable (even primitive recursive!) axiomatisation of $\mathcal{T}$: to determine whether $\psi \in B$, one simply checks whether ψ is of the above form for *some* φ and then checks whether $g(n) = \varphi$ for the appropriate n. ■

Proposition 3.4.2 (Gödel). *If $\mathcal{T}$ is a recursively axiomatisable theory, the set of theorems of $\mathcal{T}$ is recursively enumerable.*

Proof. The set $\left\{(p, q) \in \mathbb{N}^2 \,\middle|\, q \text{ codes a valid proof of } p \text{ in } \mathcal{T}\right\}$ is obviously a non-empty decidable set, and the set of theorems of $\mathcal{T}$ is a projection of this set, hence, semidecidable and recursively enumerable by propositions 3.2.7 and 3.2.4. ■

Theorem 3.4.3 (Gödel's first incompleteness theorem). *If $\mathcal{T}$ is a recursively axiomatisable sound theory of arithmetic,*[1] *such as Peano arithmetic, then there is a true arithmetical statement that $\mathcal{T}$ cannot prove.*

Proof. The hard work goes into showing that every μ-recursive partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ is definable in $\mathcal{T}$, in the sense that there is a predicate $\varphi(\vec{x}, y)$ in the language of $\mathcal{T}$ such that

$$\varphi(\vec{x}, y) \text{ holds if and only if } f(\vec{x})\downarrow \text{ and } f(\vec{x})$$

Assume this has been done. Then, it is not hard to see that the set

$$Y = \{n \in \mathbb{N} \mid \mathcal{T} \text{ proves } F_1(n, -) \text{ is total}\}$$

is a semidecidable set. If Y is empty, then there is nothing to do: $\mathcal{T}$ is unable to prove that, say, the identity function is total. Otherwise, Y is non-empty and is recursively enumerable by proposition 3.2.4, so let $h : \mathbb{N} \to \mathbb{N}$ be a recursive enumeration of Y. Consider the function $g : \mathbb{N} \to \mathbb{N}$ defined below:

$$g(x) = F_1(h(x), x) + 1$$

Assuming only the consistency of $\mathcal{T}$, we can show that $\mathcal{T}$ is unable to prove that g is total. Indeed, if $\mathcal{T}$ proves that g is total, then there is a number m such that $g = F_1(h(m), -)$, from which it follows that $\mathcal{T}$ proves

$$\begin{aligned} g(m) &= F_1(h(m), m) + 1 && \text{by definition of } g \\ g(m) &= F_1(h(m), m) && \text{by definition of } m \end{aligned}$$

[1] This means $\mathcal{T}$ does not prove any false arithmetical statements.

which means $\mathcal{T}$ proves a contradiction; but $\mathcal{T}$ is consistent, so $\mathcal{T}$ cannot prove that g is total. But if $\mathcal{T}$ is sound, then g *is* total, so we have produced a true arithmetical statement unprovable in $\mathcal{T}$. ■

Definition 3.4.4. A **productive set** is a semidecidable subset X of $\mathbb{N}$ such that there exists a μ-recursive total function $g : \mathbb{N} \to \mathbb{N}$ with the following property: for all n such that $\operatorname{im} F_1(n, -) \subseteq X$, $g(n) \notin \operatorname{im} F_1(n, -)$.

Remark 3.4.5. The above proof of Gödel's first incompleteness theorem also shows that the set of true arithmetical statements is productive.

Theorem 3.4.6 (Trakhtenbrot). *If Σ is a first-order signature with countably infinitely many unary and binary predicate symbols, then the set of all first-order sentences true in all* finite *Σ-structures is not semidecidable.*

Proof. We will reduce this to the halting problem. The main idea is to find a μ-recursive total function which maps a pair (m, s) of natural numbers to a first-order sentence φ such that there is a finite Σ-structure X falsifying φ if and only if $\boxed{m}(\bar{a})\downarrow$, where $\bar{a}$ is the tape for $\boxed{m}$ coded by s. If we had such a μ-recursive total function *and* the set of all first-order sentences true in all finite Σ-models were semidecidable, then the complement of the halting set would be semidecidable, contradicting theorem 3.3.2.

Fix a Turing machine $\mathfrak{M}$ and a tape $\bar{a}$. Since Σ has countably many unary and binary predicate symbols, we have enough symbols to represent these predicates:

- The unary predicate "x is the integer 0".
- The binary predicate "x and y are both integers and $x < y$".
- The binary predicate "x and y are integers and $y = x + 1$".
- For each a in the alphabet of $\mathfrak{M}$, the binary predicate "a is the letter at position x of the tape at time y".
- For each possible state q of $\mathfrak{M}$, the unary predicate "q is the state of $\mathfrak{M}$ at time x".

It is clear that these can be used to give a finite axiomatisation of the theory of finite courses-of-computations of $\mathfrak{M}$ on the input $\bar{a}$, so there is a single first-order formula ψ which asserts "X is a finite course-of-computation of $\mathfrak{M}$ on the input $\bar{a}$". Let θ be the first-order formula asserting "at no time in the course-of-computation X does $\mathfrak{M}$ reach an accepting state". We take φ to be the sentence $\psi \to \theta$. It is clear that $\neg\varphi$ holds if and only if X is a finite *and complete* course-of-computation of $\mathfrak{M}$ on input $\bar{a}$. Thus we have the desired sentence φ. ■

5 Oracles

Definition 3.5.1. Let A and B be subsets of $\mathbb{N}$. A **many-one reduction** from A to B is a μ-recursive total function $f : \mathbb{N} \to \mathbb{N}$ such that $n \in A$ if and only if $f(n) \in B$. Note that neither A nor B need to be semidecidable. We say A is **many-one reducible** to B just if there exists a many-one reduction as above, and we write $A \leqslant_m B$.

Informally, $A \leqslant_m B$ means that the decision problem for A is at most as hard as the decision problem for B. In terms of machines, $A \leqslant_m B$ means that there is a machine $\mathfrak{M}$ that, when given n as an input, produces a tape suitable for input to an oracle machine $\mathfrak{O}$ that decides membership of B, such that the concatenation of these two machines is a machine that decides membership of A.

Proposition 3.5.2. *The relation $\leqslant_m$ is a preorder on $\mathscr{P}(\mathbb{N})$.*

Proof. Reflexivity is clear, and transitivity follows from the fact that the composition of two μ-recursive total functions is a μ-recursive total function. ■

Proposition 3.5.3. *Let $A \subseteq \mathbb{N}$, and let K be the halting set:*

$$K = \left\{ \mathsf{pair}(m, s) \,\middle|\, m \in \mathbb{N} \textit{ and } s \in \mathbb{N} \textit{ and } s \textit{ codes a tape } \bar{a} \textit{ for } \boxed{m} \textit{ and } \boxed{m}(\bar{a})\downarrow \right\}$$

The following are equivalent:

(i) $A \leqslant_m K$.

(ii) *A is semidecidable.*

Proof. (i) $\Rightarrow$ (ii). The halting set is semidecidable by theorem 3.3.2, and the inverse image of a semidecidable set under a μ-recursive total function is again semidecidable by proposition 3.2.6.

(ii) $\Rightarrow$ (i). Let $g : \mathbb{N} \rightharpoonup \mathbb{N}$ be a partial characteristic function for A. Clearly, we may assume without loss of generality that $\operatorname{dom} g = A$. Since g is μ-recursive, it is computed by some Turing machine, say $\boxed{m}$. Let $f : \mathbb{N} \to \mathbb{N}$ be the function $n \mapsto \mathsf{pair}(m, \mathsf{in}(n))$, where $\mathsf{in} : \mathbb{N} \to \mathbb{N}$ encodes a natural number n as an input tape for $\boxed{m}$. Then, $n \in A$ if and only if $f(n) \in K$, so f is the required many-one reduction of A to K. ■

It turns out we get a subtly different notion of reducibility if we allow our machines to ask the oracle arbitrarily many questions.

Definition 3.5.4. An **augmentable Turing machine** $\mathfrak{M}$ comprises the same data as a Turing machine, except that the transition function of an oracle machine is a partial function $\delta : (Q \setminus F) \times \Gamma^2 \rightharpoonup Q \times \Gamma \times \{-1, 0, +1\}^2$, where Q is the set of states of $\mathfrak{M}$, F is the set of accepting states of $\mathfrak{M}$, and Γ is the alphabet of $\mathfrak{M}$.

An **oracular tape** for $\mathfrak{M}$ is an integer-indexed sequence $(o_i \mid i \in \mathbb{Z})$ such that each o_i is in Γ.[1] An **oracle machine** is an augmentable Turing machine equipped with a oracular tape.

Let $\mathcal{L}$ be the set of all tapes for an augmentable Turing machine $\mathfrak{M}$, and let $\mathcal{O}$ be the set of all oracular tapes for $\mathfrak{M}$. The transition function of $\mathfrak{M}$ induces a partial function $\tau : Q \times \mathcal{L} \times \mathcal{O} \rightharpoonup Q \times \mathcal{L} \times \mathcal{O}$ as follows:

- If $q \in F$, then $\tau(q, \bar{a}, \bar{o}) = (q, \bar{a}, \bar{o})$.
- If $\delta(q, a_0, o_0)\uparrow$, then $\tau(q, \bar{a}, \bar{o})\uparrow$.
- Otherwise, if $\delta(q, a_0, o_0) = (q', a', u, v)$, then $\tau(q, \bar{a}, \bar{o}) = (q', \bar{a}', \bar{o}')$, where
$$a'_i = \begin{cases} a' & \text{if } i = -u \\ a_{i+u} & \text{if } i \neq -u \end{cases}$$
and $o'_i = o_{i+v}$.

Note that this function is essentially primitive recursive if we fix $\bar{o}$ and discard $\bar{o}'$. (If Γ contains at least two symbols, then $\mathcal{O}$ is an uncountable set, so we must treat $\bar{o}$ and $\bar{o}'$ separately!) We define a partial function $\tau^* : \mathcal{L} \times \mathcal{O} \times \mathbb{N} \rightharpoonup Q \times \mathcal{L} \times \mathcal{O}$ by primitive recursion:

$$\tau^*(\bar{a}, \bar{o}, 0) = (q_0, \bar{a}, \bar{o})$$
$$\tau^*(\bar{a}, \bar{o}, n+1) = \tau(\tau^*(\bar{a}, \bar{o}, n))$$

[1] Note that we do *not* require the set $\{i \in \mathbb{Z} \mid a_i \neq b\}$ to be finite!

Finally, define a partial function $\tau^\infty : \mathcal{L} \times \mathcal{O} \rightharpoonup Q \times \mathcal{L}$ by unbounded minimisation: $\tau^\infty(\bar{a}, \bar{o}) = (q, \bar{a}')$ where $\tau^*(\bar{a}, \bar{o}, n) = (q, \bar{a}', \bar{o}')$ and n is the smallest natural number such that $q \in F$. We think of $\tau^\infty(-, \bar{o}) : \mathcal{L} \rightharpoonup \mathcal{L}$ as the partial function computed by the oracle machine $(\mathfrak{M}, \bar{o})$. In general, $\tau^\infty(-, \bar{o})$ is not μ-recursive and depends on $\bar{o}$, though it is obvious that every Turing machine can be turned into an augmentable Turing machine in such a way that $\tau^\infty(-, \bar{o})$ computes the same μ-recursive function as the original Turing machine and does not depend on $\bar{o}$.

Let $\mathcal{O}$ be the set of all oracular tapes for an augmentable Turing machine $\mathfrak{M}$. Consider the following distance function on $\mathcal{O}$:

$$d(\bar{o}, \bar{o}') = \inf\{2^{-n} \mid \forall i \in \mathbb{Z}. (-n < i < n) \to (o_i = o'_i)\}$$

The key point is that $d(\bar{o}, \bar{o}')$ is small when $\bar{o}$ and $\bar{o}'$ differ only in the cells with small indices. It is clear that d makes $\mathcal{O}$ into a compact ultrametric space.

Proposition 3.5.5. *Let* $(\mathfrak{M}, \bar{o})$ *be an oracle machine. For each tape* $\bar{a}$*, if* $\tau^\infty(\bar{a}, \bar{o})\downarrow$*, then there is a positive real number* ϵ *such that*

$$\tau^*(\bar{a}, \bar{o}, n) = \tau^*(\bar{a}, \bar{o}', n) \textit{ for all natural numbers } n$$

for all oracular tapes $\bar{o}'$ *such that* $d(\bar{o}, \bar{o}') < \epsilon$*. In particular,* $\tau^\infty : \mathcal{L} \times \mathcal{O} \rightharpoonup \mathcal{L}$ *is continuous partial function when* $\mathcal{L}$ *is given the discrete topology.*

Proof. It is clear that at each time N, $\mathfrak{M}$ can only have accessed the cells of $\bar{o}$ with indices having absolute value less than N, so for any oracular tape $\bar{o}'$ such that $d(\bar{o}, \bar{o}') < 2^{-N}$, we must have

$$\tau^*(\bar{a}, \bar{o}, n) = \tau^*(\bar{a}, \bar{o}', n) \text{ for all natural numbers } n \leqslant N$$

Since we are assuming $\tau^\infty(\bar{a}, \bar{o})\downarrow$, there is an N such that

$$\tau^*(\bar{a}, \bar{o}', n) = \tau^*(\bar{a}, \bar{o}, N) \text{ for all natural numbers } n > N$$

for all oracular tapes $\bar{o}'$ such that $d(\bar{o}, \bar{o}') < 2^{-N}$, so we may take $\epsilon = 2^{-N}$. ■

Now, fix an alphabet Γ with blank symbol b and at least two distinct non-blank symbols 0 and 1. Let $\mathcal{L}$ be the set of all tapes written in Γ. For each natural number k, choose an injective total function $\mathsf{in}_k : \mathbb{N}^k \to \mathcal{L}$, and also choose an injective total function $\mathsf{out} : \mathbb{N} \to \mathcal{L}$.

Definition 3.5.6. Let $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ be a partial function. An oracle machine $(\mathfrak{M}, \bar{o})$ **computes** f just if $(\mathfrak{M}, \bar{o})$ satisfies the following conditions:

- Γ is a subset of the alphabet of $\mathfrak{M}$, and the symbols on the oracular tape $\bar{o}$ are taken from Γ.
- $\mathfrak{M}$ has a unique accepting state $\top$.
- For all $\vec{x}$ in $\mathbb{N}^k$, $\tau^\infty(\mathsf{in}_k(\vec{x}))\downarrow$ if and only if $f(\vec{x})\downarrow$, and when $f(\vec{x})\downarrow$, we have $\tau^\infty(\mathsf{in}_k(\vec{x})) = (\top, \mathsf{out}(f(\vec{x})))$.

A **function computable relative to** $\bar{o}$ is a partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ for which there is an augmentable Turing machine $\mathfrak{M}$ such that the oracle machine $(\mathfrak{M}, \bar{o})$ computes f.

We can define a recursion-theoretic analogue of the above notion in a fairly straightforward way:

Definition 3.5.7. Let $p : \mathbb{N}^q \rightharpoonup \mathbb{N}$ be a partial function. A **function recursive relative to** p is any one of the following:

- The partial function p itself.
- A primitive recursive function.
- A composite $h \circ (g_1, \ldots, g_\ell) : \mathbb{N}^k \rightharpoonup \mathbb{N}$ of partial functions $g_1, \ldots, g_\ell : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^\ell \rightharpoonup \mathbb{N}$ recursive relative to p.
- A function $\rho[h; g] : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ obtained by primitive recursion on partial functions $g : \mathbb{N}^k \rightharpoonup \mathbb{N}$, $h : \mathbb{N}^{k+2} \rightharpoonup \mathbb{N}$ recursive relative to p.
- A partial function of the form $\mu[f] : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ for some *total* function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ recursive relative to p.

Remark 3.5.8. Every partial function $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ is obviously recursive relative to itself. It is also computable relative to *some* $\bar{o}$.

To be precise, for each natural number k, there is a *universal* augmentable Turing machine $\mathfrak{M}$ with the property that each $f : \mathbb{N}^k \rightharpoonup \mathbb{N}$ is computed by $(\mathfrak{M}, \bar{o})$ for some oracular tape $\bar{o}$. It is clear what $(\mathfrak{M}, \bar{o})$ has to be: $\bar{o}$ must encode the graph of f, and $\mathfrak{M}$ must be a machine that *looks* up the value of $f(\vec{x})$ from the oracular tape $\bar{o}$!

Definition 3.5.9. Let A and B be subsets of $\mathbb{N}$. Let $\bar{o}$ be the oracular tape given by

$$o_i = \begin{cases} b & \text{if } i < 0 \\ 0 & \text{if } i \geqslant 0 \text{ and } i \notin B \\ 1 & \text{if } i \geqslant 0 \text{ and } i \in B \end{cases}$$

We say A is **Turing reducible** to B if the characteristic function of A is computable relative to $\bar{o}$, and we write $A \leqslant_T B$.

Remark 3.5.10. We could equally well define $A \leqslant_T B$ to mean that the χ_A is *recursive* relative to χ_B, but then we would have to prove the equivalence of the two notions!

Proposition 3.5.11. *Let A and B be subsets of $\mathbb{N}$. $A \leqslant_T B$ if and only if $\mathbb{N} \setminus A \leqslant_T B$.*

Proof. Clearly, if χ_A is computable relative to $\bar{o}$, then $1 \dot{-} \chi_A$ is also computable relative to $\bar{o}$. ■

Example 3.5.12. If $A \leqslant_m B$, then $A \leqslant_T B$. However, $A \leqslant_T B$ does not imply $A \leqslant_m B$. Indeed, let K be halting set; then $\mathbb{N} \setminus K \leqslant_T K$ by the proposition above, but $A \leqslant_m K$ if and only if A is semidecidable, by proposition 3.5.3. This shows that $\leqslant_m$ is a finer relation on $\mathscr{P}(\mathbb{N})$ than $\leqslant_T$.

Proposition 3.5.13. *The relation $\leqslant_T$ is a preorder on $\mathscr{P}(\mathbb{N})$.*

Proof. We already remarked on reflexivity, and transitivity is clear (though the details may be tedious). ■

Definition 3.5.14. A **Turing degree of reducibility** is an equivalence class of subsets of $\mathbb{N}$ under the equivalence relation $\sim$ given by

$$A \sim B \text{ if and only if } A \leqslant_T B \text{ and } B \leqslant_T A$$

We write **0** for the Turing degree of decidable sets. For each Turing degree **d**, **d**$'$ denotes the Turing degree of the halting set of machines equipped with an oracle of degree **d**.

Proposition 3.5.15. *Let A and B be subsets of $\mathbb{N}$. Then, there is a subset $A \sqcup B$ of $\mathbb{N}$ such that*

$$A \leqslant_T C \textit{ and } B \leqslant_T C \textit{ if and only if } A \sqcup B \leqslant_T C$$

for all $C \subseteq \mathbb{N}$. In particular, every finite subset of $\mathscr{P}(\mathbb{N})$ has a least upper bound with respect to $\leqslant_T$.

Proof. As suggested by the notation, $A \sqcup B$ is (in a suitable sense) the disjoint union of A and B. To be precise, it is the set given below:

$$A \sqcup B = \{2n \mid n \in A\} \cup \{2n+1 \mid n \in B\}$$

Clearly, $A \leqslant_T A \sqcup B$ and $B \leqslant_T A \sqcup B$. It is also clear that if χ_A and χ_B are both computable relative to an oracular tape $\bar{o}$, then $\chi_{A \sqcup B}$ is computable relative to $\bar{o}$. Hence $A \sqcup B$ is indeed the least upper bound of A and B with respect to $\leqslant_T$. To complete the proof, we must show that the empty set has a least upper bound: in other words, we must find a *lower* bound for $\mathscr{P}(\mathbb{N})$. But this is easy: if A is a decidable set, then $A \leqslant_T B$ for any $B \subseteq \mathbb{N}$. ■

Theorem 3.5.16 (Kleene–Post). *There exist incomparable Turing degrees, i.e. there are $A \subseteq \mathbb{N}$ and $B \subseteq \mathbb{N}$ such that $A \nleqslant_T B$ and $B \nleqslant_T A$.*

We will give several proofs. If we constrain the state space and alphabet of an augmentable Turing machine to be finite subsets of $\mathbb{N}$, then we can enumerate the set of augmentable Turing machines. Let $f_m(-;X) : \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial function computed by the m-th augmented Turing machine when equipped with the oracular tape corresponding to a subset X of $\mathbb{N}$. We shall abuse notation and identify a subset of $\mathbb{N}$ with the oracular tape corresponding to it in definition 3.5.9.

Proof. Let us regard $\mathscr{P}(\mathbb{N})$ as a subspace of the space of oracular tapes, equipped with the evident subspace metric. It can be shown that $\mathscr{P}(\mathbb{N})$ is then a compact ultrametric space (and, in fact, homeomorphic to the Cantor set). In particular, every non-empty open subset of $\mathscr{P}(\mathbb{N})$ contains a non-empty subset that is both open and closed. We shall construct a sequence of approximations to A and B, and use compactness to obtain the desired subsets of $\mathbb{N}$ in the limit.

Let $\mathcal{A}_0 = \mathcal{B}_0 = \mathscr{P}(\mathbb{N})$, and let $a_0 = b_0 = 0$. Suppose $\mathcal{A}_m$ and $\mathcal{B}_m$ are open subsets of $\mathscr{P}(\mathbb{N})$, and a_m and b_m are given natural numbers. Consider the following subsets of $\mathcal{A}_m$:

$$\mathcal{A}_{m,0} = \{A \in \mathcal{A}_m \mid f_m(b_m;A)\downarrow \text{ and } f_m(b_m;A) = 0\}$$

$$\mathcal{A}_{m,1} = \{A \in \mathcal{A}_m \mid f_m(b_m;A)\downarrow \text{ and } f_m(b_m;A) = 1\}$$

These are open subsets, by proposition 3.5.5.

- If $\mathcal{A}_{m,0}$ is non-empty, then there exist A_m in $\mathcal{A}_{m,0}$ and a natural number a'_m such that
$$\mathcal{A}'_m = \{A' \subseteq \mathbb{N} \mid A' \cap \{n \in \mathbb{N} \mid n < a'_m\} = A_m\} \subseteq \mathcal{A}_{m,0}$$
and $\mathcal{A}'_m$ is both open and closed; also set
$$\mathcal{B}'_m = \{B' \in \mathcal{B}_m \mid b_m \in B'\} \qquad b'_m = b_m + 1$$
and note that $\mathcal{B}'_m$ is open.

- If $\mathcal{A}_{m,1}$ is non-empty, then there exist A_m in $\mathcal{A}_{m,0}$ and a natural number a'_m such that
$$\mathcal{A}'_m = \{A' \subseteq \mathbb{N} \mid A' \cap \{n \in \mathbb{N} \mid n < a'_m\} = A_m\} \subseteq \mathcal{A}_{m,1}$$
and $\mathcal{A}'_m$ is both open and closed; also set
$$\mathcal{B}'_m = \{B' \in \mathcal{B}_m \mid b_m \notin B'\} \qquad b'_m = b_m + 1$$
and note that $\mathcal{B}'_m$ is open.

- Otherwise, if both $\mathcal{A}_{m,0}$ and $\mathcal{A}_{m,1}$ are empty, then set
$$\begin{aligned} \mathcal{A}'_m &= \mathcal{A}_m & a'_m &= a_m \\ \mathcal{B}'_m &= \mathcal{B}_m & b'_m &= b_m \end{aligned}$$
and continue.

Now consider the following subsets of $\mathcal{B}'_m$:
$$\mathcal{B}_{m,0} = \{B \in \mathcal{B}'_m \mid f_m(a'_m; B)\downarrow \text{ and } f_m(a'_m; B) = 0\}$$
$$\mathcal{B}_{m,1} = \{B \in \mathcal{B}'_m \mid f_m(a'_m; B)\downarrow \text{ and } f_m(a'_m; B) = 1\}$$
These are open subsets, by proposition 3.5.5.

- If $\mathcal{B}_{m,0}$ is non-empty, then there exist B'_m in $\mathcal{B}_{m,0}$ and a natural number b_{m+1} such that
$$\mathcal{B}_{m+1} = \{B' \subseteq \mathbb{N} \mid B' \cap \{n \in \mathbb{N} \mid n < b_{m+1}\} = B_{m+1}\} \subseteq \mathcal{B}_{m,0}$$
and $\mathcal{B}_{m+1}$ is both open and closed; also set
$$\mathcal{A}_{m+1} = \{A' \in \mathcal{A}_{m+1} \mid a'_m \in A'\} \qquad a_{m+1} = a'_m + 1$$
and note that $\mathcal{A}_{m+1}$ is open.

- If $\mathcal{B}_{m,1}$ is non-empty, then there exist B'_m in $\mathcal{B}_{m,0}$ and a natural number b_{m+1} such that

  $$\mathcal{B}_{m+1} = \{B' \subseteq \mathbb{N} \mid B' \cap \{n \in \mathbb{N} \mid n < b_{m+1}\} = B_{m+1}\} \subseteq \mathcal{B}_{m,1}$$

  and $\mathcal{B}_{m+1}$ is both open and closed; also set

  $$\mathcal{A}_{m+1} = \{A' \in \mathcal{A}_{m+1} \mid a'_m \notin A'\} \qquad a_{m+1} = a'_m + 1$$

  and note that $\mathcal{A}_{m+1}$ is open.

- Otherwise, if both $\mathcal{B}_{m,0}$ and $\mathcal{B}_{m,1}$ are empty, then set

  $$\begin{aligned} \mathcal{A}_{m+1} &= \mathcal{A}'_m & a_{m+1} &= a'_m \\ \mathcal{B}_{m+1} &= \mathcal{B}'_m & b_{m+1} &= b'_m \end{aligned}$$

  and continue.

Thus, we obtain two decreasing chains of non-empty subsets of $\mathscr{P}(\mathbb{N})$:

$$\begin{gathered} \mathcal{A}_0 \supseteq \mathcal{A}_1 \supseteq \mathcal{A}_2 \supseteq \cdots \\ \mathcal{B}_0 \supseteq \mathcal{B}_1 \supseteq \mathcal{B}_2 \supseteq \cdots \end{gathered}$$

Choosing a sequence of elements for each chain, we may obtain a convergent subsequence by compactness; since $\mathcal{A}_m$ and $\mathcal{B}_m$ contain the closure of $\mathcal{A}_{m+1}$ and $\mathcal{B}_{m+1}$ (respectively), it follows that $\bigcap_m \mathcal{A}_m$ and $\bigcap_m \mathcal{B}_m$ are non-empty. Let $A \in \bigcap_m \mathcal{A}_m$ and $B \in \bigcap_m \mathcal{A}_m$. By construction, for each f_m, either

- $f_m(b_m; A)\downarrow$, $f_m(b_m; A) \neq 1$, and $b_m \in B$; or

- $f_m(b_m; A)\downarrow$, $f_m(b_m; A) = 1$, and $b_m \notin B$; or

- $f_m(b_m; A)\uparrow$

and similarly, either

- $f_m(a'_m; B)\downarrow$, $f_m(a'_m; B) \neq 1$, and $a'_m \in A$; or

- $f_m(a'_m; B)\downarrow$, $f_m(a'_m; B) = 1$, and $a'_m \notin A$; or

- $f_m(a'_m; B)\uparrow$

so $f_m(-; A)$ cannot be the characteristic function of B and $f_m(-; B)$ cannot be the characteristic function of A, i.e. $A \not\leqslant_T B$ and $B \not\leqslant A$. ∎

We may also rephrase the proof and obtain slightly stronger result:

Theorem 3.5.17 (Kleene–Post). *There exist incomparable Turing degrees* below $\mathbf{0}'$, *i.e. there are* $A \subseteq \mathbb{N}$ *and* $B \subseteq \mathbb{N}$ *such that* $A \leqslant_T K$, $B \leqslant_T K$, $A \not\leqslant_T B$, *and* $B \not\leqslant_T A$.

Proof. We will construct finite approximations to A and B by induction on m, and then use proposition 3.5.5 to pass to the limit.

Let $A_0 = B_0 = \varnothing$, $a_0 = b_0 = 0$. We take the following as our inductive hypotheses:

(i) $A_m \subseteq A_{m+1}$ and $d(A_m, A_{m+1}) \leqslant 2^{-a_m}$.

(ii) $a_m \leqslant a'_m \leqslant \max A_{m+1} < a_{m+1}$.

(iii) For all $A \subseteq \mathbb{N}$ such that $d(A_{m+1}, A) \leqslant 2^{-a_{m+1}}$, either

- $f_m(b'_m; A)\uparrow$, or
- $f_m(b'_m; A)\downarrow$, $f_m(b'_m; A) = 1$, and $b'_m \notin B_{m+1}$, or
- $f_m(b'_m; A)\downarrow$, $f_m(b'_m; A) \neq 1$, and $b'_m \in B_{m+1}$.

(iv) $B_m \subseteq B_{m+1}$ and $d(B_m, B_{m+1}) \leqslant 2^{-b_m}$.

(v) $b_m \leqslant b'_m \leqslant \max B_{m+1} < b_{m+1}$.

(vi) For all $B \subseteq \mathbb{N}$ such that $d(B_{m+1}, B) \leqslant 2^{-b_{m+1}}$, either

- $f_m(a'_m; B)\uparrow$, or
- $f_m(a'_m; B)\downarrow$, $f_m(a'_m; B) = 1$, and $a'_m \notin A_{m+1}$, or
- $f_m(a'_m; B)\downarrow$, $f_m(a'_m; B) \neq 1$, and $a'_m \in A_{m+1}$.

Now, assume we are given A_m, B_m, a_m, and b_m. Consider the following condition:

$$A \subseteq \mathbb{N} \text{ and } A \cap \{n \in \mathbb{N} \mid n < a_m\} = A_m \text{ and } \chi_m(b_m; A)\downarrow \tag{a}$$

If there are no sets A satisfying condition (a), then take $A' = A_m$, $B' = B_m$, $a'_m = a_m$, $b'_m = b_m$.

Otherwise, let A be a set satisfying condition (a). By continuity, there is a positive real number ϵ such that $f_m(b_m; A') = f_m(b_m; A)$ for all A such that $d(A, A') < \epsilon$. In particular, we can choose a *finite* set A' with $d(A, A') < \frac{1}{2}\epsilon$. Choose a natural number a'_m such that $\max A' < a'_m$, $2^{-a'_m} < \frac{1}{2}\epsilon$, and $a_m \leqslant a'_m$. If $f_m(b_m; A') = 1$, then set $B' = B_m$; otherwise, set $B' = B_m \cup \{b_m\}$.

Now consider this condition:

$$B \subseteq \mathbb{N} \text{ and } B \cap \{n \in \mathbb{N} \mid n < b'_m + 1\} = B' \text{ and } \chi_m(a'_m; B)\downarrow \qquad \text{(b)}$$

If there are no sets B satisfying condition (b), then take $A_{m+1} = A'$, $B_{m+1} = B'$, $a_{m+1} = a'_m + 1$, $b_{m+1} = b'_m + 1$.

Otherwise, let B be a set satisfying condition (a). By continuity, there is a positive real number ϵ' such that $f_m(a'_m; B'') = f_m(a'_m; B)$ for all B such that $d(B, B'') < \epsilon'$. In particular, we can choose a *finite* set B'' with $d(B, B'') < \epsilon'$. Choose a natural number b_{m+1} such that $\max B'' < b_{m+1}$ and $2^{-b_{m+1}} < \epsilon'$. If $f_m(a'_m; B'') = 1$, then set $A'' = A'$; otherwise, set $A'' = A' \cup \{a'_m\}$. Either way, set $A_{m+1} = A''$, $B_{m+1} = B''$, $a_{m+1} = a'_m + 1$, $b_{m+1} = b''$.

By construction, (i), (ii), (iv), (v), and (vi) hold. The triangle inequality implies (iii) holds. Now, take $A = \bigcup_m A_m$ and $B = \bigcup_m B_m$. For each natural number m, we have $d(A_{m+1}, A) \leqslant 2^{-a_{m+1}}$ and $d(B_{m+1}, B) \leqslant 2^{-b_{m+1}}$, so $f_m(-; B) \neq \chi_A$ and $f_m(-; A) \neq \chi_B$. Thus $A \nleqslant_T B$ and $B \nleqslant_T A$, as required. To see that $A \leqslant_T K$ and $B \leqslant_T K$, observe that [...] ■

Proposition 3.5.18 (Jockusch). *Let* $[\mathbb{N}]^3$ *be the set of subsets of* $\mathbb{N}$ *containing exactly* 3 *elements. There exists a primitive recursive function* $h : [\mathbb{N}]^3 \to \mathbb{N}$ *such that* $\operatorname{im} h$ *is* $\{0, 1\}$ *and the halting set is Turing reducible to any infinite subset* A *of* $\mathbb{N}$ *such that* h *is constant on* $[A]^3$.

Proof. Let us write $\boxed{m}_y(s)\downarrow$ to abbreviate the predicate "the p-th Turing machine halts on input s after at most y steps". We define h as follows. Given $\{x, y, z\}$ with $x < y < z$, set $h(\{x, y, z\}) = 1$ if

$$\forall m < x. \forall s < x. \left[\boxed{m}_y(s)\downarrow \Leftrightarrow \boxed{m}_z(s)\downarrow\right]$$

and set $h(\{x, y, z\}) = 0$ otherwise. This is primitive recursive by theorem 3.1.3. Let A be an infinite subset of $\mathbb{N}$ and suppose h is constant on $[A]^3$. Clearly, for any fixed x, $h(\{x, y, z\}) = 1$ for all sufficiently large y and z. Since A is infinite, it is unbounded, so h must be equal to the constant 1.

Now, let m and s be given numbers. Observe that, for all sufficiently large z, $\boxed{m}(s)\downarrow$ if and only if $\boxed{m}_z(s)\downarrow$. Since A is unbounded, there exist x in A such that $\max\{m, s\} < x$ and z in A such that $x < z$ and $\boxed{m}(s)\downarrow$ if and only if $\boxed{m}_z(s)\downarrow$. By construction of A, for any y in A such that $x < y < z$, we have $\boxed{m}(a)\downarrow$ if and only if $\boxed{m}_y(s)\downarrow$. Therefore, to determine whether or not $\boxed{m}(s)\downarrow$, it is enough to find x and y in A such that $\max\{m, s\} < x < y$, and then determine whether or not $\boxed{m}_y(s)\downarrow$. Thus, the halting set is Turing-reducible to A. ■

Bibliography

Barendregt, Henk P.

[1984] *The lambda calculus. Its syntax and semantics*. Revised. Studies in Logic and the Foundations of Mathematics 103. Amsterdam: North-Holland Publishing Co., 1984, pp. xv+621. isbn: 0-444-86748-1; 0-444-87508-5.

Church, Alonzo and J. Barkley Rosser

[1936] "Some properties of conversion". In: *Trans. Amer. Math. Soc.* 39.3 (1936), pp. 472–482. issn: 0002-9947. doi: 10.2307/1989762.

Cohen, Daniel E.

[1987] *Computability and logic*. Ellis Horwood Series: Mathematics and its Applications. Chichester: Ellis Horwood Ltd., 1987. 243 pp. isbn: 0-7458-0034-3.

Curry, Haskell B.

[1930a] "Grundlagen der Kombinatorischen Logik, I". In: *Amer. J. Math.* 52.3 (1930), pp. 509–536. issn: 0002-9327. doi: 10.2307/2370619.

[1930b] "Grundlagen der Kombinatorischen Logik, II". In: *Amer. J. Math.* 52.4 (1930), pp. 789–834. issn: 0002-9327. doi: 10.2307/2370716.