Notes on Simply Typed Lambda Calculus*

Ralph Loader†

February, 1998

The purpose of this course is to provide an introduction to λ-calculi, specifically the simply typed lambda calculus ($\lambda^{\rightarrow}$).

λ-calculi are formalisms that are useful in computer science. They are languages that express both *computational* and *logical* information.

Computational information in that they can be see as functional programming languages, or more realistically, a solid core on which to build a functional language.

Logical information in two ways. First, typed λ-calculi can be used directly as logics—these are 'intuitionistic type theories' such as the calculus of constructions (e.g., used in the 'Lego' theorem proving software developed in this department). Second, typed λ-calculi underly the term structure of higher order logics.

There are many different λ-calculi—everyone working in the field has their own pet language(s)—but they generally fall into a few classes of broadly similar languages. This course does not provide a wide ranging survey, but instead aims to explain a good range of theoretical techniques, most of which are applicable to a much wider range of languages than we will actually study.

The first half of the course gives a fairly thorough introduction to fairly traditional and fundamental results about λ-calculi. The second half of the course gives a detailed, quantitative analysis of $\lambda^{\rightarrow}$.

The book [3] covers much of the material in this course, especially sections 1–3. In section 1, we look at the untyped λ-calculus; [1] is a fairly comprehensive work on this. The article [2] is a fairly comprehensive work on pure type systems, which include the simply typed λ-calculus and impredicative intuitionistic type theories.

*Version 1.8 dated 1998/02/16 15:10:19.

†Department of Computer Science, University of Edinburgh. ⟨*loader@dcs.ed.ac.uk*⟩

References

[1] Hendrik P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1981 (revised edition 1984).

[2] Hendrik P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbai, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 199?

[3] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1989.

Contents

1 Untyped Lambda Calculus

A typed λ-calculus has a language in two parts: the language of types, and the language of terms. In order to introduce some basic notions, we shall first look at the **untyped λ-calculus** Λ, so that we can consider terms independently of types.

1.1 Syntax

The set $\mathcal{T}$ terms of the untyped λ-calculus is given by the following:

- A countably infinite set $\mathcal{V}$ of variables. Each variable is a term.
- If t and r are terms, then so is $t\,r$.
- If s is a term, and $\mathtt{x}$ is a variable, then $\lambda\mathtt{x}.\,s$ is a term.

Each term has a unique representation in one of the above forms, and the set of terms is the smallest set closed under the three clauses above.

In short, this is given by the following grammar:

$$\mathcal{T} := \mathcal{V} \mid \mathcal{T}\,\mathcal{T} \mid \lambda\mathcal{V}.\,\mathcal{T}$$

Induction and unique reading of terms are given as follows.

Induction: Suppose that S is a set (of terms) such that (a) $\mathtt{x} \in S$ for each variable $\mathtt{x}$, (b) if $t, r \in S$ then $t\,r \in S$ and (c) if $s \in S$ then $\lambda\mathtt{x}.\,s \in S$. Then S contains all terms.

Unique reading: Each term is exactly one of the following: a variable, an application, or an abstraction. If $t\,r = t'\,r'$ then $t = t'$ and $r = r'$. If $\lambda\mathtt{x}.\,s = \lambda\mathtt{x}'.\,s'$ then $\mathtt{x} = \mathtt{x}'$ and $s = s'$. (Also, if two variables are equal as terms, then they are equal as variables—this one is confusing.)

These two principles enable us to define functions on $\mathcal{T}$ by recursion:

Recursion: Let S be a set, and $F_{\text{var}} : \mathcal{V} \longrightarrow S$, $F_{\text{app}} : S \times S \longrightarrow S$ and $F_{\text{abs}} : \mathcal{V} \times S \longrightarrow S$ be functions. Then there is a unique function $F : \mathcal{T} \longrightarrow S$ such that

$$F(\mathtt{x}) = F_{\text{var}}(\mathtt{x}), \quad F(t\,r) = F_{\text{app}}(F(t), F(r)), \quad F(\lambda\mathtt{x}.\,s) = F_{\text{abs}}(\mathtt{x}, F(s)).$$

As an application of the above, take S to be the set of natural numbers, and let $l_{\text{var}}(\mathtt{x}) = 1$, $l_{\text{app}}(m, n) = n + m + 1$ and $l_{\text{abs}}(\mathtt{x}, m) = m + 1$. Then the function l given by recursion satisfies:

$$l(\mathtt{x}) = 1, \qquad l(t\,r) = l(t) + l(r) + 1, \qquad l(\lambda\mathtt{x}.\,s) = l(s) + 1.$$

The number $l(r)$ is the **length** of the term r.

Another instance of recursion: Let $S = \mathcal{P}_{\text{fin}}(\mathcal{V})$, the set of finite sets of variables. Let $\mathrm{FV}_{\text{var}}(\mathtt{x}) = \{\mathtt{x}\}$, $\mathrm{FV}_{\text{app}}(c, a) = c \cup a$ and $\mathrm{FV}_{\text{abs}}(\mathtt{x}, b) = b - \{\mathtt{x}\}$. Then the function FV given by recursion satisfies:

$$\mathrm{FV}(\mathtt{x}) = \{\mathtt{x}\}, \quad \mathrm{FV}(t\,r) = \mathrm{FV}(t) \cup \mathrm{FV}(r), \quad \mathrm{FV}(\lambda\mathtt{x}.\, s) = \mathrm{FV}(s) - \{\mathtt{x}\}.$$

The finite set $\mathrm{FV}(r)$ is called the set of **free variables** occurring in r. A term t is **closed** if $\mathrm{FV}(t) = \emptyset$.

The **bound variables** of a term are given by the following equations:

$$\mathrm{BV}(\mathtt{x}) = \emptyset, \quad \mathrm{BV}(t\,r) = \mathrm{BV}(t) \cup \mathrm{BV}(r), \quad \mathrm{BV}(\lambda\mathtt{x}.\, s) = \{\mathtt{x}\} \cup \mathrm{BV}(s).$$

The set of **sub-terms** of a term r is the set defined by:

$$\mathrm{sub}(\mathtt{x}) = \{\mathtt{x}\}, \qquad \mathrm{sub}(\lambda\mathtt{x}.\, s) = \{\lambda\mathtt{x}.\, s\} \cup \mathrm{sub}(s),$$
$$\mathrm{sub}(t\,r) = \mathrm{sub}(t) \cup \mathrm{sub}(r) \cup \{t\,r\}.$$

1.2 Occurrences

Consider the term $r = \mathtt{x}\,\mathtt{x}$. The variable $\mathtt{x}$ occurs twice in r. In order to reason about such situations, we shall give a formal definition of the notion of occurrence of a sub-term or variable, in terms of a path π telling us how to descend into a term. The reader who has an intuitive grasp of the notion of occurrence may wish to skip its formalisation...

The set of **paths** is $\{R, L, *\}^{<\omega}$, the set of finite sequences of Rs, Ls and $*$s. For each term t we define path(t), the set of **paths in** t, by

$$\mathrm{path}(\mathtt{x}) = \big\{\langle\rangle\big\} \qquad \mathrm{path}(\lambda\mathtt{x}.\, s) = \big\{\langle *, \overline{\sigma}\rangle \;\big|\; \langle\overline{\sigma}\rangle \in \mathrm{path}(s)\big\} \cup \big\{\langle\rangle\big\}$$
$$\mathrm{path}(t\,r) = \big\{\langle L, \overline{\sigma}\rangle \;\big|\; \langle\overline{\sigma}\rangle \in \mathrm{path}(t)\big\} \cup \big\{\langle R, \overline{\sigma}\rangle \;\big|\; \langle\overline{\sigma}\rangle \in \mathrm{path}(r)\big\} \cup \big\{\langle\rangle\big\}$$

We define a function occ such that if $t \in \mathcal{T}$ and $\sigma \in \mathrm{path}(t)$, then $\mathrm{occ}(t, \sigma)$ is a sub-term of t.

$$\mathrm{occ}(t, \langle\rangle) = t$$
$$\mathrm{occ}(t\,r, \langle L, \overline{\sigma}\rangle) = \mathrm{occ}(t, \langle\overline{\sigma}\rangle) \qquad \mathrm{occ}(t\,r, \langle R, \overline{\sigma}\rangle) = \mathrm{occ}(r, \langle\overline{\sigma}\rangle)$$
$$\mathrm{occ}(\lambda\mathtt{x}.\, s, \langle *, \overline{\sigma}\rangle) = \mathrm{occ}(s, \langle\overline{\sigma}\rangle)$$

A path σ is **an occurrence of s in r** if $\mathrm{occ}(r, \sigma) = s$.

When it is convenient (i.e., nearly always) we shall abuse our terminology, confusing a sub-term with a particular sub-term occurrence. E.g., referring to an "occurrence s in r", rather than a "sub-term s of r and some occurrence σ of s in r".

Exercise 1.1 *Refine the notions of free and bound variables of a term to free occurrences and bound occurrences of a variable in a term.*

Suppose that σ is a path in r, that τ is a path in s, where s is the sub-term at σ in r, and t is the sub-term at τ in s. *Show that the concatenation $\sigma\tau$ is a path in r, and that t is the sub-term of r at $\sigma\tau$.*

Given a bound occurrence ρ of a variable $\mathtt{x}$ in r, the **binder** of ρ is the longest initial sub-sequence σ of ρ, such that $\mathrm{occ}(r, \sigma)$ is the form $\lambda\mathtt{x}.\, s$.

Exercise 1.2 *Give a definition of the binder of a bound variable occurrence by recursion on the term r.*

1.3 Alpha Equivalence

One of the most basic operations we need to consider is the replacement of a variable $\mathtt{y}$ in a term r by another term s. This operation is called **substitution**, and is denoted $r[s/\mathtt{y}]$.

Unfortunately, there is an immediate complication in the definition of this. Consider the terms $r = \lambda\mathtt{x}.\, \mathtt{x}\,\mathtt{y}$ and $s = \mathtt{x}\,\mathtt{x}$. Then simply replacing $\mathtt{y}$ by s in r gives the term $r' = \lambda\mathtt{x}.\, \mathtt{x}\,(\mathtt{x}\,\mathtt{x})$. The free variable $\mathtt{x}$ in s has been 'captured' by the λ-abstraction, making it a bound variable in r'. This is undesirable for our purposes.

The solution is to first rename the bound variable $\mathtt{x}$ in r to some other variable, before carrying out the replacement. E.g., renaming $\mathtt{x}$ to $\mathtt{z}$ in r gives the term $\lambda\mathtt{z}.\, \mathtt{z}\,\mathtt{y}$. Then replacing $\mathtt{y}$ by $s = \mathtt{x}\,\mathtt{x}$ gives $\lambda\mathtt{z}.\, \mathtt{z}\,(\mathtt{x}\,\mathtt{x})$. The variable $\mathtt{x}$ is free in this term, as desired.

The formalisation of the 'renaming of bound variables in a term' is called **α-equivalence**. Two terms r_1 and r_2 are α-equivalent ($r_1 \equiv_\alpha r_2$) if one can be obtained from the other by (a sensible) renaming of bound variables.

We could define α-equivalence in terms of a step-by-step renaming of bound variables in a term. Instead, we define α-equivalence in terms of occurrences. Two terms r_1 and r_2 are α-equivalent if the following hold:

1. $\mathrm{path}(r_1) = \mathrm{path}(r_2)$.

2. The sets of free variable occurrences in r_1 and r_2 are equal. The sets of bound variable occurrences in r_1 and r_2 are equal.

3. If σ is a free variable occurrence in r_1 (and by 2., in r_2 also), then $\mathrm{occ}(r_1, \sigma) = \mathrm{occ}(r_2, \sigma)$.

4. If σ is a bound variable occurrence in r_1 and r_2, then the binder of σ in r_1 is equal to the binder of σ in r_2.

Exercise 1.3 *Show that α-equivalence is an equivalence relation.*

Let r_1 and r_2 be terms with $\mathrm{path}(r_1) = \mathrm{path}(r_2)$. (1) Take $\sigma \in \mathrm{path}(r_1) = \mathrm{path}(r_2)$ and let s_1 and s_2 be the corresponding sub-terms occurring at σ. *Show that s_1 and s_2 are* either *both variables,* or *both applications,* or *both abstractions. Show that the two parts of 2. in the definition of $\equiv_\alpha$ are equivalent for r_1 and r_2.*

Exercise 1.4 *Give a formal definition of what it means to replace a bound variable $\mathtt{x}$ by $\mathtt{y}$ in a term r. Give a definition of α-equivalence in terms of this replacement, and prove that it gives the same relation as our definition.* You must be careful to avoid variable capture (e.g., avoid renaming $\lambda\mathtt{x}.\,\lambda\mathtt{y}.\,\mathtt{x}$ to $\lambda\mathtt{x}.\,\lambda\mathtt{x}.\,\mathtt{x}$).

The exercise below shows that every term has plenty of nice α-equivalents.

Exercise 1.5 A term t is **regular** if it satisfies the condition that for every variable $\mathtt{x}$, there is at most one occurrence in t of a sub-term in the form $\lambda\mathtt{x}.\,s$, and no variable is both free and bound in t.

Let t_0 be a term, and let S be a finite set of variables (more generally, such that $\mathcal{V} - S$ is infinite). *Show that there is a regular $t \equiv_\alpha t_0$ such that no member of S is bound in t.*

When it is convenient, we consider the set $\mathcal{T}/{\equiv_\alpha}$ of α-equivalence classes of terms, rather than the set $\mathcal{T}$ of terms. Often we will not distinguish between a term and its α-equivalence class, and write such things as $\lambda\mathtt{x}.\,\mathtt{x} = \lambda\mathtt{y}.\,\mathtt{y}$.

Observe that the basic syntactical operations of abstraction and application are well-defined on α-classes, and that we may prove that a predicate holds for all α-classes by structural induction. However, the unique-reading property does not hold for α-classes, as the α-class of a λ-abstraction $\lambda\mathtt{x}.\,s$ does not determine the variable $\mathtt{x}$. Hence, to define a function on α-classes by structural recursion, we must carry out the recursion on terms, and verify that the function on terms leads to a well-defined function on α-classes.

1.4 Substitution

Let r and s be terms, and $\mathtt{y}$ be a variable, such that no variable is both free in s and bound in r, and $\mathtt{y}$ is not bound in r. The substitution $r[s/\mathtt{y}]$ is defined by recursion on r, as follows:

- $\mathtt{y}[s/\mathtt{y}] = s$, and $\mathtt{x}[s/\mathtt{y}] = \mathtt{x}$ for $\mathtt{x} \neq \mathtt{y}$.
- $(t\,r)[s/\mathtt{y}] = (t[s/\mathtt{y}])(r[s/\mathtt{y}])$.

- $(\lambda \mathtt{x}.\, r)[s/\mathtt{y}] = \lambda \mathtt{x}.(r[s/\mathtt{y}])$.

The following exercise shows that the substitution above gives a well defined, and total, operation on α-equivalence classes.

Exercise 1.6 *Show that the paths in $r[s/\mathtt{y}]$ are (a) the paths ρ in r, (b) concatenations $\rho\sigma$ where ρ is a free occurrence of $\mathtt{y}$ in r and σ is a path in s. Express the sub-term at a path in $r[s/\mathtt{y}]$ in terms of sub-terms at paths in r and s.*

Show that if $r_1 \equiv_\alpha r_2$ and $s_1 \equiv_\alpha s_2$ are such that $r_1[s_1/\mathtt{y}]$ and $r_2[s_2/\mathtt{y}]$ are both defined, then these two terms are α-equivalent.

Show that for any r, s and $\mathtt{y}$, there is $r' \equiv_\alpha r$ such that $r'[s/\mathtt{y}]$ is defined.

Exercise 1.7 *Show that if a variable $\mathtt{x}'$ is neither free nor bound in s', then $s \equiv_\alpha s'$ iff $\lambda \mathtt{x}.\, s \equiv_\alpha \lambda \mathtt{x}'.\, s'[\mathtt{x}'/\mathtt{x}]$.*

Exercise 1.8 *Formulate a notion of **simultaneous substitution**, carrying out several substitutions at once: $s[r_1/\mathtt{x}_1 \ldots r_m/\mathtt{x}_m]$.* Note that in general $s[r_1/\mathtt{x}_1, r_2/\mathtt{x}_2] \neq s[r_1/\mathtt{x}_1][r_2/\mathtt{x}_2]$.

1.5 Beta Conversion

The intuition behind the syntactical constructs of abstraction and application, is that $\lambda \mathtt{x}.\, s$ is the function F defined by $F\,\mathtt{x} = s$ (i.e., $F(r)$ is given by replacing $\mathtt{x}$ by r in s), and that $t\,r$ is the result of applying the function t to argument r. This intuition is captured by the notion of β-reduction.

A **redex** is a term in the form $(\lambda \mathtt{x}.\, s)\, r$. The **basic β-reduction** relation is given by $(\lambda \mathtt{x}.\, s)\, r \longrightarrow_b s'[r/\mathtt{x}]$, whenever $s' \equiv_\alpha s$ is such that $s'[r/\mathtt{x}]$ is defined.

The **one step β-reduction** relation $\longrightarrow_{\beta 1}$ is given by allowing a basic β-reduction of redex occurrence. In other words, $\longrightarrow_{\beta 1}$ is the least relation containing $\longrightarrow_b$ and closed under

$$\frac{t \longrightarrow_{\beta 1} t'}{t\,r \longrightarrow_{\beta 1} t'\,r}, \qquad \frac{r \longrightarrow_{\beta 1} r'}{t\,r \longrightarrow_{\beta 1} t\,r'}, \qquad \frac{s \longrightarrow_{\beta 1} s'}{\lambda \mathtt{x}.\, s \longrightarrow_{\beta 1} \lambda \mathtt{x}.\, s'}.$$

The n **step β-reduction** relations $\longrightarrow_{\beta n}$ are given inductively by $r \longrightarrow_{\beta 0} r$ and, if $r \longrightarrow_{\beta n} r' \longrightarrow_{\beta 1} r''$, then $r \longrightarrow_{\beta n+1} r''$. **$\beta$-reduction** $\longrightarrow_\beta$ is the least pre-order containing $\longrightarrow_{\beta 1}$, so that $r \longrightarrow_\beta r'$ iff there is n such that $r \longrightarrow_{\beta n} r'$. **$\beta$-conversion** ($\equiv_\beta$) is the least equivalence relation containing $\longrightarrow_{\beta 1}$.

Exercise 1.9 *For each of the reduction relations $\longrightarrow_\bullet$, give an analogous definition of $\longrightarrow_\bullet$ as a relation on α-equivalence classes. Show that you have made the correct definition, by showing that for α-classes r and r', the following are equivalent (a) $r \longrightarrow_\bullet r'$, (b) there are $r_0 \in r$ and $r'_0 \in r'$ such that $r_0 \longrightarrow_\bullet r'_0$, and (c) for every $r_0 \in r$ there is $r'_0 \in r'$ such $r_0 \longrightarrow_\bullet r'_0$.*

Show that (a) if $s \longrightarrow_{\beta 1} s'$ then $s[r/\mathtt{x}] \longrightarrow_{\beta 1} s'[r/\mathtt{x}]$, and (b) if $r \longrightarrow_\beta r'$, then $s[r/\mathtt{x}] \longrightarrow_\beta s[r'/\mathtt{x}]$.

A term is called **normal** if no redexes occur in it. A term is **weakly normalising** (WN) if it β-reduces to a normal term. A term r is **strongly normalising** (SN) if there is no infinite sequence $r = r_0 \longrightarrow_\beta r_1 \longrightarrow_\beta \cdots$. Given a term t, we define $\mathrm{sn}(t) = \sup\{n \mid \exists t' \text{ s.t. } t \longrightarrow_{\beta n} t'\} \in \mathcal{N} \cup \{\infty\}$.

Exercise 1.10 *Show that every strongly normalising term is weakly normalising, but not conversely. Show that $\mathrm{sn}(t) < \infty$ iff t is SN.*[1]

1.6 Some Examples—Recursive Functions

We give some examples of terms, and their behaviour under reduction.

The **Church Numerals** are an encoding of the natural numbers, given by

$$\ulcorner n \urcorner = \lambda \mathtt{f}.\, \lambda \mathtt{x}.\, \mathtt{f}^n(\mathtt{x})$$

where $\mathtt{f}^n(\mathtt{x})$ indicates the usual iterated application: $\mathtt{f}^0(\mathtt{x}) = \mathtt{x}$, $\mathtt{f}^{n+1}(\mathtt{x}) = \mathtt{f}\,(\mathtt{f}^n(\mathtt{x}))$. A term F **encodes** a function f if $F \ulcorner i \urcorner \longrightarrow_\beta \ulcorner f(i) \urcorner$ for each n.[2] More generally, F **encodes** a function f of N arguments if

$$F \ulcorner i_1 \urcorner \ldots \ulcorner i_N \urcorner \longrightarrow_\beta \ulcorner f(i_1 \ldots i_N) \urcorner$$

for all $i_1 \ldots i_N$.

The successor function $n \mapsto n{+}1$ is encoded by

$$S = \lambda \mathtt{n}.\, \lambda \mathtt{f}.\, \lambda \mathtt{x}.\, \mathtt{f}\,(\mathtt{n}\,\mathtt{f}\,\mathtt{x}),$$

so that $S \ulcorner n \urcorner \longrightarrow_\beta \ulcorner n{+}1 \urcorner$. Addition is given by

$$\mathrm{add} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \lambda \mathtt{f}.\, \lambda \mathtt{x}.\, \mathtt{m}\,\mathtt{f}\,(\mathtt{n}\,\mathtt{f}\,\mathtt{x}),$$

so that $\mathrm{add} \ulcorner n \urcorner \ulcorner m \urcorner \longrightarrow_\beta \ulcorner n + m \urcorner$, and multiplication by

$$\mathrm{mult} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \lambda \mathtt{f}.\, \mathtt{m}\,(\mathtt{n}\,\mathtt{f}).$$

[1] Note that one direction of this is not constructive. Everything else in these notes is constructive.

[2] By theorem 1.18 (Church-Rosser), a term encodes at most one function.

There is a **zero test**

$$?_0 = \lambda \mathtt{n}.\, \lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathtt{n}\, (\lambda \mathtt{z}.\, \mathtt{y})\, \mathtt{x}$$

with the property that $?_0 \ulcorner 0 \urcorner\, r\, s \longrightarrow_\beta r$ and $?_0 \ulcorner n{+}1 \urcorner\, r\, s \longrightarrow_\beta s$.

There is the **Church pairing**, with

$$\mathrm{pair} = \lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \lambda \mathtt{z}.\, \mathtt{z}\, \mathtt{x}\, \mathtt{y} \qquad \pi_1 = \lambda \mathtt{p}.\, \mathtt{p}\, (\lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathtt{x}) \qquad \pi_2 = \lambda \mathtt{p}.\, \mathtt{p}\, (\lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathtt{y})$$

so that

$$\pi_1\, (\mathrm{pair}\, r\, s) \longrightarrow_\beta r \quad \text{and} \quad \pi_2\, (\mathrm{pair}\, r\, s) \longrightarrow_\beta s.$$

This enables us to encode **primitive recursion**. Let

$$\mathrm{step} = \lambda \mathtt{p}.\, \mathtt{p}\, \big(\lambda \mathtt{x}.\, \lambda \mathtt{y}.\, \mathrm{pair}\, (S\, \mathtt{x})\, (\mathtt{f}\, x\, y)\big)$$

so that $\mathrm{step}\, (\mathrm{pair} \ulcorner n \urcorner\, a) \longrightarrow_\beta \mathrm{pair} \ulcorner n{+}1 \urcorner\, (\mathtt{f}\, a)$. Defining

$$\mathrm{rec}_p = \lambda \mathtt{n}.\, \mathtt{n}\, \mathrm{step}\, (\mathrm{pair} \ulcorner 0 \urcorner\, \mathtt{x})$$

we have

$$\mathrm{rec}_p \ulcorner 0 \urcorner \longrightarrow_\beta \mathrm{pair} \ulcorner 0 \urcorner\, \mathtt{x}$$

and, if $\mathrm{rec}_p \ulcorner n \urcorner \longrightarrow_\beta \mathrm{pair} \ulcorner n \urcorner\, a$, then

$$\mathrm{rec}_p \ulcorner n{+}1 \urcorner \longrightarrow_\beta \mathrm{pair} \ulcorner n{+}1 \urcorner\, (\mathtt{f} \ulcorner n \urcorner\, a).$$

Letting $\mathrm{rec} = \lambda \mathtt{f}.\, \lambda \mathtt{x}.\, \lambda \mathtt{n}.\, \pi_1(\mathrm{rec_p}\, \mathtt{f}\, \mathtt{x}\, \mathtt{n})$, we have that

$$\mathrm{rec}\, f\, a \ulcorner 0 \urcorner \equiv_\beta a \qquad \mathrm{rec}\, f\, a \ulcorner n{+}1 \urcorner \equiv_\beta f \ulcorner n \urcorner\, (\mathrm{rec}\, f\, a \ulcorner n \urcorner).$$

Not every term has a normal form. For example, letting

$$\Omega = (\lambda \mathtt{x}.\, \mathtt{x}\, \mathtt{x})(\lambda \mathtt{x}.\, \mathtt{x}\, \mathtt{x}),$$

we have that $\Omega \longrightarrow_{\beta 1} \Omega \longrightarrow_{\beta 1} \cdots$. Some terms without normal forms are useful, as the example following shows. Programming languages usually allow recursive function definitions: $F\, \mathtt{x} = t$, where t is some expression containing both $\mathtt{x}$ and F. In λ-calculus notation, $F = \lambda \mathtt{x}.\, t$. Of course, we cannot solve this up to equality; the next best thing is to find a term F such that $F \longrightarrow_\beta \lambda \mathtt{x}.\, t$. More generally, letting $G = \lambda F.\, \lambda \mathtt{x}.\, t$, it suffices to find Y_G such that $Y_G \longrightarrow_\beta G\, Y_G$. For example, take

$$Y_G = \big(\lambda \mathtt{x}.\, G\, (\mathtt{x}\, \mathtt{x})\big)\big(\lambda \mathtt{x}.\, G\, (\mathtt{x}\, \mathtt{x})\big).$$

Exercise 1.11 *Check all claims made above.*

Exercise 1.12 *Find a* ***fixed-point combinator*** *Y that β-reduces to the term $\lambda\mathtt{f}.\,\mathtt{f}\,(Y\,\mathtt{f})$.* (Given Y, we could have taken the Y_G above to be $Y\,G$. You can also go the other way round, and find G such that $Y = Y_G$ has the required property.)

Using Y, we can encode minimisation. If f is a function on the natural numbers such that $f(i) = 0$ for some i, then μf is the least such i. Similarly, let $\mu_{\geqslant n} f$ be the least $i \geqslant n$ such that $f(i) = 0$, if such exists. This is given recursively by:

$$\mu_{\geqslant n} f = \begin{cases} n & \text{if } f(n) = 0 \\ \mu_{\geqslant n+1} f & \text{if } f(n) \neq 0 \end{cases}$$

Hence, we want a term $\mu_{\geqslant}$, such that $\mu_{\geqslant}\, n\, f \longrightarrow_\beta\ ?_0\,(f\,n)\,n\,(\mu_{\geqslant}\,(S\,n)\,f)$. Letting

$$\mu_{\geqslant} = Y\,\big(\lambda\mathtt{M}.\,\lambda\mathtt{n}.\,\lambda\mathtt{f}.\,?_0\,(\mathtt{f}\,\mathtt{n})\,\mathtt{n}\,(\mathtt{M}\,(S\,\mathtt{n})\,\mathtt{f})\big)$$

does the trick, and then let $\mu = \mu_{\geqslant}\ulcorner 0 \urcorner$. With these definitions, if F encodes a function f, and f has a zero, then

$$\mu\, F \longrightarrow_\beta \ulcorner \mu f \urcorner.$$

Exercise 1.13 The **total recursive functions** are the least set of functions (of arbitrary arity) on the natural numbers, that

1. Contains the constant functions, the successor function and the projections $(x_1 \dots x_m) \mapsto x_i$ (whenever $1 \leqslant i \leqslant m$).
2. Is closed under composition: $h(\overline{x}) = f(g_1(\overline{x}) \dots g_n(\overline{x}))$.
3. Is closed under primitive recursion: $h(\overline{x}, 0) = f(\overline{x})$, $h(\overline{x}, n+1) = g(\overline{x}, n, f(\overline{x}, n))$.
4. Is closed under minimisation, whenever this gives a *total* function: if for each $\overline{x}$ there is n such that $f(\overline{x}, n) = 0$, then take $g(\overline{x})$ to be the least such n.

Show that every total recursive function can be encoded by a λ-term.

It was conjectured by Church that the computable functions on the natural numbers are precisely those that can be encoded by a λ-term. This is the famous **Church's Thesis**. The exercise above is one half of showing that the functions encodable in Λ coincide with the recursive functions. Unsurprisingly, the functions on the natural numbers encodable in Λ coincide with the functions computable in lots of other mathematical models of computability.

Theorem 1.14 *The predicate of being weakly normalising is not decidable.*

PROOF SKETCH: Suppose that WN were decidable. Then we could define a computable function F on λ-terms such that $F(t)$ is the normal form of t if there is one, and $F(t) = \lambda\mathtt{x}.\,\mathtt{x}$ otherwise. Then F would be a total computable function majorising every total recursive function, which is not possible. □

1.7 Church-Rosser (Confluence) Property

Our reduction relation is not deterministic—given a term r, it may contain several redexes, and hence several different ways of reducing it.

However, there is a certain determinism to the intuition behind our calculus—function application gives a unique value. This is expressed locally in the fact that there is (up to $\equiv_\alpha$) only one way of reducing a given redex. It can also be expressed globally in the confluence, or Church-Rosser, property: if $r \longrightarrow_\beta r_1$ and $r \longrightarrow_\beta r_2$, then there is r' such that $r_1 \longrightarrow_\beta r'$ and $r_2 \longrightarrow_\beta r'$. In particular, a term has at most one normal form.

The proof of Church-Rosser is based on a quite simple idea—if r contains two redexes, then the reduction of one redex does not stop the reduction of the other redex, and it does not matter in what order the reductions are carried out. This makes it quite simple to show that if $r \longrightarrow_{\beta 1} r_1, r_2$, then there is r' such that $r_1, r_2 \longrightarrow_\beta r'$. However, note the $\beta 1$- and β- reductions in this. To extend this result to confluence for β-reduction, we must be careful to avoid an infinite regress. The proof we give (due to Hyland) replaces $\longrightarrow_{\beta 1}$ with a 'parallel' reduction, which simultaneously reduces any collection of redexes in a term.

Let R be a relation, and $\leqslant$ be the least pre-order containing R. R is **confluent**, if whenever $a\ R\ a_1$ and $a\ R\ a_2$, there is a' such that $a_1\ R\ a'$ and $a_2\ R\ a'$. R is **weakly confluent**, if whenever $a\ R\ a_1, a_2$, there is a' such that $a_1, a_2 \leqslant a'$.

Exercise 1.15 Let R be a relation and $\leqslant$ the least pre-order containing R. *Show that if R is confluent, then so is $\leqslant$. Find a weakly confluent R such that $\leqslant$ is not confluent.*

The **parallel reduction** relation, $\longrightarrow_p$, is the relation on α-classes given inductively by:

$$\frac{}{\mathtt{x} \longrightarrow_p \mathtt{x}} \qquad \frac{t \longrightarrow_p t' \quad r \longrightarrow_p r'}{t\,r \longrightarrow_p t'\,r'} \qquad \frac{s \longrightarrow_p s'}{\lambda\mathtt{x}.\,s \longrightarrow_p \lambda\mathtt{x}.\,s'}$$

$$\frac{s \longrightarrow_p s' \quad r \longrightarrow_p r'}{(\lambda \mathtt{x}.\, s')\, r \longrightarrow_p s'[r'/\mathtt{x}]}$$

The lemma below follows easily from the definitions of the various reduction relations.

Lemma 1.16 *If $r \longrightarrow_{\beta 1} r'$ then $r \longrightarrow_p r'$. If $r \longrightarrow_p r'$, then $r \longrightarrow_\beta r'$. The least pre-order containing $\longrightarrow_p$ is $\longrightarrow_\beta$. If $s \longrightarrow_p s'$ and $r \longrightarrow_p r'$, then $s[r/\mathtt{x}] \longrightarrow_p s'[r'/\mathtt{x}]$.* □

Lemma 1.17 *$\longrightarrow_p$ is confluent.*

PROOF: We show that if t, t_1 and t_2 are α-classes, with $t \longrightarrow_p t_1$ and $t \longrightarrow_p t_2$, then, then there is t' such that $t_1 \longrightarrow_p t'$ and $t_2 \longrightarrow_p t'$. We perform the construction by induction on t.

The non-trivial step is for t in the form $(\lambda \mathtt{x}.\, s)\, r$. Then (for $i = 1, 2$) there are s_i and r_i such that $s \longrightarrow_p s_i$, $r \longrightarrow_p r_i$, and either $t_i = (\lambda \mathtt{x}.\, s_i)\, r_i$ or $t_i = s_i[r_i/\mathtt{x}]$. By the induction hypothesis, there are r' and s' such that (for $i = 1, 2$) $r_i \longrightarrow_p r'$ and $s_i \longrightarrow_p s'$. Let $t' = s'[r'/\mathtt{x}]$. Then we have that both $(\lambda \mathtt{x}.\, s_i)\, r_i \longrightarrow_p t'$ and $s_i[r_i/\mathtt{x}] \longrightarrow_p t'$, so that $t_1, t_2 \longrightarrow_p t'$.

The other cases are easy: e.g. if t is in the form $s\, r$, but is not a redex, then the t_i are in the form $s_i\, r_i$ where $s \longrightarrow_p s_1, s_2$ and $r \longrightarrow_p r_1, r_2$. Applying the induction hypothesis, there are s' and r' such that $s_1, s_2 \longrightarrow_p s'$ and $r_1, r_2 \longrightarrow_p r'$. Letting $t' = s'\, r'$, we have $t_1, t_2 \longrightarrow_p t'$, as required. □

Theorem 1.18 (Church-Rosser) *The relation $\longrightarrow_\beta$ is confluent (on α-classes).* □

Corollary 1.19 *If a term β-reduces to a normal form, then the normal form is unique (up to $\equiv_\alpha$).* □

Exercise 1.20 (Standardisation) The **standard** reduction relation $\longrightarrow_s$ is given by:

$$\frac{r_1 \longrightarrow_s r_1' \quad \ldots \quad r_m \longrightarrow_s r_m'}{\mathtt{x}\, r_1 \ldots r_m \longrightarrow_s \mathtt{x}\, r_1' \ldots r_m'} \qquad \frac{s[r_0/\mathtt{x}]\, r_1 \ldots r_m \longrightarrow_s t}{(\lambda \mathtt{x}.\, s)\, r_0\, r_1 \ldots r_m \longrightarrow_s t}$$

$$\frac{s \longrightarrow_s s' \quad r_1 \longrightarrow_s r_1' \quad \ldots \quad r_m \longrightarrow_s r_m'}{(\lambda \mathtt{x}.\, s)\, r_1 \ldots r_m \longrightarrow_s (\lambda \mathtt{x}.\, s)\, r_1' \ldots r_m'}$$

Show that if $r \longrightarrow_s r'$, then $r \longrightarrow_\beta r'$, and give a characterisation of the reduction sequences obtained (in terms of what redexes are reduced in what *order). Show that if r' is normal, then $r \longrightarrow_\beta r'$ by a reduction sequence that always reduces the leftmost redex (appropriately defined).*

Show that $\longrightarrow_s$ is closed under the following:

$$\frac{s \longrightarrow_s s' \quad r \longrightarrow_s r'}{s[r/\mathtt{x}] \longrightarrow_s s'[r'/\mathtt{x}]} \qquad \frac{t \longrightarrow_s (\lambda \mathtt{x}.\, s')\, r'}{t \longrightarrow_s s'[r'/\mathtt{x}]} \qquad \frac{t \longrightarrow_s t' \longrightarrow_{\beta 1} t''}{t \longrightarrow_s t''}$$

and that $\longrightarrow_s$ and $\longrightarrow_\beta$ are the same relation.

Conclude that if $r \longrightarrow_\beta r'$, then r reduces to r' with a reduction sequence in some special form. In particular, if $r \longrightarrow_\beta r'$, and r' is normal, then leftmost reduction reduces r to r'. These are the **standardisation theorems**.

The notion of a term F encoding a (total) function f, is extended to F **encoding** a partial function f, by requiring that if $f(n)$ is undefined, then $F \ulcorner n \urcorner$ does not β-reduce to a λ-abstraction.[3]

In exercise 1.13, we referred to only *total* recursive functions. This is because we didn't have results useful for showing partiality as encoded above. The standardisation results in the previous exercise are such results.

Exercise 1.21 The **partial recursive functions** are given by removing the condition (in exercise 1.13) that functions given by minimisation are total. *Show that every partial recursive function can be encoded by a λ-term.* You will need to take care of the fact that mathematical functions are strict, in that $f(x_1 \ldots x_m)$ is undefined if any of the x_i are, while the obvious encodings in Λ are not strict.

2 Simply Typed Lambda Calculus

So far, we have dealt with the *untyped* λ-calculus. There is no restriction on how we can use objects—for example, any value can be used as a function. In real life, this is undesirable; what does it mean to apply the number 5 to the function cos? Type systems are one way of (statically) ensuring that operations are only applied to appropriate objects. We shall introduce a type system for the λ-calculus, obtaining the **simply typed λ-calculus**, $\lambda^{\rightarrow}$ for short.

2.1 Syntax

The syntax of $\lambda^{\rightarrow}$ is presented in stages.

[3] Perhaps not having a normal form would seem more intuitive. However, the stronger condition given is easier to work with.

1. The types are defined.
2. The pre-terms are defined. Pre-terms are essentially terms of Λ, and include things that we do not wish to consider in $\lambda^{\rightarrow}$, such as $\lambda \mathtt{x}{:}o.\, \mathtt{x}\,\mathtt{x}$.
3. A system of typing judgements is given, that identifies the terms—those pre-terms that we wish to consider in $\lambda^{\rightarrow}$.

We take an arbitrary non-empty set $\mathcal{G}$ of **ground** types. For concreteness and simplicity, we shall take $\mathcal{G} = \{o\}$; there is no reason why some other (larger) set should not be used. Types are then given by the following grammar:

$$\tau := \mathcal{G} \mid \tau \rightarrow \tau.$$

In order to avoid ambiguity in $\lambda^{\rightarrow}$, we decorate λ-abstractions with types. The grammar for **pre-terms** of $\lambda^{\rightarrow}$ is:

$$\mathcal{T} := \mathcal{V} \mid \mathcal{T}\,\mathcal{T} \mid \lambda\mathcal{V}{:}\tau.\, \mathcal{T}.$$

The notions such as α-equivalence, substitution and reduction can be carried across more or less verbatim from Λ to $\lambda^{\rightarrow}$. The definition of α-equivalence must be modified to ensure that the type decorations are preserved, e.g., so that $\lambda\mathtt{x}{:}A.\,\mathtt{x} \equiv_\alpha \lambda\mathtt{y}{:}B.\,\mathtt{y}$ if and only if $A = B$. Our proof of the Church-Rosser theorem still works for the pre-terms of the typed calculus.

A **context** is a function mapping a finite set of variables into the set of types. Contexts will be written in the form $\mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m$. They should be read as a typing declaration: the context $\mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m$ indicates that each variable $\mathtt{x}_i$ will be considered to have the corresponding type A_i. We present a ternary relation $\Gamma \vdash s : B$ on contexts Γ, pre-terms s and types B by the following clauses:

$$\frac{}{\mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m \vdash \mathtt{x}_i : A_i} \quad \text{for } 1 \leqslant i \leqslant m$$

$$\frac{\Gamma \vdash f : A \rightarrow B \qquad \Gamma \vdash r : A}{\Gamma \vdash f\,r : B}$$

$$\frac{\Gamma\,\mathtt{x}{:}A \vdash s : B}{\Gamma \vdash \lambda\mathtt{x}.\, s : A \rightarrow B}$$

A pre-term s is said to be a **term** (of type B in context Γ) if $\Gamma \vdash s : B$.

We also define the relation $\Gamma \vdash s : B$ for α-classes s of pre-terms, using identical clauses.

Exercise 2.1 *Show that the relations $\vdash$ for pre-terms, and for α-classes, are related as follows: an α-class s has $\Gamma \vdash s : B$ derivable if and only if there is $s_0 \in s$ such that $\Gamma \vdash s_0 : B$ is derivable.*

2.2 Basic Properties

Lemma 2.2 (Renaming) *If* $\Gamma\,\mathtt{z}{:}C\,\Delta \vdash s : B$ *for some* α*-class* s*, and* $\mathtt{z}'$ *is not in the context, then* $\Gamma\,\mathtt{z}'{:}C\,\Delta \vdash s[\mathtt{z}'/\mathtt{z}] : B$.

Proof: We in fact show that simultaneous renaming is possible, in that if $\mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m \vdash s : B$, and $\mathtt{x}'_1 \ldots \mathtt{x}'_m$ is a sequence of distinct variables, then $\mathtt{x}'_1{:}A_1 \ldots \mathtt{x}'_m{:}A_m \vdash s[\mathtt{x}'_1/\mathtt{x}_1 \ldots \mathtt{x}'_m/\mathtt{x}_m] : B$.

We show this by induction on derivations. We do the case of a derivation ending

$$\frac{\mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m\,\mathtt{x}{:}A \vdash s : B}{\mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m \vdash \lambda\mathtt{x}{:}A.\,s : A \to B.}$$

Let $\mathtt{x}'$ be a variable distinct from the $\mathtt{x}'_i$. By the induction hypothesis,

$$\mathtt{x}'_1{:}A_1 \ldots \mathtt{x}'_m{:}A_m\,\mathtt{x}'{:}A \vdash s[\mathtt{x}'_1/\mathtt{x}_1 \ldots \mathtt{x}'_m/\mathtt{x}_m, \mathtt{x}'/\mathtt{x}] : B$$

and hence

$$\mathtt{x}'_1{:}A_1 \ldots \mathtt{x}'_m{:}A_m \vdash \lambda\mathtt{x}'{:}A.\big(s[\mathtt{x}'_1/\mathtt{x}_1 \ldots \mathtt{x}'_m/\mathtt{x}_m, \mathtt{x}'/\mathtt{x}]\big) : B$$

are derivable. The equation

$$\lambda\mathtt{x}'{:}A.\big(s[\mathtt{x}'_1/\mathtt{x}_1 \ldots \mathtt{x}'_m/\mathtt{x}_m, \mathtt{x}'/\mathtt{x}]\big) = (\lambda\mathtt{x}{:}A.\,s)[\mathtt{x}'_1/\mathtt{x}_1 \ldots \mathtt{x}'_m/\mathtt{x}_m]$$

holds, and the induction step follows. □

Lemma 2.3 (Weakening) *Let* s *be an* α*-class such that* $\Gamma\Delta \vdash s : B$*. Let* $\mathtt{z}$ *be a variable not in* $\Gamma\Delta$*. Then* $\Gamma\,\mathtt{z}{:}C\,\Delta \vdash s : B$

Proof: By the renaming lemma, it suffices to show that there is $\mathtt{z}$ such that $\Gamma\,\mathtt{z}{:}C\,\Delta \vdash s : B$. This can be shown by an easy induction on derivations. □

Lemma 2.4 (Uniqueness of Derivations) *If* s *is a pre-term, with* $\Gamma \vdash s : B$*, then the derivation of this fact is unique.* □

Lemma 2.5 (Unicity of Typing) *If* s *is an* α*-class, with* $\Gamma \vdash s : B$ *and* $\Gamma \vdash s : B'$*, then* $B = B'$*.* □

Exercise 2.6 *Prove the two lemmas above.*

Lemma 2.7 (Substitution) *Let* r *and* s *be* α*-classes such that* $\Gamma \vdash r : A$ *and* $\Gamma\,\mathtt{x}{:}A\,\Delta \vdash s : B$*. Then* $\Gamma\Delta \vdash s[r/\mathtt{x}] : B$*.* □

Exercise 2.8 *Prove it.* Use induction on a derivation of $\Gamma\,\mathtt{x}{:}A\,\Delta \vdash s : B$.

Lemma 2.9 (Subject Reduction) *Let s be an α-class with $\Gamma \vdash s : B$, and $s \longrightarrow_\beta s'$. Then $\Gamma \vdash s' : B$ also.*

PROOF: The case when $s \longrightarrow_b s'$ reduces to the substitution lemma above. Closing under the term constructs of application and abstraction shows that the lemma holds for $s \longrightarrow_{\beta 1} s'$. An induction shows that the lemma holds for $\longrightarrow_\beta$ also. $\square$

2.3 Models

We give a notion of model of $\lambda^\to$. What is given here is far from the most general notion of model possible. A **model** is a map $\mathcal{M}$ assigning a set $\mathcal{M}(o)$ to each ground type. We define the **interpretation** $[\![\cdot]\!]$, first for types, and then for terms. Put $[\![o]\!] = \mathcal{M}(o)$, and $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$ (the set of all functions from $[\![A]\!]$ to $[\![B]\!]$).

Given a context $\Gamma = \mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m$, we define a **valuation** σ on Γ to be a function defined on the $\mathtt{x}_i$ such that $\sigma(\mathtt{x}_i) \in [\![A_i]\!]$ for $1 \leqslant i \leqslant m$. Given another variable $\mathtt{x}$ and $a \in [\![A]\!]$, we define $\sigma[\mathtt{x} := a](\mathtt{x}_i) = \sigma(\mathtt{x}_i)$ and $\sigma[\mathtt{x} := a](\mathtt{x}) = a$, so that $\sigma[\mathtt{x} := a]$ is a valuation on $\Gamma\, \mathtt{x}{:}A$.

We define $[\![s]\!]_\sigma \in [\![B]\!]$ whenever $\Gamma \vdash s : B$ and σ is a valuation on Γ, by

$$[\![\mathtt{x}]\!]_\sigma = \sigma(\mathtt{x}) \qquad [\![t\, r]\!]_\sigma = [\![t]\!]_\sigma\big([\![r]\!]_\sigma\big)$$

and, for $a \in [\![A]\!]$,

$$[\![\lambda\mathtt{x}{:}A.\, s]\!]_\sigma(a) = [\![s]\!]_{\sigma[\mathtt{x}:=a]}.$$

The following lemma shows that our semantics gives some meaning to β-reduction, as well as to syntax.

Lemma 2.10 *If $\Gamma \vdash r : A$ and $\Gamma\, \mathtt{x}{:}A \vdash s : B$, then*

$$[\![s[r/\mathtt{x}]]\!]_\sigma = [\![s]\!]_{\sigma[\mathtt{x} := [\![r]\!]_\sigma]}.$$

If $\Gamma \vdash s : B$ and $s \longrightarrow_\beta s'$, then $[\![s]\!]_\sigma = [\![s']\!]_\sigma$. $\square$

Exercise 2.11 *Prove the lemma.* Be careful about contexts!

3 Strong Normalisation

The set-theoretic semantics of $\lambda^\to$ of section 2.3 gives a sense in which $\lambda^\to$ meaningful—it formalises the intuition that $A \to B$ is a type of functions, that terms of a function type are functions, and that terms are values of some sort. The semantics gives meaning to β-reduction, by lemma 2.10.

The meaning given to $\lambda^{\to}$ by the semantics is *static* and *external*. Static in the sense that terms are represented by values, rather than by computations; it gives no sense of the term t computing the value $[\![t]\!]$. External in the sense that terms are given meaning in terms of some structure outside of the syntax.

The normalisation theorem proved in this section gives meaning to $\lambda^{\to}$ in a way that is *dynamic* and *internal*, by showing that every term is strongly normalising. This is dynamic in the sense that the meaning of a term is given by a computational process: the reduction of a term t to a normal form v is in a series of steps $t = t_0 \longrightarrow_{\beta 1} t_1 \longrightarrow_{\beta 1} \cdots \longrightarrow_{\beta 1} t_n = v$. These steps can be seen as the successive states of a computer running the program t to reach a value v. Strong normalisation is *internal*, in that it gives meaning to the calculus without reference to another structure.

A common way to prove the termination of some sequence $t_0, t_1, \ldots$ is to find some measure $|t_i| \in \mathcal{N}$ that is strictly decreasing in i. Later, we will carry out such a construction, in order to give a quantitative analysis of strong normalisation for $\lambda^{\to}$. However, such a proof is non-trivial. A first attempt might be to take $|t|$ to be the length of the term t. This fails as reduction of a redex $(\lambda \mathtt{x}.\, s)\, r$ to $s[r/\mathtt{x}]$ can increase length. In the proof of strong normalisation that follows, we avoid this problem by taking into account the meaning of terms: we show that a term $f : A \to B$ is not just SN, but that f maps SN terms of type A to SN terms of type B.

Normalisation for $\lambda^{\to}$ was first proved by Turing, although Gentzen had earlier given similar cut-elimination results for sequent calculi. The proof method we use is due to Tait. Girard extended it to prove normalisation for F, the second order λ-calculus, and since then it has been used to prove normalisation for just about everything.

3.1 Preliminaries

We start by noting some properties of SN, the set of strong normalising terms of Λ. We leave it as an exercise to rewrite these for $\lambda^{\to}$.

Exercise 3.1 *If $t \in \mathrm{SN}$ and $t \longrightarrow_{\beta} t'$, then $t' \in \mathrm{SN}$. If $t' \in \mathrm{SN}$ for all t' such that $t \longrightarrow_{\beta 1} t'$, then $t' \in \mathrm{SN}$. If $t \in \mathrm{SN}$ and t' is a sub-term of t, then $t' \in \mathrm{SN}$.*

Lemma 3.2 *The set* SN *is closed under the following:*

$$\frac{r_1 \quad \dots \quad r_m}{\mathtt{x}\, r_1 \dots r_m} \qquad \frac{s}{\lambda \mathtt{x}.\, s} \qquad \frac{s[r_0/\mathtt{x}]\, r_1 \dots r_m \quad r_0}{(\lambda \mathtt{x}.\, s)\, r_0\, r_1 \dots r_m} \tag{1}$$

PROOF: The first two clauses are more or less trivial. We do the last clause. Recall that $\mathrm{sn}(t)$ is the l.u.b. of the length of the reduction sequences starting from t. Suppose that $s[r_0/\mathtt{x}]\, r_1 \dots r_m \in \mathrm{SN}$ and $r_0 \in \mathrm{SN}$.

We show by induction on the number $\mathrm{sn}\big(s[r_0/\mathtt{x}]\, r_1 \dots r_m\big) + \mathrm{sn}\big(r_0\big)$, that $(\lambda \mathtt{x}.\, s)\, r_0\, r_1 \dots r_m \in \mathrm{SN}$.

Suppose $(\lambda \mathtt{x}.\, s)\, r_0\, r_1 \dots r_m \longrightarrow_{\beta 1} t$. We show that t is SN by considering the possibilities for t.

If $t = s[r_0/\mathtt{x}]\, r_1 \dots r_m$, then $t \in \mathrm{SN}$.

If $t = (\lambda \mathtt{x}.\, s)\, r'_0\, r_1 \dots r_m$ where $r_0 \longrightarrow_{\beta 1} r'_0$, then we get the β-reduction $s[r_0/\mathtt{x}]\, r_1 \dots r_m \longrightarrow_{\beta} s[r'_0/\mathtt{x}]\, r_1 \dots r_m$. This gives the inequalities

$$\mathrm{sn}\big(s[r'_0/\mathtt{x}]\, r_1 \dots r_m\big) \leqslant \mathrm{sn}\big(s[r_0/\mathtt{x}]\, r_1 \dots r_m\big) \quad \text{and} \quad \mathrm{sn}(r'_0) < \mathrm{sn}(r_0).$$

By the induction hypothesis, $t \in \mathrm{SN}$.

Otherwise, the redex reduced is in s or one of $r_1 \dots r_m$, and we have $t = (\lambda \mathtt{x}.\, s')\, r_0\, r'_1 \dots r'_m$ where $s[r_0/\mathtt{x}]\, r_1 \dots r_m \longrightarrow_{\beta 1} s'[r_0/\mathtt{x}]\, r'_1 \dots r'_m$, and applying the induction hypothesis gives $t \in \mathrm{SN}$. □

Exercise 3.3 *Refine the proof of lemma 3.2 to show that if t can be derived by using the clauses of (1) n times, then $\mathrm{sn}(t) \leqslant n$.*

Show that SN *is the least set closed under the three clauses (1)—i.e., that every SN term can be derived using those clauses.*

Exercise 3.4 *Show that if* $\mathtt{x}$ *is free in s, and* $s[r_0/\mathtt{x}]\, r_1 \dots r_m \in \mathrm{SN}$*, then* $(\lambda \mathtt{x}.\, s)\, r_0 \dots r_m \in \mathrm{SN}$.

The λI-calculus Λ_I is given in exactly the same way as Λ, except that in all λ-abstractions $\lambda \mathtt{x}.\, s$, the variable $\mathtt{x}$ is required to be free in s. *Show that every weakly normalising Λ_I term is strongly normalising.* Hint: use the standardisation results of exercise 1.20 and the first part of this exercise.

We wish to make the following definition, recursive in types:

- Red_o is the set of strongly normalising terms of type o, for any ground type o.

- $\mathrm{Red}_{A \to B}$ is the set of terms f of type $A \to B$, such that for any $r \in \mathrm{Red}_A$, we have $f\, r \in \mathrm{Red}_B$.

Unfortunately, this does not quite work for our purposes, as the contexts of f and r might be different. One way of avoiding this issue is to reformulate $\lambda^{\to}$ slightly, so that variables have types pre-assigned, rather than being declared in contexts. Another way is to simply keep track of contexts; we therefore make the following definition of **the sets of reducible terms**:

$$\mathrm{Red}_o = \big\{(\Gamma, r) \bigm| r \in \mathrm{SN},\ \Gamma \vdash r : o\big\}$$

for ground types o, and

$$\mathrm{Red}_{A\to B} = \big\{(\Gamma, f) \bigm| \Gamma \vdash f : A \to B,\ (\Gamma\Delta, f\,r) \in \mathrm{Red}_B\ \forall(\Gamma\Delta, r) \in \mathrm{Red}_A\big\}.$$

Lemma 3.5 *1. If $(\Gamma, s) \in \mathrm{Red}_B$ then $s \in \mathrm{SN}$.*

2. If $\Gamma \vdash \mathtt{x}\,r_1 \ldots r_m : B$, and $r_1, \ldots, r_m \in \mathrm{SN}$, then

$$\big(\Gamma, \mathtt{x}\,r_1 \ldots r_m\big) \in \mathrm{Red}_B.$$

3. If $(\Gamma, s[r_0/\mathtt{x}]\,r_1 \ldots r_m) \in \mathrm{Red}_B$, $\Gamma \vdash r_0 : A$ and $r_0 \in \mathrm{SN}$, then

$$\big(\Gamma, (\lambda\mathtt{x}{:}A.\,s)\,r_0\,r_1 \ldots r_m\big) \in \mathrm{Red}_B.$$

Proof: We prove the properties simultaneously by induction on the type B. For $B = o$, 1. is trivial, and 2. and 3. are properties of SN. Suppose that $B = C \to D$.

For 1., let $\mathtt{z}$ be a variable not in Γ, so that by the induction hypothesis of 2. for C, $(\Gamma\,\mathtt{z}{:}C, \mathtt{z}) \in \mathrm{Red}_C$. As $(\Gamma, s) \in \mathrm{Red}_{C\to D}$, we have $(\Gamma\,\mathtt{z}{:}C, s\,\mathtt{z}) \in \mathrm{Red}_D$, so that by the induction hypothesis of 1. for D, $s\,\mathtt{z} \in \mathrm{SN}$ and hence $s \in \mathrm{SN}$ also.

For 2., take any $(\Gamma\Delta, r_{m+1}) \in \mathrm{Red}_C$. By the IH 1. for C, we have that $r_{m+1} \in \mathrm{SN}$, so that by the IH 2. for D, $(\Gamma\Delta, \mathtt{x}\,r_1 \ldots r_{m+1}) \in \mathrm{Red}_D$. The definition of $\mathrm{Red}_{C\to D}$ now gives $(\Gamma, \mathtt{x}\,r_1 \ldots r_m) \in \mathrm{Red}_B$.

For 3., take any $(\Gamma\Delta, r_{m+1}) \in \mathrm{Red}_C$. Then, $(\Gamma\Delta, s[r_0/\mathtt{x}]\,r_1 \ldots r_{m+1}) \in \mathrm{Red}_D$, so that by the IH, $(\Gamma\Delta, (\lambda\mathtt{x}{:}A.\,s)\,r_0\,r_1 \ldots r_{m+1}) \in \mathrm{Red}_D$. This shows that $(\Gamma, (\lambda\mathtt{x}{:}A.\,s)\,r_0\,r_1 \ldots r_m) \in \mathrm{Red}_B$. □

Exercise 3.6 *Check that if $(\Gamma, s) \in \mathrm{Red}_B$, then $(\Gamma\,\mathtt{x}{:}A, s) \in \mathrm{Red}_B$ also.*

3.2 The Main Induction

We prove that terms of $\lambda^{\to}$ are SN, by showing that they are members of the reducibility sets defined above. The definition of $\mathrm{Red}_{A\to B}$ as the set of those terms which give a function from Red_A to Red_B, means that the sets Red_A form something like a model of $\lambda^{\to}$. Our proof that every term is reducible will be a structural induction bearing a more than passing similarity to the definition of the interpretation of a term in a model.

Lemma 3.7 *Suppose* $\mathtt{y}_1{:}B_1 \ldots \mathtt{y}_n{:}B_n \vdash r : A$. *For any* $(\Gamma, s_j) \in \mathrm{Red}_{B_j}$ *($1 \leqslant j \leqslant n$), we have* $(\Gamma, r[s_1/\mathtt{y}_1 \ldots s_n/\mathtt{y}_n]) \in \mathrm{Red}_A$.

PROOF: We write $\overline{s}/\overline{\mathtt{y}}$ for $s_1/\mathtt{y}_1 \ldots s_n/\mathtt{y}_n$. The proof is by induction on the term r.

For r a variable, the lemma is trivial.

Consider an application $t\,r$ where $r : A$ and $t : A \to B$. The induction hypotheses are that $(\Gamma, t[\overline{s}/\overline{\mathtt{y}}]) \in \mathrm{Red}_{A\to B}$ and $(\Gamma, r[\overline{s}/\overline{\mathtt{y}}]) \in \mathrm{Red}_A$. The definition of $\mathrm{Red}_{A\to B}$ gives $(\Gamma, (t\,r)[\overline{s}/\overline{\mathtt{y}}]) \in \mathrm{Red}_B$, as required.

Consider a λ-abstraction, $\lambda\mathtt{x}{:}A.\, b : A \to B$. Take any $(\Gamma\Gamma', a) \in \mathrm{Red}_A$. For each j, $(\Gamma\Gamma', s_j) \in \mathrm{Red}_{B_j}$, so that by the induction hypothesis,

$$\big(\Gamma\Gamma', b[\overline{s}/\overline{\mathtt{y}}, a/\mathtt{x}]\big) \in \mathrm{Red}_B.$$

Also, $a \in \mathrm{SN}$ as $a \in \mathrm{Red}_A$, so that by 3. of lemma 3.5, we have

$$\big(\Gamma\Gamma', (\lambda\mathtt{x}{:}A.\, b)[\overline{s}/\overline{\mathtt{y}}]\, a\big) \in \mathrm{Red}_B.$$

By the definition of $\mathrm{Red}_{A\to B}$, we get $\big(\Gamma, (\lambda\mathtt{x}{:}A.\, b)[\overline{s}/\overline{\mathtt{y}}]\big) \in \mathrm{Red}_{A\to B}$. □

Theorem 3.8 *Every term of* $\lambda^{\to}$ *is SN.*

PROOF: Given $\mathtt{y}_1{:}B_1 \ldots \mathtt{y}_n{:}B_n \vdash r : A$, take $\Gamma = \mathtt{y}_1{:}B_1 \ldots \mathtt{y}_n{:}B_n$ and $s_j = \mathtt{y}_j$ in the lemma above. This gives $(\Gamma, r) = \big(\Gamma, r[\overline{\mathtt{y}}/\overline{\mathtt{y}}]\big) \in \mathrm{Red}_A$, and so $r \in \mathrm{SN}$. □

Exercise 3.9 Gödel's system T is the following extension of $\lambda^{\to}$: take the ground type to be $\mathcal{N}$, and introduce constructs 0, S and R with the following rules:

$$\frac{}{\Gamma \vdash 0 : \mathcal{N}} \qquad \frac{\Gamma \vdash n : \mathcal{N}}{\Gamma \vdash S(n) : \mathcal{N}}$$

$$\frac{\Gamma \vdash n : \mathcal{N} \quad \Gamma \vdash a : A \quad \Gamma, \mathtt{x}{:}A \vdash r : A}{\Gamma \vdash R(n, a, \mathtt{x}.\, r)}$$

The type $\mathcal{N}$ is intended to represent the natural numbers, with 0 and S being zero and the successor function. The operator R gives definitions by recursion: $R(n, a, \mathtt{x}.\, f\,\mathtt{x})$ is intended to be $f^n(a)$. This is expressed by the following reduction rules:

$$R(0, a, \mathtt{x}.\, r) \longrightarrow a \qquad R(S(n), a, \mathtt{x}.\, r) \longrightarrow r[R(n, a, \mathtt{x}.\, r)/\mathtt{x}].$$

Adapt the proof of theorem 3.8 to prove SN for Gödel's T.

3.3 A Combinatorial Proof

We give another proof of SN for $\lambda^{\rightarrow}$ that is less abstract in the sense that it doesn't require the predicates Red_A. The combinatorial methods we use here are less general—for instance, cannot be applied to Gödel's T—but can be formalised in a logic of first order arithmetic, such as Peano Arithmetic. The technique is in essence that used by Gentzen to prove cut-elimination for LK, the classical sequent calculus. Later, we shall give an even less abstract proof, one that is formalisable in primitive recursive arithmetic.

We shall call a term **inductively normalising** (IN) if it is derivable by the three clauses (1). By lemma 3.2, the inductively normalising terms are strongly normalising.

We shall call a type A **substitutive** if IN is closed under the following clause:

$$\frac{\Gamma \vdash r : A \quad \Gamma\, \mathtt{x}{:}A\, \Delta \vdash s : B \quad r \in \mathrm{IN} \quad s \in \mathrm{IN}}{s[r/\mathtt{x}] \in \mathrm{IN}} \quad \forall\, \Gamma, r, \mathtt{x}, \Delta, s, B. \tag{2}$$

The type A is called **applicative** if IN is closed under:

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash r : A \quad t \in \mathrm{IN} \quad r \in \mathrm{IN}}{t\, r \in \mathrm{IN}} \quad \forall\, \Gamma, t, B, r. \tag{3}$$

Lemma 3.10 *Every substitutive type is applicative.*

Proof: Let A be a substitutive type. We show that IN is closed under (3) by induction on the fact that $t \in \mathrm{IN}$.

If $t = \mathtt{x}\, r_1 \dots r_m$ with $r_1, \dots, r_m \in \mathrm{IN}$, then as $r \in \mathrm{IN}$ also, we have $t\, r = \mathtt{x}\, r_1 \dots r_m\, r \in \mathrm{IN}$.

If $t = \lambda\mathtt{x}{:}A.\, s$ with $s \in \mathrm{IN}$, then as A is substitutive, we have $s[r/\mathtt{x}] \in \mathrm{IN}$ and hence $t\, r = (\lambda\mathtt{x}{:}A.\, s)\, r \in \mathrm{IN}$.

If $t = (\lambda\mathtt{z}{:}C.\, s)\, r_0 \dots r_m$ where $s[r_0/\mathtt{z}]\, r_1 \dots r_m \in \mathrm{IN}$ and $r_0 \in \mathrm{IN}$, then by the induction hypothesis, $s[r_0/\mathtt{z}]\, r_1 \dots r_m\, r \in \mathrm{IN}$. As also $r_0 \in \mathrm{IN}$, we get $t\, r = (\lambda\mathtt{z}{:}C.\, s)\, r_0 \dots r_m\, r \in \mathrm{IN}$. $\square$

Lemma 3.11 *Suppose that $A = A_1 \rightarrow \cdots \rightarrow A_M \rightarrow o$, where each A_i is applicative. Then A is substitutive.*

Proof: We show that IN is closed under (2) by induction on the fact that $s \in \mathrm{IN}$.

If $s = \mathtt{x}\, s_1 \dots s_m$ where $s_1, \dots, s_m \in \mathrm{IN}$, then by the induction hypothesis, $s_i[r/\mathtt{x}] \in \mathrm{IN}$ $(1 \leqslant i \leqslant m)$. The term $s_i[r/\mathtt{x}]$ has type A_i, which is applicative, so that $s[r/\mathtt{x}] = r\, s_1[r/\mathtt{x}] \dots s_m[r/\mathtt{x}] \in \mathrm{IN}$.

If $s = \mathtt{y}\, s_1 \ldots s_m$ where $s_1, \ldots, s_m \in \mathrm{IN}$ and $\mathtt{y} \neq \mathtt{x}$, then by the induction hypothesis, $s_i[r/\mathtt{x}] \in \mathrm{IN}$ $(1 \leqslant i \leqslant m)$. Then $s[r/\mathtt{x}] = \mathtt{y}\, s_1[r/\mathtt{x}] \ldots s_m[r/\mathtt{x}] \in \mathrm{IN}$.

If $s = \lambda\mathtt{y}{:}B.\, s'$ where $s' \in \mathrm{IN}$, then by the induction hypothesis, $s'[r/\mathtt{x}] \in \mathrm{IN}$ so that $s[r/\mathtt{x}] = \lambda\mathtt{y}{:}B.\, s'[r/\mathtt{x}] \in \mathrm{IN}$.

If $s = (\lambda\mathtt{y}{:}B.\, s')\, r_0 \ldots r_m$ where $s'[r_0/\mathtt{y}]\, r_1 \ldots r_m \in \mathrm{IN}$ and $r_0 \in \mathrm{IN}$, then by the induction hypothesis,

$$s'[\tfrac{r}{\mathtt{x}}]\big[r_0[\tfrac{r}{\mathtt{x}}]/\mathtt{y}\big]\,\big(r_1[\tfrac{r}{\mathtt{x}}]\big) \ldots \big(r_m[\tfrac{r}{\mathtt{x}}]\big) = \big(s'[r_0/\mathtt{y}]\, r_1 \ldots r_m\big)[r/\mathtt{x}] \in \mathrm{IN},$$

and $r_0[r/\mathtt{x}] \in \mathrm{IN}$, so that $s[r/\mathtt{x}] = \big(\lambda\mathtt{y}{:}B.\, s'[\tfrac{r}{\mathtt{x}}]\big)\big(r_0[\tfrac{r}{\mathtt{x}}]\big) \ldots \big(r_m[\tfrac{r}{\mathtt{x}}]\big) \in \mathrm{IN}$. □

Theorem 3.12 *Every term of* $\lambda^{\rightarrow}$ *is SN.*

PROOF: An induction on types using lemmas 3.10 and 3.11 shows that every type is applicative. Together with the definition of IN, this shows that IN is closed under all the typing rules of $\lambda^{\rightarrow}$, so that induction on the derivation of $\Gamma \vdash s : B$ gives $s \in \mathrm{IN}$. □

4 Quantitative Normalisation

In this section we give a *quantitative* analysis of normalisation in $\lambda^{\rightarrow}$. We shall give an upper bound for these questions: given a term t, how many reduction steps need to be carried out to reach normal form? how big is the normal form? The results we give were originally proven by Statman.

Later, we will also give lower bounds, showing that any algorithm that gives any information about the normal form of a term t, has a run-time comparable to that needed to reduce terms to normal form.

4.1 Complete Developments

Our analysis starts with the following notion of the **complete development** $\mathcal{D}(t)$ of a term t. Intuitively, $\mathcal{D}(t)$ is given by reducing every redex in t, but not any of the new redexes created by reducing the original redexes. Formally,

$$\mathcal{D}(\mathtt{x}) = \mathtt{x} \qquad \mathcal{D}(\lambda\mathtt{x}{:}A.\, s) = \lambda\mathtt{x}{:}A.\, \mathcal{D}(s) \qquad \mathcal{D}((\lambda\mathtt{x}{:}A.\, s)\, r) = \mathcal{D}(s)[\mathcal{D}(r)/\mathtt{x}]$$

and

$$\mathcal{D}(\mathtt{x}\, r) = \mathtt{x}\, \mathcal{D}(r) \qquad \mathcal{D}(t\, s\, r) = \mathcal{D}(t\, s)\, \mathcal{D}(r).$$

The last two clauses are equivalent to $\mathcal{D}(t\, r) = \mathcal{D}(t)\, \mathcal{D}(r)$ for t not a λ-abstraction.

Exercise 4.1 *Show that if $r \longrightarrow_{\beta 1} s$, then $s \longrightarrow_{\beta} \mathcal{D}(r) \longrightarrow_{\beta} \mathcal{D}(s)$. Show that if $r \longrightarrow_{\beta n} s$, then $s \longrightarrow_{\beta} \mathcal{D}^n(s)$, and (a) give another proof of the Church-Rosser property, and (b) show that if r is weakly normalising, then $\mathcal{D}^N(r)$ is normal for some r.*

4.2 Redex Depths

Given $\Gamma \vdash s : B$, and a sub-term occurrence t in the term s, there is a corresponding sequent in the form $\Gamma\,\Delta \vdash t : C$ in the (unique) derivation of $\Gamma \vdash s : B$. We call C the **type** of t (or $t : C$).

Exercise 4.2 *Formalise the notion of the sequent corresponding to an occurrence above. Show that for an α-class s, and an occurrence in $s_0 \in s$, that the type of the occurrence does not depend on s_0.*

Given a redex $(\lambda \mathtt{x}{:}A.\, s)\, r$ in t, its **redex type** is the type of $\lambda \mathtt{x}{:}A.\, s$.[4]

If $t \longrightarrow_{\beta 1} t'$ by reducing a redex $(\lambda \mathtt{x}{:}A.\, s)\, r$ with redex type $A \to B$, then the redexes in t' can be classified as follows:

1. Those redexes in t' which correspond to redexes in t.
2. If s is in the form $\lambda \mathtt{x}'{:}A'.\, s'$, and the redex occurs in a position in the form $(\lambda \mathtt{x}{:}A.\, s)\, r\, r'$ in t, then $s[r/\mathtt{x}]\, r' = (\lambda \mathtt{x}'{:}A'.\, s'[r/\mathtt{x}])\, r'$ is a redex in t', with redex type B.
3. If r is in the form $\lambda \mathtt{z}{:}C.\, r'$, and there is an occurrence $\mathtt{x}\, s'$ in s, then there is the redex $r\, s'[r/\mathtt{x}] = (\lambda \mathtt{z}{:}C.\, r')(s'[r/\mathtt{x}])$ in t', with redex type A.

The redex types in t' are thus among: 1. the redex types of redexes in t, 2. the type A, and 3. the type B. Considering instead $\mathcal{D}(t)$, we don't get any redexes of class 1., and so:

Lemma 4.3 *Let $\Gamma \vdash t : C$. The redex type of a redex in $\mathcal{D}(t)$ is either A or B, where $A \to B$ is a redex type in t.*

PROOF: A redex in $s[r/\mathtt{x}]$ has redex type which is either a redex type in s or r, or type A, where $r : A$.

We use induction over the clauses defining $\mathcal{D}$. Only two of the clauses can create new redexes. For, $\mathcal{D}((\lambda \mathtt{x}{:}A.\, s)\, r) = \mathcal{D}(s)[\mathcal{D}(r)/\mathtt{x}]$, the redexes of $\mathcal{D}((\lambda \mathtt{x}{:}A.\, s)\, r)$ are either from $\mathcal{D}(s)$, from $\mathcal{D}(r)$, or are created by the substitution and have redex type A. For $\mathcal{D}(t\, s\, r) = \mathcal{D}(t\, s)\, \mathcal{D}(r)$, each redex

[4] This is confusing, because the redex is a sub-term, and the type of the sub-term is not the redex type.

of $\mathcal{D}(t\,s\,r)$ is either from $\mathcal{D}(t\,s)$, $\mathcal{D}(r)$, or is $\mathcal{D}(t\,s)\,\mathcal{D}(r)$. In the latter case, $\mathcal{D}(t\,s)$ must be λ-abstraction of some type B, so that $t\,s$ must be a redex with redex type $A \to B$. □

The **depth** $d(A)$ of a type A is defined by $d(o) = 0$ and $d(A \to B) = 1 + \max(d(A), d(B))$. The **redex depth** of a term t is the maximum of the depths of the redex types in t. The lemma above gives us another proof that every term is weakly normalising:

Corollary 4.4 *Let $\Gamma \vdash s : B$, and s have redex depth N. Then $\mathcal{D}^N(s)$ is normal.*

4.3 Calculating Upper Bounds

Recall that the **length** $l(s)$ of a term is defined by $l(\mathtt{x}) = 1$, $l(\lambda\mathtt{x}{:}A.\,s) = 1 + l(s)$ and $l(t\,r) = 1 + l(t) + l(r)$. We also define the **depth** $d(s)$ of a term s by $d(\mathtt{x}) = 1$, $d(\lambda\mathtt{x}{:}A.\,s) = 1 + d(s)$ and $d(t\,r) = 1 + \max(d(t), d(r))$.

Lemma 4.5

1. *$l(s) < 2^{d(s)}$. $d(s[r/\mathtt{x}]) < d(s) + d(r)$.*
2. *$d(\mathcal{D}(s)) \leqslant l(s)$. $l(\mathcal{D}(s)) \leqslant 2^{l(s)}$.*
3. *If s has m redexes, then $s \longrightarrow_\beta \mathcal{D}(s)$ by a sequence of m one-step reductions.*
4. *s has at most $\frac{l(s)}{2}$ redexes.*

□

Exercise 4.6 *Prove the above.*

We define the **repeated exponential** 2^m_N by $2^m_0 = m$ and $2^m_{N+1} = 2^{2^m_N}$.

Theorem 4.7 *For $\Gamma \vdash s : B$, let N be the redex depth of s. Then the normal form of s has length $\leqslant 2^{l(s)}_N$, and s (if not normal) has a reduction sequence to normal form with length $\leqslant 2^{l(s)}_{N-1}$.*

Proof: By corollary 4.4, $\mathcal{D}^N(s)$ is the normal form of s. That $l(\mathcal{D}^N(s)) \leqslant 2^{l(s)}_N$ follows by induction using 2. of the previous lemma.

We show that $s \longrightarrow_\beta \mathcal{D}^N(s)$ in $\leqslant 2^{l(s)}_{N-1}$ steps by induction on N. For $N = 1$, we have that $s \longrightarrow_\beta \mathcal{D}(s)$ in $\leqslant \frac{1}{2}l(s) \leqslant 2^{l(s)}_0$ steps. By 3. and 4. of the lemma above, we have that $\mathcal{D}^N(s) \longrightarrow_\beta \mathcal{D}^{N+1}(s)$ in $\leqslant \frac{1}{2}l(\mathcal{D}^N(s)) \leqslant \frac{1}{2}2^{l(s)}_N$ steps. Then $s \longrightarrow_\beta \mathcal{D}^N(s) \longrightarrow_\beta \mathcal{D}^{N+1}(s)$ in at most

$$2^{l(s)}_{N-1} + \frac{1}{2}2^{l(s)}_N = \frac{1}{2}(2 \cdot 2^{l(s)}_{N-1} + 2^{l(s)}_N) \leqslant \frac{1}{2}(2^{2^{l(s)}_{N-1}} + 2^{l(s)}_N) = 2^{l(s)}_N$$

steps. $\square$

Exercise 4.8 The **rank** of a type is defined by $\mathrm{rank}(o) = 0$, $\mathrm{rank}(A \to B) = \max(\mathrm{rank}(A) + 1, \mathrm{rank}(B))$. The **redex rank** of a term is the maximum of the ranks of the redex types in the term. Modify the definition of $\mathcal{D}$ so that

$$\mathcal{D}\big((\lambda \mathtt{x}_1 \ldots \lambda \mathtt{x}_m.\, s)\, r_1 \ldots r_m\big) = \mathcal{D}(s)\big[\mathcal{D}(r_1)/\mathtt{x}_1 \ldots \mathcal{D}(r_m)/\mathtt{x}_m\big].$$

Refine theorem 4.7 to have N the redex rank instead of the redex depth.

5 Quantitative Strong Normalisation

In the previous section we gave an upper bound for the lengths of reduction sequences for a particular reduction strategy. In this section we will give an upper bound for $\mathrm{sn}(t)$, the maximum lengths of any reduction sequences starting from t.

We will do this by a method of translations. Give a term t, we will find a new term $|t| : o$, such that $\mathrm{sn}(t)$ is in some sense computed by $|t|$. Combined with the results of the previous section, this will give an upper bound for $\mathrm{sn}(t)$. In order to make the translation work, given a term f of function type $A \to B$, we need not only $|f|$, but also a term mapping $|r|$ to $|f\, r|$. We do this by first defining functionals $(\!|r|\!) : A$ whenever $r : A$, and operators $\aleph_A : A \to o$ mapping $(\!|r|\!)$ to $|r|$. The technique is due to Loader, who used it to prove normalisation results. Gandy earlier gave a somewhat similar translation, in order to derive strong normalisation from weak.

5.1 Translations

We introduce quite a lot of notation. Take two arbitrary variables Σ and C. Let Ξ be the context $\Sigma{:}o \to o \to o\, C{:}o$. We define operators $\Sigma_A(\cdot,\cdot)$ and $\aleph_A(\cdot)$, and terms C_A by recursion on types A:

$$\Sigma_o(i, a) = \Sigma\, i\, a, \quad \aleph_o(i) = i, \quad C_o = C$$

and, taking $\mathtt{x}$ to avoid variable capture,

$$\Sigma_{A\to B}(i, f) = \lambda \mathtt{x}{:}A.\, \Sigma_B(i, f\, \mathtt{x}),$$

$$\aleph_{A\to B}(f) = \aleph_B(f\, C_A).$$

$$C_{A\to B} = \lambda \mathtt{x}{:}A.\, \Sigma_B(\aleph_A \mathtt{x}, C_B),$$

We have the following derived rules:

$$\frac{\Gamma \vdash i : o \quad \Gamma \vdash r : A}{\Xi\Gamma \vdash \Sigma_A(i, r) : A} \qquad \frac{\Gamma \vdash r : A}{\Xi\Gamma \vdash \aleph_A(r) : A} \qquad \frac{}{\Xi\Gamma \vdash C_A : A}.$$

Given a term r, we define $(\!| r |\!)$ by recursion on r:

$$(\!| \mathtt{x} |\!) = \mathtt{x} \qquad (\!| t\, r |\!) = (\!| t |\!)(\!| r |\!) \qquad (\!| \lambda\mathtt{x}{:}A.\, s |\!) = \lambda\mathtt{x}{:}A.\, \Sigma_A(\aleph_A\mathtt{x}, (\!| s |\!)).$$

This gives us the following derived rule:

$$\frac{\Gamma \vdash s : B}{\Xi\Gamma \vdash (\!| s |\!) : B.}$$

We also have that $(\!| \cdot |\!)$ commutes with substitution:

$$(\!| s[r/\mathtt{x}] |\!) = (\!| s |\!)\big[(\!| r |\!)/\mathtt{x}\big].$$

Given $\mathtt{x}_1{:}A_1 \ldots \mathtt{x}_m{:}A_m \vdash s : B$, we define

$$|s| = \aleph_B (\!| s |\!)[C_{A_1}/\mathtt{x}_1 \ldots C_{A_m}/\mathtt{x}_m],$$

so that $\Xi \vdash |s| : o$.

5.2 Reduction of Translations

The relationship between the term $|s|$ and the length of reduction sequences starting from s is derived by examining the behaviour of $|s|$ under reduction.

Lemma 5.1

$$\aleph_A\big(\Sigma_A(i, r)\big) \longrightarrow_\beta \Sigma\, i\, (\aleph_A r) \tag{4}$$

$$\big|\lambda\mathtt{x}{:}A.\, s\big| \longrightarrow_\beta \Sigma\, (\aleph_A C_A)\, |s| \tag{5}$$

If $\Gamma \vdash \mathtt{x} : A_1 \to \cdots \to A_m \to B$ *and* $\Gamma \vdash r_i : A_i$ *(*$1 \leqslant i \leqslant m$*), then*

$$\big|\mathtt{x}\, r_1 \ldots r_m\big| \longrightarrow_\beta \Sigma\, |r_1|\, \big(\ldots \big(\Sigma\, |r_m|\, (\aleph_B C_B)\big) \ldots\big). \tag{6}$$

$$\big|(\lambda\mathtt{x}{:}A.\, s)\, r_0 \ldots r_m\big| \longrightarrow_\beta \Sigma\, \big|r_0\big|\, \big|s[r_0/\mathtt{x}]\, r_1 \ldots r_m\big| \tag{7}$$

Proof: For (4), we use induction on types. The base case holds by definition, and the induction step is

$$\begin{aligned} \aleph_{A\to B}\big(\Sigma_{A\to B}(i, f)\big) \quad &= \quad \aleph_B\big(\Sigma_{A\to B}(i, f)\, C_A\big) \\ &\longrightarrow_\beta \quad \aleph_B\big(\Sigma_B(i, f\, C_A)\big) \\ &\longrightarrow_\beta \quad \Sigma\, i\, (\aleph_B(f\, C_A)) \quad = \quad \Sigma\, i\, (\aleph_{A\to B} f) \end{aligned}$$

We denote the substitution in the definition of $|\cdot|$ by σ, so that $|s| = \aleph_B (\!|s|\!) [\sigma]$. For (5), we have

$$\begin{array}{rcl} |\,\lambda\mathtt{x}{:}A.\, s| & = & \aleph_{A\to B}\big(\lambda\mathtt{x}{:}A.\,\Sigma_B(\aleph_A\mathtt{x}, (\!|s|\!))[\sigma]\big) \\ & \longrightarrow_\beta & \aleph_B\big(\Sigma_B(\aleph_A C_A, (\!|s|\!)[\sigma, C_A/\mathtt{x}])\big) \\ & \longrightarrow_\beta & \Sigma\,(\aleph_A C_A)\,(\aleph_B(\!|s|\!)[\sigma, C_A/\mathtt{x}]) \\ & = & \Sigma\,(\aleph_A C_A)\,|s|. \end{array}$$

For (6),

$$\begin{array}{rcl} \big|\mathtt{x}\, r_1 \ldots r_m\big| & = & \aleph_B\big(C_A\,(\!|r_1|\!)[\sigma]\ldots(\!|r_m|\!)[\sigma]\big) \\ & \longrightarrow_\beta & \aleph_B\big(\Sigma_B\big(\aleph_{A_1}(\!|r_1|\!)[\sigma], \ldots, \Sigma_B\big(\aleph_{A_m}(\!|r_m|\!)[\sigma], C_B\big)\ldots\big)\big) \\ & = & \aleph_B\big(\Sigma_B\big(|r_1|, \ldots, \Sigma_B\big(|r_m|, C_B\big)\ldots\big)\big) \\ & \longrightarrow_\beta & \Sigma\,|r_1|\,\big(\ldots\big(\Sigma\,|r_m|\,(\aleph_B C_B)\big)\ldots\big), \end{array}$$

where the last step uses (4) m times. For (7),

$$\begin{array}{rcl} (\!|(\lambda\mathtt{x}{:}A.\, s)\, r_0 \ldots r_m|\!) & = & \big(\lambda\mathtt{x}{:}A.\,\Sigma_C(\aleph_A\mathtt{x}, (\!|s|\!))\big)\,(\!|r_0|\!)\ldots(\!|r_m|\!) \\ & \longrightarrow_\beta & \Sigma_C\big(\aleph_A(\!|r_0|\!), (\!|s|\!)[(\!|r_0|\!)/\mathtt{x}]\big)\,(\!|r_1|\!)\ldots(\!|r_m|\!) \\ & \longrightarrow_\beta & \Sigma_B\big(\aleph_A(\!|r_0|\!), (\!|s|\!)[(\!|r_0|\!)/\mathtt{x}]\,(\!|r_1|\!)\ldots(\!|r_m|\!)\big) \\ & = & \Sigma_B\big(\aleph_A(\!|r_0|\!), (\!|s[r_0/\mathtt{x}]\, r_1 \ldots r_m|\!)\big). \end{array}$$

where $C = A_1 \to \cdots \to A_m \to B$. Applying the definition $|t| = \aleph_B(\!|t|\!)[\sigma]$ to the above gives:

$$\begin{array}{rcl} \big|(\lambda\mathtt{x}{:}A.\, s)\, r_0 \ldots r_m\big| & \longrightarrow_\beta & \aleph_B\big(\Sigma_B(\aleph_A(\!|r_0|\!), (\!|s[r_0/\mathtt{x}]\, r_1 \ldots r_m|\!))\big)[\sigma] \\ & \longrightarrow_\beta & \Sigma\,\big(\aleph_A(\!|r_0|\!)[\sigma]\big)\,\big(\aleph_B(\!|s[r_0/\mathtt{x}]\, r_1 \ldots r_m|\!)[\sigma]\big) \\ & = & \Sigma\,\big|r_0\big|\,\big|s[r_0/\mathtt{x}]\, r_1 \ldots r_m\big| \end{array}$$

as required. $\square$

For us, the crucial facts in the lemma above are that $\big|\mathtt{x}\, r_1 \ldots r_m\big|$ is expressed in terms of the $|r_i|$, $\big|\lambda\mathtt{x}{:}A.\, s\big|$ in terms of $|s|$, and $\big|(\lambda\mathtt{x}{:}A.\, s)\, r_0 \ldots r_m\big|$ in terms of $|r_0|$ and $|s[r_0/\mathtt{x}]\, r_1 \ldots r_m|$. This will enable us to relate the size of the normal form of $|t|$ to the number of instances of the clauses of (1) needed to derive t, and hence, by exercise 3.3, to $\mathrm{sn}(t)$.

5.3 Translations and Strong Normalisation

Lemma 5.2 *Let μ be a natural number valued function defined on those terms n with $\Xi \vdash n : o$, such that (a) if $n \longrightarrow_\beta n'$, then $\mu(n) = \mu(n')$, and (b) $\mu(\Sigma\, n\, m) \geqslant \mu(n) + \mu(m) + 1$. Then $\mathrm{sn}(t) \leqslant \mu|t|$ for any term t.*

PROOF: By exercise 3.3, it suffices to show that

$$\mu\big|\mathtt{x}\, r_1 \ldots r_m\big| \geqslant \mu\big|r_1\big| + \cdots + \mu\big|r_m\big| + 1, \qquad \mu\big|\lambda\mathtt{x}{:}A.\, s\big| \geqslant \mu\big|s\big| + 1$$

and

$$\mu\big|(\lambda\mathtt{x}{:}A.\, s)\, r_0 \ldots r_m\big| \geqslant \mu\big|s[r_0/\mathtt{x}]\, r_1 \ldots r_m\big| + \big|r_0\big| + 1.$$

These three inequalities follow immediately from the two properties of μ and 2., 3. and 4. of lemma 5.1. □

One such μ is given by letting $\mu(s)$ be the length of the normal form of s. Combined with theorem 4.7, this gives:

Theorem 5.3 *There is a constant K such that, for any term s of $\lambda^{\to}$, we have $\mathrm{sn}(s) \leqslant 2_N^{K\cdot M\cdot l(s)}$, where M (resp. N) is the maximum of the lengths (resp. depths) of the types of the sub-terms of s.*

PROOF: By inspection, there is K such that $l|s| \leqslant K \cdot M \cdot l(s)$. The redexes in $|s|$ are sub-types of the types of sub-terms of s. Letting μ be the length of normal form function, and applying theorem 4.7 and lemma 5.2 gives the result. □

Exercise 5.4 Let $\mathcal{M}$ be the model of $\lambda^{\to}$ with $\mathcal{M}(o)$ the set of natural numbers. *By choosing a suitable valuation on Ξ, reprove strong normalisation using lemma 5.2.* This is a quite general technique for proving strong normalisation.

Exercise 5.5 *Refine theorem 5.3 to give* $\mathrm{sn}(s) \leqslant 2_N^{K\cdot l(s)}$, *where N is redex rank of s.* Hints:

The translation we used is more sophisticated than is required if we just wish to bound $\mathrm{sn}(t)$. Show that taking the C_A to be variables, and $\aleph_A(r) = C_{A\to o}\, r$, in fact gives $\mathrm{sn}(t)$ bounded by the length the normal form of $(\!|t|\!)$.

Additionally, modify $\Sigma_{A\to B}$ so that $\Sigma_{A\to B}(i, \lambda\mathtt{x}{:}A.\, s) = \lambda\mathtt{x}{:}A.\, \Sigma_B(i, s)$ (the original definition β-reduces to this). Also, check that, if the t for which we wish to bound $\mathrm{sn}(t)$ has no sub-term of type A, then we can take $\Sigma_{A\to B}(i, f) = f$.

These modifications ensure that the types of redexes in $(\!|t|\!)$ are exactly the types of redexes in t, so that the N of our bound can be taken to be the redex rank of t (using exercise 4.8).

To show that the exponent $K\cdot M\cdot l(t)$ can be reduced to $K\cdot l(t)$, carefully β-expand (i.e., replace $s[r/\mathtt{x}]$ with $(\lambda\mathtt{x}{:}A.\, s)r$) the term $(\!|t|\!)$. Doing this correctly should give a term with length bounded by $K \cdot l(t)$, and hence give the bound $\mathrm{sn}(t) \leqslant 2_N^{K\cdot l(t)}$.

6 Lower Bounds for Normalisation

The upper bounds given in the previous two sections are not too bad, as the following example shows.

For any number n and type A, let n_A be the term $\lambda \mathtt{f}{:}A{\to}A.\, \lambda \mathtt{x}{:}A.\, \mathtt{f}^n \mathtt{x}$ of type $(A{\to}A) \to A \to A$. It is easy to show that

$$m_{A\to A}\, n_A = (n^m)_A.$$

Let the term $\mathrm{big}_{A,n}$ be given by

$$\mathrm{big}_{A,0} = 0_A \qquad \mathrm{big}_{A,n+1} = \mathrm{big}_{A\to A,n}\, 2_A.$$

By induction on n, we see that

$$\mathrm{big}_{A,n} \longrightarrow_\beta (2^0_n)_A.$$

The term $\mathrm{big}_{o,n}$ has redex rank $n+2$, and length $O(n)$, so that the upper bound on the size of the normal form of $\mathrm{big}_{o,n}$ given in section 4 is $2^{O(n)}_{n+2}$.

A program P that takes a natural number as input is called **elementary recursive** (or Kalmar elementary) if there is K such that $P(n)$ has run time $\leqslant 2^n_K$. A function f is called **elementary recursive** if it is computed by an elementary recursive program.[5]

The example above shows that the function computing the normal form of a typed λ-term is not elementary recursive, while the bounds of the previous sections show that that function is only just outside of elementary recursive.

In the following, we shall prove stronger versions of this, showing that any function giving non-trivial information about normal forms of terms is not elementary recursive.

For example, consider the closed terms of type $o \to o \to o$. There are just two closed normal forms of this type: $\lambda \mathtt{x}{:}o.\, \lambda \mathtt{y}{:}o.\, \mathtt{x}$ and $\lambda \mathtt{x}{:}o.\, \lambda \mathtt{y}{:}o.\, \mathtt{y}$. Computing the normal form of a closed term of type $o{\to}o{\to}o$ by β-reduction is not elementary recursive. However, this leaves the possibility that there is an elementary recursive program computing the same function by a different method. We shall show that this is not the case:

Theorem 6.1 *There is no elementary recursive program computing the normal form of closed terms of type $o \to o \to o$.* $\square$

This extends easily to give the no-non-trivial-information result:

[5] Note that due to the size of the functions 2^n_K, it is pretty much irrelevant what formalism is used to express programs and their run-times—even an exponential difference in efficiency between two formalisms can be accommodated by changing K.

Corollary 6.2 *Suppose that $\Gamma \vdash s_1 : A$, $\Gamma \vdash s_2 : A$. Let P be a program such that if $\Gamma \vdash t : A$ and $t \longrightarrow_\beta s_i$ then $P(t)$ returns i ($i = 1, 2$). Then P is not elementary recursive.*

PROOF (of corollary from theorem): Let P be a such a program. We find a program Q computing the normal of closed terms of type $o \to o \to o$ as follows.

Define the type substitution $B[A/o]$ by $o[A/o] = A$, $(C \to D)[A/o] = C[A/o] \to D[A/o]$. Extend to terms by performing the type substitution in each λ-abstraction. Define $Q(t)$ to return $\lambda \mathtt{x}{:}o.\, \lambda \mathtt{y}{:}o.\, \mathtt{x}$ if $P(t[A/o]\, s_1\, s_2)$ returns 1, and to return $\lambda \mathtt{x}{:}o.\, \lambda \mathtt{y}{:}o.\, \mathtt{y}$ if $P(t[A/o]\, s_1\, s_2)$ returns 2.

Clearly, if $\vdash t : o \to o \to o$, then $Q(t)$ computes the normal form of t, and by theorem 6.1, cannot be elementary recursive. The run-time of Q is only a little more than that of P: if P were elementary recursive then so would be Q. Hence P is not elementary recursive. $\square$

6.1 Higher Order Propositional Classical Logic

Our proof of theorem 6.1 will proceed via encoding **higher order propositional classical logic** (PCL_ω). The language of PCL_ω is as follows. There are countably many variables $\mathcal{V}^N = \{\mathtt{x}_0^N, \mathtt{x}_1^N, \dots\}$ of each rank N. Atomic formulas are $\mathtt{x}_i^0$ and $\mathtt{x}_i^N \in \mathtt{x}_j^{N+1}$, connectives are $\vee$ (binary) and $\neg$ (unary), and we have quantifiers $\exists^N$ of each rank. As a grammar:

$$\mathcal{F} := \mathcal{V}^0 \mid \mathcal{V}^N \in \mathcal{V}^{N+1} \mid \mathcal{F} \vee \mathcal{F} \mid \neg \mathcal{F} \mid \exists^N \mathcal{V}^N.\, \mathcal{F}.$$

The logic is classical, so we define other connectives in the usual way, e.g.,

$$\phi \wedge \psi := \neg(\neg\phi \vee \neg\psi) \qquad \phi \Rightarrow \psi := \neg\phi \vee \psi$$

$$\phi \Leftrightarrow \psi := (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \qquad \phi \oplus \psi := (\phi \wedge \neg\psi) \vee (\neg\phi \wedge \psi)$$

$$\mathtt{x}^N \notin \mathtt{y}^{N+1} := \neg(\mathtt{x}^N \in \mathtt{y}^{N+1}) \qquad \forall^N \mathtt{x}^N.\, \phi := \neg\exists^N \mathtt{x}^N.\, \neg\phi.$$

The semantics of PCL_ω is given as follows. Let $\mathcal{B} = \{\mathtt{tt}, \mathtt{ff}\}$, and let $\mathcal{B}_0 = \mathcal{B}$, $\mathcal{B}_{N+1} = \mathcal{B}^{\mathcal{B}_N}$. Then the rank N variables $\mathtt{x}^N$ are interpreted as ranging over $\mathcal{B}_N$. The atomic formula $\mathtt{x}^N \in \mathtt{y}^{N+1}$ is interpreted as $\mathtt{y}^{N+1}(\mathtt{x}^N)$. The connectives are interpreted in the usual way.

Exercise 6.3 *Formalise the semantics.* Define a **valuation** to be a function σ mapping each variable $\mathtt{x}^N$ into $\mathcal{B}_N$, and then define $[\![\phi]\!]_\sigma \in \mathcal{B}$ for formulae ϕ and valuations σ.

Show that $[\![\phi]\!]_\sigma$ depends only on ϕ and $\sigma\mathtt{x}$ for variables $\mathtt{x}$ free in ϕ. In particular, $[\![\phi]\!]$ is well-defined for closed ϕ.

The set $\mathcal{B}_N$ has 2^2_N elements, so that the obvious algorithm computing $[\![\phi]\!]$ is non-elementary. In the appendix, we prove that:

Theorem 6.4 *The function $[\![\phi]\!]$ for closed ϕ is not elementary.* $\square$

We shall encode $\mathcal{B}_N$ in $\lambda^{\rightarrow}$, in such a way that we can give a reasonably short encoding of iteration over the members of the $\mathcal{B}_N$, and then of the semantics $[\![\phi]\!]$. The function giving the encoding will be elementary recursive, which together with theorem 6.4 will give a proof of theorem 6.1.

6.2 Encoding the Boolean Hierarchy

We encode the elements of the sets $\mathcal{B}_N$ as closed terms of $\lambda^{\rightarrow}$. The encoding is 1-many rather than 1-1. We define the types $\mathcal{B} = o \rightarrow o \rightarrow o$, and $\mathcal{B}_0 = \mathcal{B}$, $\mathcal{B}_{N+1} = \mathcal{B}_N \rightarrow \mathcal{B}$. The intention is that elements of the set $\mathcal{B}_N$ will be represented as closed type $\mathcal{B}_N$ terms. We define the relations $\sim_N$ as follows.

For $a \in \mathcal{B}$ and $\Gamma \vdash r : \mathcal{B}$, put $a \sim r$ iff either $a = \mathtt{tt}$ and $r \longrightarrow_\beta \lambda\mathtt{x}{:}o.\, \lambda\mathtt{y}{:}o.\, \mathtt{x}$ or $a = \mathtt{ff}$ and $r \longrightarrow_\beta \lambda\mathtt{x}{:}o.\, \lambda\mathtt{y}{:}o.\, \mathtt{y}$.

Let $\sim_0$ be $\sim$, and put $b \sim_{N+1} s$ iff $b(a) \sim s\, r$ whenever $a \sim_N r$. We let $\mathtt{tt}, \mathtt{ff} : \mathcal{B}$ be given by $\mathtt{tt} = \lambda\mathtt{x}{:}o.\, \lambda\mathtt{y}{:}o.\, \mathtt{x}$ and $\mathtt{ff} = \lambda\mathtt{x}{:}o.\, \lambda\mathtt{y}{:}o.\, \mathtt{y}$, so that $\mathtt{tt} \sim \mathtt{tt}$ and $\mathtt{ff} \sim \mathtt{ff}$.

Exercise 6.5 *Show that for each $a \in \mathcal{B}_N$ there is r such that $a \sim_N r$.*

If $\vdash R : (\mathcal{B}_N {\rightarrow} A {\rightarrow} A) \rightarrow A \rightarrow A$, then R is called an N-A-**iterator** if it β-reduces to

$$\lambda\mathtt{f}{:}\mathcal{B}_N {\rightarrow} A {\rightarrow} A.\, \lambda\mathtt{x}{:}A.\, \mathtt{f}\, r_0\, (\ldots(\mathtt{f}\, r_{2^2_N-1}\, \mathtt{x})\ldots) \tag{8}$$

where the r_i **enumerate** $\mathcal{B}_N$, in the sense that for each $a \in \mathcal{B}_N$, there is exactly one i such that $a \sim_N r_i$.

We define $r \vee r' = \lambda\mathtt{x}{:}o.\, \lambda\mathtt{y}{:}o.\, r\, \mathtt{x}\, (r'\, \mathtt{x}\, \mathtt{y})$, $\neg r = \lambda\mathtt{x}{:}o.\, \lambda\mathtt{y}{:}o.\, r\, \mathtt{y}\, \mathtt{x}$ for terms $r, r' : o$, so that $r \vee r', \neg r : o$ also. Other Boolean connectives are defined in terms of these.

Given a N-$\mathcal{B}$-iterator R, we define $\exists^N, \forall^N : \mathcal{B}_{N+1} \rightarrow \mathcal{B}$ by

$$\exists^N = \lambda\mathtt{f}{:}\mathcal{B}_{N+1}.\, R\, (\lambda\mathtt{x}{:}\mathcal{B}_N.\, \lambda\mathtt{y}{:}\mathcal{B}.\, \mathtt{y} \vee \mathtt{f}\, \mathtt{x})\, \mathtt{ff}$$

and

$$\forall^N = \lambda\mathtt{f}{:}\mathcal{B}_{N+1}.\, R\, (\lambda\mathtt{x}{:}\mathcal{B}_N.\, \lambda\mathtt{y}{:}\mathcal{B}.\, \mathtt{y} \wedge \mathtt{f}\, \mathtt{x})\, \mathtt{tt}.$$

We will write $\exists^N_R$ and $\forall^N_R$ when it is desired to emphasise the iterator.

Exercise 6.6 Let R be a N-$\mathcal{B}$-iterator. Show that if $a \sim_{N+1} r$ then $\mathtt{tt} \sim \exists_R^N r$ iff $a(b) = \mathtt{tt}$ for some $b \in \mathcal{B}_N$, and $\mathtt{ff} \sim \exists_R^N r$ iff $a(b) = \mathtt{ff}$ for all $b \in B_N$. Show the corresponding property of $\forall_R^N$.

An N-**equality test** is a term $\vdash \mathrm{eq} : \mathcal{B}_N \to \mathcal{B}_N \to \mathcal{B}$ such that if $a \sim r$, $a' \sim r'$, then $\mathtt{tt} \sim \mathrm{eq}\, r\, r'$ if $a = a'$, and $\mathtt{ff} \sim \mathrm{eq}\, r\, r'$ if $a \neq a'$. Given an N-$\mathcal{B}$-iterator R, we define an N+1-equality test

$$\mathrm{eq}_R = \lambda \mathtt{x}, \mathtt{y}{:}\mathcal{B}_{N+1}.\, \forall_R^N\, (\lambda \mathtt{z}{:}\mathcal{B}_N.\, \mathtt{x}\,\mathtt{z} \Leftrightarrow \mathtt{y}\,\mathtt{z}).$$

Exercise 6.7 *Prove that* eq_R *really is an equality test.*

Lemma 6.8 *Given a type A, let $I_{N,A}$ be the type $(\mathcal{B}_N \to A \to A) \to A \to A$ of N-A-iterators. Suppose that R is an N-$I_{N+1,A}$-iterator, and that* eq *is an N-equality test. Then there is an N+1-A-iterator.*

In fact, the existence of N-A-iterators is trivial, since for each $a \in \mathcal{B}_n$ there is r such that $a \sim r$. However, the obvious construction gives iterators of size $O(2_N^2)$ or worse. The construction we give in the proof below will lead to an N-A-iterator that is much smaller.

Observe that a N-I_A-iterator iterates over a $|\mathcal{B}_N|$ element set, while a N+1-A-iterator iterates over a $|\mathcal{B}_{N+1}| = 2^{|\mathcal{B}_N|}$ element set. The idea of the proof below is to define an operator Φ such that if G iterates over a m element set, then $\Phi\, G$ iterates over a $2m$ element set. Then, applying R to Φ, we obtain a term that iterates over a $2^{|\mathcal{B}_N|} = |\mathcal{B}_{N+1}|$ element set.

Proof: Given a set $S \subset \mathcal{B}_{N+1}$, an S-A-**iterator** is a term of type $I_{N+1,A}$ which β-reduces to

$$\lambda \mathtt{f}{:}\mathcal{B}_N{\to}A{\to}A.\, \lambda \mathtt{x}{:}A.\, \mathtt{f}\, s_0\, (\ldots (\mathtt{f}\, s_{m-1}\, \mathtt{x}) \ldots)$$

where $s_0 \ldots s_{m-1}$ enumerates S.

Given a set $T \subset \mathcal{B}_N$, a T-**based** A-**iterator** is a S-A-iterator where

$$S = \big\{ b \in \mathcal{B}_{N+1} \;\big|\; b(x) \neq \mathtt{ff} \text{ implies } x \in T \big\}.$$

A $\mathcal{B}_N$-based A-iterator is just an N+1-A-iterator. We will find a term

$$\Phi : \mathcal{B}_N \to I_{N+1,A} \to I_{N+1,A}$$

such that if $a \sim_N r$, and F is a T-based A-iterator, with $a \notin T$, then $\Phi\, r\, F$ is a $T{\cup}\{a\}$-based A-iterator.

Given $b \in \mathcal{B}_{N+1}$ and $a \in \mathcal{B}_N$, define $b \cup \{a\} \in \mathcal{B}_{N+1}$ by $(b \cup \{a\})(a) = \mathtt{tt}$, and $(b \cup \{a\})(x) = b(x)$ for $x \neq a$. Given terms $s : \mathcal{B}_{N+1}$ and $r : \mathcal{B}_N$, define $s \cup \{r\} = \lambda\mathtt{x}{:}\mathcal{B}_N.(s\,\mathtt{x}) \vee (\mathrm{eq}\,r\,\mathtt{x})$. If $b \sim_{N+1} s$ and $a \sim_N r$, then $b \cup \{a\} \sim_{N+1} s \cup \{r\}$.

If G is an S-A-iterator, for some $S \subset \mathcal{B}_{N+1}$, β-reducing to

$$\lambda\mathtt{f}{:}\mathcal{B}_{N+1}{\to}A{\to}A.\,\lambda\mathtt{x}{:}A.\,\mathtt{f}\,s_0\,\big(\ldots\big(\mathtt{f}\,s_{m-1}\,\mathtt{x}\big)\ldots\big), \tag{9}$$

then

$$\lambda\mathtt{f}{:}\mathcal{B}_{N+1}{\to}A{\to}A.\,\lambda\mathtt{x}{:}A.\,G\,\big(\lambda\mathtt{y}{:}\mathcal{B}_{N+1}.\,\mathtt{f}\,(\mathtt{y} \cup \{r\})\big)\,\mathtt{x} \tag{10}$$

reduces to

$$\lambda\mathtt{f}{:}\mathcal{B}_{N+1}{\to}A{\to}A.\,\lambda\mathtt{x}{:}A.\,\mathtt{f}\,s'_0\,\big(\ldots\big(\mathtt{f}\,s'_{m-1}\,\mathtt{x}\big)\ldots\big), \tag{11}$$

where $s'_i = s_i \cup \{r\}$. If $a \sim r$, then the s'_i enumerate $S' = \big\{b \cup \{a\} \bigm| b \in S\big\}$, so that the term (10) is a S'-A-iterator.

Let Φ be

$$\begin{aligned}&\lambda\mathtt{z}{:}\mathcal{B}_N.\,\lambda\mathtt{G}{:}I_{N+1,A}.\\ &\qquad\lambda\mathtt{f}{:}\mathcal{B}_{N+1}{\to}A{\to}A.\,\lambda\mathtt{x}{:}A.\,\mathtt{G}\,\big(\lambda\mathtt{y}{:}\mathcal{B}_{N+1}.\,\mathtt{f}\,(\mathtt{y} \cup \{\mathtt{z}\})\big)\big(\mathtt{G}\,\mathtt{f}\,\mathtt{x}\big).\end{aligned}$$

Given $a \sim_N r$ and an S-A-iterator G with normal form (9), combining (9) and (11) gives that the term $\Phi\,r\,G$ reduces to

$$\lambda\mathtt{f}{:}\mathcal{B}_{N+1}{\to}A{\to}A.\,\lambda\mathtt{x}{:}A.\,\mathtt{f}\,s'_0\,\big(\ldots\big(\mathtt{f}\,s'_{m-1}\,(\mathtt{f}\,s_0\,(\ldots(\mathtt{f}\,s_{m-1}\,\mathtt{x})\ldots))\big)\ldots\big).$$

This shows that $\Phi\,r\,G$ is an S''-A-iterator where $S'' = S \cup S'$. In particular, if G is a T-based A-iterator, then $\Phi\,r\,G$ is a $T{\cup}\{a\}$-A-iterator, which is the property we desire from Φ.

A $\emptyset$-based A-iterator is given by

$$G_0 = \lambda\mathtt{f}{:}\mathcal{B}_{N+1}{\to}A{\to}A.\,\lambda\mathtt{x}{:}A.\,\mathtt{f}\,(\lambda\mathtt{z}{:}\mathcal{B}_N.\,\mathtt{ff})\,\mathtt{x}.$$

Let R' be the term $R\,\Phi\,G_0$. As R is an N-I_A-iterator, R' reduces to

$$\Phi\,r_0\,\big(\ldots\big(\Phi\,r_{m-1}\,G_0\big)\ldots\big)$$

where (for some a_i) $a_i \sim r_i$ and the a_i enumerate $\mathcal{B}_N$. As $\mathcal{B}_N = \emptyset \cup \{a_{m-1}\} \cup \cdots \cup \{a_0\}$, we have that R' is a $\mathcal{B}_N$-based A-iterator, or in other words, a $N{+}1$-A-iterator. $\Box$

Lemma 6.9 *There is an elementary recursive function that given N and A, gives a N-A-iterator.*

Proof: A 0-A-iterator is given explicitly by

$$\lambda\mathtt{f}{:}\mathcal{B}{\to}A{\to}A.\,\lambda\mathtt{x}{:}A.\,\mathtt{f}\,\mathtt{tt}\,(\mathtt{f}\,\mathtt{ff}\,\mathtt{x})$$

and $\lambda\mathtt{x}{:}\mathcal{B}.\,\lambda\mathtt{y}{:}\mathcal{B}.\,\mathtt{x} \Leftrightarrow \mathtt{y}$ is a 0-equality test. Induction with lemma 6.8 gives N-A-iterators and N-equality tests for all N. We leave it to the reader to check that these constructions are actually elementary recursive. $\Box$

6.3 Encoding the Logic

We encode formulae of PCL_ω as $\lambda^\to$ terms, in such a way that the terms compute the semantics. Specifically, if ϕ is a closed formula, then $[\![\phi]\!] \sim \ulcorner\phi\urcorner$. The encoding is elementary recursive, so this will derive theorem 6.1 from theorem 6.4. For each N, let R_N be a N-$\mathcal{B}$-iterator as given by 6.9. The iterators R_N give us $\exists^N : \mathcal{B}_{N+1} \to \mathcal{B}$.

Given a formula ϕ with free variables among $\mathtt{x}_1^{N_1} \ldots \mathtt{x}_m^{N_m}$, we define $\ulcorner\phi\urcorner$ such that

$$\mathtt{x}_1{:}\mathcal{B}_{N_1} \ldots \mathtt{x}_m{:}\mathcal{B}_{N_m} \vdash \ulcorner\phi\urcorner : \mathcal{B}$$

by induction on ϕ. We let $\ulcorner\mathtt{x}\urcorner = \mathtt{x}$ for any variable $\mathtt{x}$. Let

$$\ulcorner\mathtt{x}_i^0\urcorner = \mathtt{x}_i \qquad \ulcorner\mathtt{x}_i^{N_i} \in \mathtt{x}_j^{N+1}\urcorner = \mathtt{x}_j(\mathtt{x}_i)$$

$$\ulcorner\phi \vee \psi\urcorner = \ulcorner\phi\urcorner \vee \ulcorner\psi\urcorner \qquad \ulcorner\neg\phi\urcorner = \neg\ulcorner\phi\urcorner$$

and

$$\ulcorner\exists^N \mathtt{x}^N .\, \phi\urcorner = \exists^N\,(\lambda\mathtt{x}{:}\mathcal{B}_N.\ulcorner\phi\urcorner).$$

Theorem 6.1 follows by checking that $\ulcorner\phi\urcorner$ is well-behaved:

Exercise 6.10 Suppose that σ is a valuation, such that $\sigma(\mathtt{x}_i^{N_i}) \sim_{N_i} r_i$ $(1 \leqslant i \leqslant m)$. *Show that* $[\![\phi]\!]_\sigma \sim \ulcorner\phi\urcorner[r_1/\mathtt{x}_1 \ldots r_m/\mathtt{x}_m]$.

Check that $\ulcorner\phi\urcorner$ *is an elementary recursive function of* ϕ.

A PCL_ω is not Elementary Recursive

We outline a proof of theorem 6.4 that the valuation $[\![\cdot]\!]$ for the logic PCL_ω is not elementary recursive. We do this by encoding Turing machines with a bound 2_N^2 on their behaviour.

A.1 Basic Encoding in PCL_ω

We first examine the structure of the sets $\mathcal{B}_N$. $\mathcal{B}_N$ has 2_N^2 elements. We define a bijection $\mathrm{seq}_N : \{0 \ldots 2_N^2{-}1\} \longrightarrow \mathcal{B}_N$ in a natural way. Let $\mathrm{seq}(0) = \mathtt{ff}$ and $\mathrm{seq}(1) = \mathtt{tt}$ and $\mathrm{seq}_0 = \mathrm{seq}$. Given seq_N, and a number $\sum_{i<2_N^2} d_i \cdot 2^i$ in binary notation, put

$$\mathrm{seq}_{N+1}\Big(\sum_{i<2_N^2} d_i \cdot 2^i\Big)\big(\mathrm{seq}_N(m)\big) = \mathrm{seq}(d_m).$$

This considers elements of $\mathcal{B}_{N+1}$ as numbers in binary notation, the order of the digits given by the numbering of $\mathcal{B}_N$.

We represent a series of predicates as PCL_ω formulae.

Take two binary numbers $d = \sum_{i<n} d_i \cdot 2^i$ and $e = \sum_{i<n} e_i \cdot 2^i$. We have that $d < e$ iff and only if there is a 'pivot' $i_0 < n$ such that $d_{i_0} < e_{i_0}$ and $d_i = e_i$ for $i_0 < i < n$. This inspires the following definition of the formulae $a <_N a'$: let $a <_0 a'$ be $\neg a \wedge a'$, and let $b <_{N+1} b'$ be

$$\exists^N \mathtt{x}.\big(\neg \mathtt{x} \in b \wedge \mathtt{x} \in b' \wedge \forall^N \mathtt{x}'.\big(\mathtt{x} <_N \mathtt{x}' \Rightarrow (\mathtt{x}' \in b \Leftrightarrow \mathtt{x}' \in b')\big)\big).$$

Defining $S_N(a, a') := \forall^N \mathtt{x}.\big((a <_N \mathtt{x}) \oplus (\mathtt{x} <_N a')\big)$ encodes the **successor** relation $a' = a + 1$.

We turn to encoding combinatorial notions. **Equality** $a =_0 b$, at rank 0, is given by $a \Leftrightarrow b$. **Equality** $a =_{N+1} b$ at rank $N+1$, is given by $\forall^N \mathtt{x}.\, \mathtt{x} \in a \Leftrightarrow \mathtt{x} \in b$.

We encode **finite sets** as follows. For $a_1 \dots a_m$ rank N and b rank $N+1$, the relation $b = \{a_1 \dots a_m\}$ is given by

$$\forall^N \mathtt{x}.\big(\mathtt{x} \in b \Leftrightarrow (\mathtt{x} =_N a_1 \vee \dots \vee \mathtt{x} =_N a_m)\big).$$

The **Kuratowski pairing**, $c = (a, b)$ (where $(a, b) = \{\{a\}, \{a, b\}\}$) is given by

$$\exists^{N+1} \mathtt{x}.\, \exists^{N+1} \mathtt{y}.\big(c = \{\mathtt{x}, \mathtt{y}\} \wedge \mathtt{x} = \{a\} \wedge \mathtt{y} = \{a, b\}\big).$$

where c has rank $N + 2$ and a, b have rank N.

We can give an efficient encoding of $a =_N n$, where n is some given number, and a has rank N. $a =_0 0$ is $\neg a$, $a =_0 1$ is a, and $a =_0 n$ is $a \wedge \neg a$ for $n \geqslant 2$. Given $n = 2^{n_1} + \dots + 2^{n_m}$ with the n_i distinct, $b =_{N+1} n$ is

$$\exists^N \mathtt{x}_1 \dots \exists^N \mathtt{x}_m.\big(b = \{\mathtt{x}_1 \dots \mathtt{x}_m\} \wedge \mathtt{x}_1 = n_1 \wedge \dots \wedge \mathtt{x}_m = n_m\big).$$

Finally, $b = \langle a_0 \dots a_{m-1} \rangle$, encoding **small tuples**, is

$$\begin{aligned} &\exists^{N+2} \mathtt{p}_0 \dots \exists^{N+2} \mathtt{p}_{m-1}.\, \exists^N \mathtt{j}_0 \dots \exists^N \mathtt{j}_{m-1}. \\ &\qquad b = \{\mathtt{p}_0 \dots \mathtt{p}_{m-1}\} \wedge \bigwedge_{i<m} \mathtt{p}_i = (a_i, \mathtt{j}_i) \wedge \bigwedge_{i<m} \mathtt{j}_i = i \end{aligned}$$

for b rank $N + 3$ and the a_i rank N.

Exercise A.1 *Check that all the encodings above do what they should.*

Exercise A.2 *For a, b, c rank $N+1$, encode the predicates $0 \in a$, $2^2_{N+1} - 1 \in a$, and $c = a + b$.*

A.2 Encoding Turing Machines

We apply the constructs above to encode Turing machines. Since we are encoding in the decidable logic PCL_ω, we cannot encode arbitrary runs of Turing machines; instead we must consider a tape of some finite length, fixed for that run.

We will use a slight variant on the usual notion of Turing machine. Instead of the head being at a particular cell on the tape at each step, the head is between two adjacent cells. The action taken depends on the state, and the two cells adjacent to the head; the actions taken are to move either left or right, updating the cell traversed.

Tapes are encoded as two **tape halves**—one to the left of the head and one to the right of the head. In both directions, positions are numbered by some $\mathcal{B}_N$, starting with 0 adjacent to the head, so that the two tape halves l and r are encoded by elements of $\mathcal{B}_{N+1}$. The state s is encoded by another element of $\mathcal{B}_{N+1}$, and the overall configuration of the machine by $\langle l, s, r\rangle$.

We encode various operations. We give four **adjacency** predicates of a pair of tape halves:

$$\mathrm{adj}_{\mathtt{tt},\mathtt{tt}}(l,r) := 0 \in l \wedge 0 \in r \qquad \mathrm{adj}_{\mathtt{tt},\mathtt{ff}}(l,r) := 0 \in l \wedge 0 \notin r$$

$$\mathrm{adj}_{\mathtt{ff},\mathtt{tt}}(l,r) := 0 \notin l \wedge 0 \in r \qquad \mathrm{adj}_{\mathtt{ff},\mathtt{ff}}(l,r) := 0 \notin l \wedge 0 \notin r$$

These check for the four possibilities of the cells next to the tape head. The **shift** predicate $\mathrm{sh}(t,t')$ is given by

$$\big(\forall^N \mathtt{i}.\, \forall^N \mathtt{j}.\, S(\mathtt{i},\mathtt{j}) \Rightarrow (\mathtt{i} \in t \Leftrightarrow \mathtt{j} \in t')\big) \wedge 2^2_N{-}1 \notin t.$$

This shifts a tape half along by one, and checks that at the far end, a $\mathtt{ff}$ comes into, or goes out of, view.

There are four possible actions on tapes: moving either left or right, while updating the cell traversed with either $\mathtt{tt}$ or $\mathtt{ff}$. These are given by the **action** predicates below:

$$\mathrm{act}_{L,\mathtt{tt}}(l,r,l',r') := \mathrm{sh}(l',l) \wedge 0 \in r' \wedge \mathrm{sh}(r,r')$$

$$\mathrm{act}_{R,\mathtt{tt}}(l,r,l',r') := \mathrm{sh}(l,l') \wedge 0 \in l' \wedge \mathrm{sh}(r',r)$$

$$\mathrm{act}_{L,\mathtt{ff}}(l,r,l',r') := \mathrm{sh}(l',l) \wedge 0 \notin r' \wedge \mathrm{sh}(r,r')$$

$$\mathrm{act}_{R,\mathtt{ff}}(l,r,l',r') := \mathrm{sh}(l,l') \wedge 0 \notin l' \wedge \mathrm{sh}(r',r)$$

A rule of the Turing machine is given by a sextuple $(n,\alpha,\beta,d,\gamma,n')$, which is read as follows: if we are in a configuration with state n and with α and

β to the left and right, respectively, of the head, then move in direction $d \in \{L, R\}$, update the cell traversed with γ, and transfer to state n'. The **update predicate** $\mathrm{tr}_{n,\alpha,\beta,d,\gamma,n'}(l, r, s, l', r', s')$ saying that this rule updates the configuration (l, s, r) to give (l', s', r') is

$$s = n \wedge \mathrm{adj}_{\alpha,\beta}(l, r) \wedge \mathrm{act}_{d,\gamma}(l, r, l', r') \wedge s' = n'.$$

Encoding tuples, we also let $\mathrm{tr}_{n,\alpha,\beta,d,\gamma,n'}(c, c')$ be

$$\exists^{N+1}\mathtt{l}\,\mathtt{s}\,\mathtt{r}\,\mathtt{l}'\,\mathtt{s}'\,\mathtt{r}'.\, c = \langle \mathtt{l}, \mathtt{s}, \mathtt{r}\rangle \wedge c' = \langle \mathtt{l}', \mathtt{r}', \mathtt{s}'\rangle \wedge \mathrm{tr}_{n,\alpha,\beta,d,\gamma,n'}(\mathtt{l}, \mathtt{s}, \mathtt{r}, \mathtt{l}', \mathtt{s}', \mathtt{r}').$$

Taking the disjunction over a set of rules $R = \{\rho_1 \ldots \rho_M\}$ gives us the **transition relation** $\mathrm{trans_R(c, c')}$:

$$\mathrm{tr}_{\rho_1}(c, c') \vee \ldots \vee \mathrm{tr}_{\rho_M}(c, c').$$

The **initial configuration** of the machine is defined to have $\mathtt{ff}$ everywhere on the tape, and state 0. Hence $\mathrm{init}(c)$ is

$$\exists^{N+1}\mathtt{x}.\, \mathtt{x} = 0 \wedge c = \langle \mathtt{x}, \mathtt{x}, \mathtt{x}\rangle.$$

A set S of states is **run closed** if it contains the initial configuration, and whenever $c \in S$, there is $c' \in S$ updated from c by some rule. This is given by $\mathrm{closed_R(S)}$:

$$\exists^{N+4}\mathsf{c}.\big(\mathrm{init}(\mathsf{c}) \wedge \mathsf{c} \in S\big) \wedge \forall^{N+4}\mathsf{c}.\big(\mathsf{c} \in S \Rightarrow \exists^{N+4}\mathsf{c}'.\big(\mathsf{c}' \in S \wedge \mathrm{trans}(\mathsf{c}, \mathsf{c}')\big)\big).$$

There can fail to be a run closed set for a Turing machine under the following two conditions: either the machine reaches a state to which no rule applies, or the machine reaches a state with a $\mathtt{tt}$ on the tape $2^2_N + 1$ cells away from the head. In these cases we say the machine **2^2_N-terminates**, and otherwise the machine **loops**.

Lemma A.3 *There is an algorithm that given a Turing machine and N, decides whether or not the machine 2^2_N-terminates.*

PROOF: There are only finitely many configurations with no $\mathtt{tt}$ more than 2^2_N cells from the head. These can be enumerated, and the machine run until either it terminates, or the machine loops. $\square$

Exercise A.4 *Check that $[\![\exists^{N+5}\mathsf{S}.\, \mathrm{closed}_R(\mathsf{S})]\!] = \mathtt{ff}$ if and only if R 2^2_N-terminates.*

This gives a proof of theorem 6.4: if there were an elementary recursive decision procedure for PCL_ω, then the encoding above would contradict the following:

Lemma A.5 *There is no elementary recursive algorithm that given a Turing machine and N, decides whether or not the machine 2^2_N-terminates.*

PROOF SKETCH: Suppose there was such an algorithm.

Let $P(\rho, c, N)$ to be the predicate of a Turing machine ρ terminating from initial configuration c in time 2^2_N. Then $P(\rho, c, N)$ is elementary recursive. By the usual sort of diagonalisation argument, this is impossible. $\square$