A dummy first page

If you want to print 2-up, run of from page 2 (the next page).
This will get the book page number in the correct corner.

An introduction to

λ-calculi and arithmetic

with a decent selection of exercises.

H. Simmons and A. Schalk

Mathematical Foundations Group
The University
Manchester

http://www.cs.man.ac.uk/mfg

This version was produced for part of a course in the

Mathematical Logic MSc
Department of Mathematics
The University of Manchester

October – December, 2005

Contents

What's this all about?

A λ-calculus is a syntactic system of a certain kind. There are many such systems, but in these notes will consider only a few, the most common ones. We will look at the untyped calculus $\boldsymbol{\lambda}\mathsf{0}$, the simply typed calculus $\boldsymbol{\lambda}\mathsf{1}$ (with and without products), and an applied calculus $\boldsymbol{\lambda}\mathsf{G}$ designed specifically for the natural numbers. We will look at the basic facilities of the calculi, and analyse how they can capture numeric gadgets.

Each λ-calculus has various components.

There is always a syntactic category of terms. These are just syntactic expressions, strings of primitive symbols, built up according to certain rules. (They are the analogues of formulas in propositional and predicate calculus.) In the untyped calculus these terms are very easy to describe. In a typed system we first set up the family of raw terms and then extract the well-formed terms using a derivation system which controls the typing discipline. It is this aspect which links λ-calculus with various formal proof systems.

The most important, and characteristic, aspect of any λ-calculus is its reduction mechanism. Essentially there is a technique where one term can be transformed into another term by a process that is very like the evaluation of a function. The mechanism is built around the notion of β-reduction and substitution. It is this aspect that we will look at first, and this is the main reason that we consider the untyped calculus.

The reduction relation of a calculus can be used to capture the notion of an algorithm. This relation can have quite complicated combinatorial properties. We will see how a typing discipline can be used to tame these properties and to classify the complexity properties of the numeric gadgets captured in the system.

These notes from an introduction to the subject and as such miss out many interesting aspects and skirt around certain topics. The emphasis is on showing how various calculi work and how they can be used to describe natural number gadgets. A decent number of examples are included in the notes together with some exercises.

Any comments, typos, or corrections can be sent to Harold Simmons at

hsimmons@cs.man.ac.uk

Part I

Development and Exercises

Chapter 1

The untyped calculus $\boldsymbol{\lambda}0$

In this chapter we consider the most primitive of the λ-calculi, the pure untyped calculus. This occurs in some guise within every λ-calculus. At first sight it seems a rather simple calculus. The term structure is uncomplicated, and the reduction mechanism appears to be straight forward. However, the calculus has hidden demons.

1.1 The basic syntax

In this section we set up the family of terms of this most primitive calculus, and we look at the notion of substitution. In one way these terms are unlike the terms in any other part of mathematics (because of the special use of 'λ'). In another way they capture many of the aspects and problems that terms can have throughout mathematics.

If this is the first time you have seen a λ-calculus then you will almost certainly have a particular problem. In fact, you might have this problem even if you have seen a λ-calculus before. You are bound to ask: What are these syntactic shufflings supposed to mean? As yet they are not supposed to mean anything, they are just syntactic shufflings. They are important because they capture in certain ways several different things. Here we are interested in the way they capture the evaluation of functions, and later we will see how this idea helps us to understand, or at least remember, some of this strange looking syntax. For the time being simply take the stuff at face value.

1.1 DEFINITION. The terms of the pure untyped λ-calculus are generated from an unlimited stock of identifiers using three rules of construction (a base and two steps).

- Each identifier is a term.
- If q, p are terms, then the application (qp) is a term.
- If x is an identifier and r is a term, then the abstraction $(\lambda x\,.\,r)$ is a term.

There are no other terms. ■

It is convenient to let

$x, y, z, \ldots$	range over identifiers
$p, q, r, s, t \ldots$	range over terms

but, of course, we are in control so we can break this informal convention if it suits us. We also use upper case letters

$$A, B, C, \ldots, \mathsf{A}, \mathsf{B}, \mathsf{C}, \ldots$$

for certain particular terms designed to to a specific job. These are often referred to as combinators, a name we will explain shortly.

$\overline{0}$	$(\lambda y\,.\,(\lambda x\,.\,x))$	$\lambda y, x\,.\,x$
$\overline{1}$	$(\lambda y\,.\,(\lambda x\,.\,(yx)))$	$\lambda y, x\,.\,yx$
$\overline{2}$	$(\lambda y\,.\,(\lambda x\,.\,(y(yx))))$	$\lambda y, x\,.\,y(yx)$
$\overline{3}$	$(\lambda y\,.\,(\lambda x\,.\,(y(y(yx)))))$	$\lambda y, x\,.\,y(y(yx))$
S	$(\lambda u\,.\,(\lambda y\,.\,(\lambda x\,.\,(y((uy)x)))))$	$\lambda u, y, x\,.\,y(uyx)$
A	$(\lambda v\,.\,(\lambda u\,.\,(\lambda y\,.\,(\lambda x\,.\,((vy)((uy)x))))))$	$\lambda v, u, y, x\,.\,(vy)(uyx)$
M	$(\lambda v\,.\,(\lambda u\,.\,(\lambda y\,.\,(\lambda x\,.\,(v(uy))x))))$	$\lambda v, u, y, x\,.\,v(uy)x$
E	$(\lambda v\,.\,(\lambda u\,.\,(\lambda y\,.\,(\lambda x\,.\,(((vu)y)x)))))$	$\lambda v, u, y, x\,.\,vuyx$
I	$(\lambda x\,.\,x)$	$\lambda x\,.\,x$
K	$(\lambda y\,.\,(\lambda x\,.\,y))$	$\lambda y, x\,.\,y$
S	$(\lambda z\,.\,(\lambda y\,.\,(\lambda x\,.\,((zx)(yx)))))$	$\lambda z, y, x\,.\,(zx)(yx)$
B	$(\lambda z\,.\,(\lambda y\,.\,(\lambda x\,.\,(z(yx)))))$	$\lambda z, y, x\,.\,z(yx)$
C	$(\lambda z\,.\,(\lambda y\,.\,(\lambda x\,.\,((zx)y))))$	$\lambda z, y, x\,.\,zxy$
D	$(\lambda z\,.\,(\lambda y\,.\,(\lambda x\,.\,(y(zx)))))$	$\lambda z, y, x\,.\,y(zx)$

Table 1.1: Some examples of terms (combinators)

The brackets in a term are part of its official construction. They are there to ensure that it is uniquely parsed. However, to help make some terms more legible we often leave out some brackets. We discuss the informal conventions for this at the end of this section. For now let's look at some examples of terms.

1.2 EXAMPLES. Table 1.1 gives some examples of terms (combinators, in a sense to be explained shortly). Each of these terms does a particular job within the calculus, as will become clear as the story unfolds. Each term has been given a name, as indicated in the left column. The official construction is given in the middle column, and then the informal, more legible version is given in the right column (and follows the conventions for omitted brackets). ■

There are several points to notice about these examples.

The first is that the official construction can be quite unreadable. The official version of A suffers, like Henry Beau-geste, from a surfeit of parenthesis. The shortened version is much easier to read.

The second point is that S seems to occur twice, and the two two occurrences are not the same. This is simply one of those instances where the same letter is used as the common name for two different things.

The third point is that the terms are given in three blocks and there seems to be some uniformity in each block. It should be clear how the first block can be continued. Perhaps you can think of an even better, more condensed, notation for these terms. We will use these terms together with those in the middle block in the next section and the next chapter. The third block is a list of standard combinators which we won't use much here but are important in later developments. (There is a companion calculus which deals entirely with combinators and contains no explicit abstraction mechanisms. Rather the perversely this calculus is called Combinatory Logic.)

The fourth point is that from these examples it seems that each term consists of a prefix – a list of abstractions – followed by a body built up from applications alone. This is not the case in general. The following term B illustrates this, and will be used as a running example. (Of course, this B is not the same as the term B of the table. Different things can have the same common name.)

1.3 EXAMPLE. Consider the term B given in full on the left and in abbreviated form on the right.

$$(\lambda u\,.\,(\lambda v\,.\,(\lambda w\,.\,((u(\lambda z\,.\,(\lambda y\,.\,(\lambda x\,.\,(((zv)y)x)))))w)))) \qquad \lambda u,v,w\,.\,u(\lambda z,y,x\,.\,zvyx)w$$

This has the form

$$\lambda u,v,w\,.\,uVw \quad \text{where} \quad V = (\lambda z,y,x\,.\,zvyx)$$

so the abstractions and applications are intertwined. ■

A term is a string of characters selected from the stock of identifiers, the special symbol 'λ', and various punctuation devices. In particular, the brackets in a term punctuate it to ensure it is uniquely parsed. It is sometimes conceptually convenient to think of a term in the form of its parsing tree.

1.4 DEFINITION. The parsing tree of a term t is a finite rooted tree of terms with t at the root, identifiers at the leaves, and grown according to the two rules.

$$\frac{q \quad p}{(qp)} \qquad \frac{r}{(\lambda x\,.\,r)}\,(x)$$

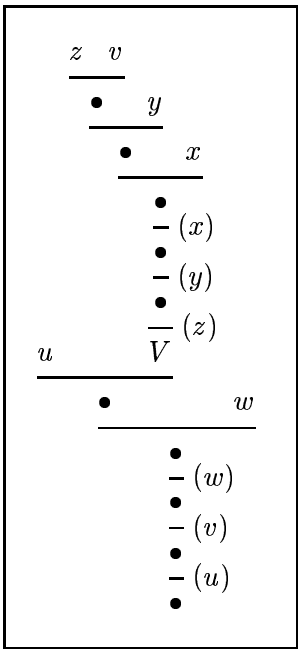
Notice that the abstraction rule is labelled with important identifier. ■

In other words, the parsing tree of a term lays out its construction from the leaves to the root. We don't need to make a meal of this, but it is worth spending just a little time looking at the idea. (A more sophisticated version of the idea is a central tool of each typed λ-calculus.)

Let's use the term

$$\mathsf{B} = \lambda u,v,w\,.\,uVw$$

(of Examples 1.3) to illustrate the idea. Sit down and grow the tree according to the rules given in Definition 1.4. Once you have done that you will realize that much of the information in it can be omitted. All we need to know are the which identifiers occur at the leaves and which identifiers are used in the abstractions. The tree to the right is this Spartan version of the parsing tree. Observe how the original term can be constructed from this tree by simply tracking from leaves to root.



Parsing trees are conceptually neater, and don't need explicit punctuation. However, because of their size and shape parsing trees are not used in the day to day business of the untyped calculus.

The construct $(\lambda - \,.\, -)$ is the cause of many of the syntactic problems that occur with the λ-calculus. It is a binding construct, like quantifiers, bounded summations, and definite integrals. As such it converts a free identifier into a bound one. To bring out this let's make a formal definition.

1.5 DEFINITION. The support ∂t of a term t is generated by

$$\partial x = \{x\} \qquad \partial(qp) = \partial q \cup \partial p \qquad \partial(\lambda x \,.\, r) = \partial r - \{x\}$$

for terms p, q, r and identifiers x. ■

In other words, the support of a term is the finite set of identifiers which occur free at least once in the term. As such the set is often written in a more imaginative way as $\mathrm{fv}(t)$ and called the set of free identifiers or set of identifiers free in the term.

With this notion we can formally defined an idea mentioned several times already.

1.6 DEFINITION. A combinator is a term with empty support. ■

Notice that an identifier can occur both freely and bound in a term. As an example work out the support of

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

by tracking the construction of the term. (This is a rather daft way of writing a term, and is done merely to illustrate the point.) In practice, we normally use different letters to do different jobs, but the formal definition of a term allows multiple uses.

1.7 DEFINITION. Two terms are α-equivalent if they agree except for the choice of bound identifiers. ■

In general we tend not distinguish between α-equivalent terms. Thus we feel free to swap from one term to an α-variant whenever it seems sensible to do so. For example, the two terms

$$(\lambda y \,.\, yx) \qquad (\lambda z \,.\, zx)$$

are α-equivalent (provided x, y, z are pairwise distinct identifiers). We could set up the identification in a formal way, but for what we do here it isn't worth the effort. We will see a similar bit of jiggery-pokery later in this section when we look at substitution.

(This informal identification is sometimes known as the Barendregt convention which is not very nice on Barendregt for it is hardly more than cheating.)

At this stage it is worth explaining the terminology. An identifier (as used here) is a place-holder, and is often called a variable. However, later on this can cause a problem. When we look at the simply typed λ-calculus we will have two kinds of place-holders. We will have identifiers out of which terms are built (as here) and variables out of which things called types are built. These two kinds of place-holders should not be confused, for they do very different jobs. Unfortunately, they always occur together so they need to be called different things. Using two different words, 'identifier' and 'variable' helps this. Of course, informally, in speech, we often call both gadgets variables, because we always have the opportunity to clear up any confusion.

The syntactic construction of terms can seem a bit odd. However, there is a partial explanation of what is going on and, in these notes, it is the main way we use λ-calculi.

This explanation is not part of the formal development, but it does suit our purposes. There are other uses of λ-calculus for which this informal idea doesn't make sense.

Suppose we have a function

$$f : \mathbb{Y} \times \mathbb{X} \longrightarrow \mathbb{T}$$

for sets $\mathbb{Y}, \mathbb{X}, \mathbb{T}$. Thus, given $y \in \mathbb{Y}$ and $x \in \mathbb{X}$ we may tuple these into a pair (y, x) and evaluate f at this input to obtain $f(y, x)$ as the value. In certain circumstances we may have some kind of description of this value

$$f(y, x) = \text{expression}(\cdots y \cdots x \cdots)$$

using some 'expression' built up from y and x and various other gadgets. Think of this 'expression' as a λ-term with its two free identifiers displayed.

We can view f in a different way. We may input any $y \in \mathbb{Y}$ to obtain a function

$$f(y, \cdot) : \mathbb{X} \longrightarrow \mathbb{T}$$

which converts members of $\mathbb{X}$ into members of $\mathbb{T}$. More generally, we can think of that function which when supplied with $y \in \mathbb{Y}$ will return the function $f(y, \cdot)$. This is the curried view of f.

How can we describe the behaviour of the function

$$y \longmapsto f(y, \cdot)$$

the one that converts $y \in \mathbb{Y}$ into a 1-placed function $f(y, \cdot) : \mathbb{X} \longrightarrow \mathbb{T}$? We should be able to do something to the 'expression' to get this description. Whatever we do the result can only have y occurring freely, and not x. We use the notation

$$\lambda x \,.\, \text{expression}(\cdots y \cdots x \cdots)$$

to bind x and leave y free.

This is one of the things that the λ-calculus is trying to get at, and it is the one that will motivate much of what we do here.

An informal explanation of the abstraction $(\lambda x \,.\, r)$ is that it is a typical value of the curried form of the function of which r is a typical value. (Perhaps this is the point to remind you of a piece of pedantry. You often see in older textbooks a phrase such as 'the function $f(x, y)$'. What this really means is 'the function f of which $f(x, y)$ is a typical value'. Usually this distinction doesn't matter, but it does here.)

How should we think of the application (qp)? Think of q as the name of a function and p as the name of an input for that function. If we use these two names to evaluate the function at the input, then we should be able to produce a name for the output. That is what (qp) does; it names the output.

Given a term r, which we think of as a name of a typical value of some function, we can form $(\lambda x \,.\, r)$, which is a name of a typical value of a different, but related, function. Now suppose s is a term which we think of as a name of an input for $(\lambda x \,.\, r)$. Then $(\lambda x \,.\, r)s$ is a name for the output of that function at s. But surely we know a 'better' name for this output. Take the term r and replace each free occurrence of x by s. The term $(\lambda x \,.\, r)s$ and the modified version of r are not the same term, but there is some family relationship.

1.8 DEFINITION. (Not quite) For all terms r, s and each identifier x

$$r[x := s]$$

is the term obtained from r by replacing all free occurrences of x by s, and taking due precautions not to catch anything. ■

You can read this notation as 'r with x replaced by s'. Sometimes the substitution operator $[x := s]$ is written on the left of its operand

$$[s \text{ for } x]r$$

and is read accordingly, 's for x in r'. There are various notations for this operator, some of which are idiotic. The compound $[x/y]$ can mean x replacing y or y replacing x. If you ever see anybody who uses this you should laugh at him in the street.

There are several ways of setting up the substitution process in a formal manner. For what we do here we don't need to get into the details, but shortly we will look at one of the more common ways of handling substitution.

When we perform a substitution, why do we need to take precautions? Consider the term

$$r = (\lambda y \,.\, yx)$$

with $\partial r = \{x\}$. It is perfectly clear that for each term s

$$r[x := s] \text{ is } (\lambda y \,.\, ys)$$

isn't it? It might be clear, but it's wrong. Each identifier which is free in s ought to remain free in $r[x := s]$, so that

$$\partial(r[x := s]) = \partial s$$

for this example. (The general case is a bit more complicated.) But what if y is free in s? For this case in the crude replacement the binding $(\lambda y \,.\, -)$ captures the y in s. As an extreme case when s is just y we have

$$(\lambda y \,.\, yx)[x := y]$$

and the crude replacement gives

$$(\lambda y \,.\, yy)$$

which has empty support, whereas the support should be $\{y\}$.

There is nothing very deep here. When there are free and bound place-holders around (such as in summations or integrals) then a substitution inside the scope of a bound place-holder can be dodgy. The solution is immediate; change the symbol used for the bound place-holder.

We will do the same here. Whenever we perform a substitution we will take the precaution of avoiding identifier capture, by changing any bound identifier which might get above itself. (In fact, this will hardly ever occur here because of the judicious choice of notations in the examples.)

As promised above, let's now look at one of the more common substitution algorithms. The details are not important here, but it is worth seeing some.

1.9 DEFINITION. The term

$$t[x := s]$$

arising by replacing in the term t all free occurrences of the identifier x by the term s is obtained by recursion on t using the following clauses.

$$\begin{array}{ll} x[x := s] = s & (qp)[x := s] = (q[x := s])(p[x := s]) \\ y[x := s] = y & (\lambda y \,.\, r)[x := s] = \lambda z \,.\, (r'[x := s]) \end{array}$$

At the bottom left y is any identifier different from x. At the bottom right z is any 'safe' identifier and r' is $r[y := z]$. In other words all free occurrences of y in r are changed to z so as to avoid identifier capture when the substitution $[x := s]$ is performed. ■

The main problem with any substitution algorithm is organizing the way the 'safe' renaming of bound identifiers is performed. Given this the algorithm above is the first one that come to mind. In fact, it is not a very good algorithm (mainly because it makes nested calls on itself), but that won't bother us here.

To conclude this section let's look at the informal conventions for omitting brackets.

We write

$$tsr \quad \text{for} \quad ((ts)r)$$

for arbitrary terms r, s, t. Thus we reinstate the omitted brackets from the left. This choice of convention is not arbitrary. It is used because it matches well with currying.

Iterated abstraction can contain a lot of brackets.

$$(\lambda x_1 \,.\, (\lambda x_2 \,.\, (\cdots (\lambda x_n \,.\, s) \cdots)))$$

We omit some of these and write

$$\lambda x_1 \,.\, \lambda x_2 \,.\, \cdots \lambda x_n \,.\, s \quad \text{or even} \quad \lambda x_1, x_2 \,\ldots, x_n \,.\, s$$

for this term. Examples 1.2 illustrates this convention in use.

Exercises

1.1 (a) Write out the parsing tree of each of the terms in Table 1.1. Use these trees to work out the support of each subterm of these terms.

(b) Do the same for the daft term

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

which occurs between Definitions 1.6 and 1.7. Can you find a more sensible α-equivalent variant of this term?

1.2 Consider the substitution algorithm of Definition 1.9. Using this give a full and correct proof that

$$t[y := s][x := r] = t[x := r][y := s[x := r]]$$

for arbitrary terms r, s, t and identifiers x, y. If necessary you may impose some restrictions on x and y.

1.2 The reduction mechanism

Every λ-calculi has a reduction mechanism, and it is this aspect this distinguishes λ-calculi from all other syntactic calculi. The reduction mechanism of the untyped calculus occurs, in some guise, in all λ-calculi, so let's get to grips with it.

It is worth repeating, word for word, an explanation used in the previous section.

Given a term r, which we think of as a name of a typical value of some function, we can form $(\lambda x\,.\,r)$, which is a name of a typical value of a different, but related, function. Now suppose s is a term which we think of as a name of an input for $(\lambda x\,.\,r)$. Then $(\lambda x\,.\,r)s$ is a name for the output of that function at s. But surely we know a 'better' name for this output. Take the term r and replace each free occurrence of x by s to obtain $r[x := s]$. The terms $(\lambda x\,.\,r)s$ and $r[x := s]$ are not the same term, but there is some family relationship. The idea is that $r[x := s]$ is nearer to the 'canonical name' of the output than $(\lambda x\,.\,r)$. We try to make this precise.

1.10 DEFINITION. A redex or reducible expression is a compound term

$$(\lambda x\,.\,r)s$$

formed by an outer application and an inner abstraction.

The immediate reduct of the redex

$$r[x := s]$$

is obtained by performing the indicated substitution.

A 1-step reduction

$$(\lambda x\,.\,r)s \rhd r[x := s]$$

converts a redex into its reduct. ■

This is just a formalization of the informal idea above. The process of reduction is just the repeated use of 1-step reductions inside a term.

1.11 DEFINITION. For terms t^- and t^+ we write

$$t^- \rhd\!\!\rhd t^+$$

and say t^- reduces to t^+ if we can move from t^- to t^+ in a finite number of steps where at each step we replace a redex subterm of the current term by its immediate reduct (or take an α-variant).

A term is normal if no subterm is a redex.

Let $\rhd\!\!\rhd\!\!\rhd$ be the reflexive closure of $\rhd\!\!\rhd$. Thus

$$t^- \rhd\!\!\rhd\!\!\rhd t^+$$

holds (for terms t^-, t^+) precisely when either t^- and t^+ are the same term (up to α-equivalence) or $t^- \rhd\!\!\rhd t^+$. ■

Let's expand on this notion. To show $t^- \rhd\!\!\rhd t^+$ we must produce a finite sequence of terms starting with t^- and finishing with t^+. Suppose we have got to one term t in this sequence. To get to the next term we locate a subterm $(\lambda x\,.\,r)s$ of t which happens to

be a redex. We then replace that subterm by its immediate reduct $r[x := s]$ to produce the next term in the sequence.

The reflexive version $\rhd\!\!\rhd\!\!\rhd$ of the reduction relation is mentioned here merely in passing. It will not be needed in this chapter, but will be necessary in a later chapter.

At this point there are three things that should be done. We should formalize the description of $\rhd\!\!\rhd$ is a more precise way. We should give some examples of $\rhd\!\!\rhd$ in use. We should ask, and perhaps answer, some questions about $\rhd\!\!\rhd$ and its behaviour. We can't do these all at once, so we do them one at a time in the order listed. You may prefer to do them in a different order, in which case you can flit about for the remainder of this section without coming to any serious harm.

Here is a more generic way of getting at the reduction relation $\rhd\!\!\rhd$.

1.12 DEFINITION. The reduction relation $\rhd\!\!\rhd$ is the least relation on terms generated by the following rules.

$$\text{Leaf} \qquad \frac{(\lambda x\,.\,r)s \rhd r[x := s]}{(\lambda x\,.\,r)s \rhd\!\!\rhd r[x := s]}$$

$$\text{Application} \qquad \frac{q^- \rhd\!\!\rhd q^+}{q^- p \rhd\!\!\rhd q^+ p} \qquad \frac{p^- \rhd\!\!\rhd p^+}{q p^- \rhd\!\!\rhd q p^+}$$

$$\text{Abstraction} \qquad \frac{r^- \rhd\!\!\rhd r^+}{(\lambda x\,.\,r^-) \rhd\!\!\rhd (\lambda x\,.\,r^+)}$$

$$\text{Composition} \qquad \frac{t^- \rhd\!\!\rhd t^0 \quad t^0 \rhd\!\!\rhd t^+}{t^- \rhd\!\!\rhd t^+}$$

That is, each instance

$$t^- \rhd\!\!\rhd t^+$$

can be witnessed by a finite rooted tree of instances grown according to the rules above and with the particular instance at the root. ■

Rarely, and in these notes never, do we need to use this generic way of describing $\rhd\!\!\rhd$. Usually we show that an instance $t^- \rhd\!\!\rhd t^+$ holds by displaying a chain

$$t^- = t_0 \rhd\!\!\rhd t_1 \rhd\!\!\rhd \cdots \rhd\!\!\rhd t_{l-1} \rhd\!\!\rhd t_l \rhd\!\!\rhd t^+$$

of redex removals, that is instances of 1-step reductions embedded in a term.

1.13 EXAMPLE. Consider the compound term

$$\mathsf{B}\,\overline{3}\,\overline{2}\,\overline{1}$$

using term

$$\mathsf{B} = \lambda u, v, w\,.\,uVw \quad \text{where} \quad V = \lambda z, y, x\,.\,zvyx$$

of Examples 1.3 and the terms $\overline{3}, \overline{2}, \overline{1}$ of Table 1.1. For this example let us write

$$t^- \mathrel{\dot\rhd} t^+$$

to indicate that the term t^+ is obtained from the term t^- by replacing just one subterm which is a redex by the immediate reduct. Thus a reduction $t^- \rhd\!\!\rhd t^+$ can be witnessed by a sequence of uses of $\dot{\rhd}$, one after the other.

The compound $\mathsf{B}\overline{3}$ is a redex, and hence

$$\mathsf{B}\,\overline{3}\,\overline{2}\,\overline{1} \;\dot{\rhd}\; (\lambda v, w\,.\,\overline{3}Vw)\,\overline{2}\,\overline{1}$$

holds. Within this term there are at least two redexes

$$(\lambda v, w\,.\,\overline{3}Vw)\,\overline{2} \qquad \overline{3}V$$

and we can attack these in either order. Let's have a go at the left hand one. With

$$\mathsf{2} = \lambda z, y, x\,.\,z\overline{2}yx$$

we get

$$\mathsf{B}\,\overline{3}\,\overline{2}\,\overline{1} \;\dot{\rhd}\; (\lambda v, w\,.\,\overline{3}Vw)\,\overline{2}\,\overline{1} \;\dot{\rhd}\; \left(\lambda w\,.\,\overline{3}\,\mathsf{2}w\right)\,\overline{1} \;\dot{\rhd}\; \overline{3}\,\mathsf{2}\,\overline{1}$$

where the third step removes a redex created by the second step.

We now remember the shape of $\overline{3}$ to get

$$\mathsf{B}\,\overline{3}\,\overline{2}\,\overline{1} \rhd\!\!\rhd \overline{3}\,\mathsf{2}\,\overline{1} \;\dot{\rhd}\; \left(\lambda x\,.\,\mathsf{2}\,(\mathsf{2}\,(\mathsf{2}x))\right)\,\overline{1} \;\dot{\rhd}\; \mathsf{2}\,(\mathsf{2}\,(\mathsf{2}\overline{1}))$$

which, perhaps, give you a clue as to what is going on.

Within this right-hand term there are at least three redexes. We choose to attack the inner-most one. Thus, remembering the shape of $\mathsf{2}$ and $\overline{1}$ and then $\overline{2}$, we have

$$\mathsf{2}\overline{1} \;\dot{\rhd}\; \lambda y, x\,.\,\overline{1}\,\overline{2}yx \;\dot{\rhd}\; \lambda y, x\,.\,(\lambda x\,.\,\overline{2}x)yx \;\dot{\rhd}\; \lambda y, x\,.\,\overline{2}yx$$

and then

$$\mathsf{2}\overline{1} \rhd\!\!\rhd \lambda y, x\,.\,\overline{2}yx \;\dot{\rhd}\; \lambda y, x\,.\,(\lambda x\,.\,y(yx))x \;\dot{\rhd}\; \lambda y, x\,.\,y(yx) = \overline{2}$$

to give

$$\mathsf{B}\,\overline{3}\,\overline{2}\,\overline{1} \rhd\!\!\rhd \mathsf{2}\,(\mathsf{2}\,(\mathsf{2}\overline{1})) \rhd\!\!\rhd \mathsf{2}\,(\mathsf{2}\overline{2})$$

which seems to be getting closer to a result.

You should go through this last collection of calculations again to make sure you understand exactly how the various substitutions have worked.

There are still several redexes left to get rid of. We have

$$\mathsf{2}\overline{2} \;\dot{\rhd}\; \lambda y, x\,.\,\overline{2}\,\overline{2}yx \;\dot{\rhd}\; \lambda y, x\,.\,(\lambda x\,.\,\overline{2}(\overline{2}x))yx \;\dot{\rhd}\; \lambda y, x\,.\,\overline{2}(\overline{2}y)x$$

and then

$$\mathsf{2}\overline{2} \rhd\!\!\rhd \lambda y, x\,.\,\overline{2}(\overline{2}y)x \rhd\!\!\rhd \lambda y, x\,.\,(\overline{2}y)(\overline{2}yx) \rhd\!\!\rhd \lambda y, x\,.\,(\overline{2}y)(y(yx)) \rhd\!\!\rhd \overline{4}$$

where you should work out what $\overline{4}$ is.

Finally, all this leads to

$$\mathsf{B}\,\overline{3}\,\overline{2}\,\overline{1} \rhd\!\!\rhd \mathsf{2}\,(\mathsf{2}\,(\mathsf{2}\overline{1})) \rhd\!\!\rhd \mathsf{2}\,(\mathsf{2}\overline{2}) \rhd\!\!\rhd \mathsf{2}\overline{4} \rhd\!\!\rhd \overline{2^4}$$

which has given the game away. ■

In the next chapter we return to this kind of calculation a more general setting.

At this stage there may be several questions that come to mind about the behaviour and workings of the calculus. We postpone these until the last section of this chapter.

Exercises

1.3 Using the daft term

$$t = (\lambda x\,.\,((\lambda x\,.\,((\lambda x\,.\,x)x))x))$$

consider the application tx. Starting from tx generate as many reduction paths as possible. Each path should finish with a normal term.

Does it make any difference if you start from a sensible version of t?

1.4 Consider the term

$$\mathsf{B}\overline{2}\,\overline{2}\,\overline{2}$$

which is like (but not the same as) the term used in Example 1.13. Starting with the term generate as many reduction paths as possible, and show that each terminates in the same term (up to α-equivalence).

How many terminating reduction paths are there?

Do they all have the same length?

1.3 Fixed point and other combinators

From Definition 1.7 a combinator is a term with empty support, that is with no freely occurring identifiers. Table 1.1 gives a list of combinators in three block. Those in the top two blocks will will be used in Chapter 2. Those in the bottom block are important for certain aspects of the λ-calculi, but are only incidental for the material developed in these notes.

In this section we look at some special combinators which have weird and wonderful properties.

1.14 DEFINITION. Let

$$\mathsf{Y} = \Omega\Omega \quad \text{where} \quad \Omega = \lambda x, y\,.\,y(xxy)$$

to obtain the Turing fixed point combinator. ■

This name needs an explanation. Unlike many gadgets in Mathematics, which are incorrectly of deliberately miss-attributed, this term *was* devised by Turing. It is a fixed point combinator in the sense to be described shortly. There are many such fixed point combinators, but this one is the best.

1.15 LEMMA. *For each term f we have* $\mathsf{Y}f \rhd\!\rhd f(\mathsf{Y}f)$.

Proof. In fact, we have

$$\mathsf{Y} = \Omega\Omega \rhd\!\rhd \lambda y\,.\,y(\Omega\Omega y) \rhd\!\rhd \lambda y\,.\,y(\mathsf{Y}y)$$

which gives the required result. ■

In this result think of the term f as the description of a function. The result show that $\mathsf{Y}f$ is a description of an input to f such that the description $f(\mathsf{Y}f)$ of the output has the same value as $\mathsf{Y}f$. In other words $\mathsf{Y}f$ is a description of a fixed point of the function described by f.

This is a little strange for, as we will see, many functions can be described in the calculus which simply do not have fixed points. The simplest example is the successor function on the natural numbers. Something dodgy is going on here.

For each identifier y we have

$$\mathsf{Y}y \rhd\!\!\rhd y(\mathsf{Y}y) \rhd\!\!\rhd y^2(\mathsf{Y}y) \rhd\!\!\rhd y^3(\mathsf{Y}y) \rhd\!\!\rhd \cdots$$

so that

$$\mathsf{Y} \rhd\!\!\rhd \lambda y\,.\,\mathsf{Y}y \rhd\!\!\rhd \lambda y\,.\,y(\mathsf{Y}y) \rhd\!\!\rhd \lambda y\,.\,y^2(\mathsf{Y}y) \rhd\!\!\rhd \lambda y\,.\,y^3(\mathsf{Y}y) \rhd\!\!\rhd \cdots$$

and this reduction path can continue for ever. In fact no reduction path from Y ever terminates.

We will make use of this combinator in Chapter 2.

Reduction in this calculus can be quite weird.

1.16 EXAMPLE. Let

$$\omega = \delta\delta \quad \text{where} \quad \delta = \lambda x\,.\,xx$$

to obtain a perverse combinator. For each term t we have

$$\delta t \rhd\!\!\rhd tt$$

and hence

$$\omega = \delta\delta \rhd\!\!\rhd \delta\delta = \omega$$

to obtain show that ω can reduce for ever

$$\omega \rhd\!\!\rhd \omega \rhd\!\!\rhd \omega \rhd\!\!\rhd \cdots$$

and never get anywhere.

In fact, this term is even stranger. The term ω is not normal, since it is a reduct. The two components δ are normal, so that $\omega \rhd\!\!\rhd \omega$ is the only possible reduction from ω. Thus, there is precisely one reduction path from ω, this path never terminates, and merely repeats the term over and over again! ■

Even stranger things than this can happen. Exercise 1.5 shows that there can be different reduction paths from a term one of which leads to a normal term and one of which continues for ever. This shows the need for reduction strategies if normalization is desired. This is a topic not dealt with in these notes.

The term δ of Example 1.16 can be hit with the fixed point combinator Y to get

$$\mathsf{Y}\delta \rhd\!\!\rhd \delta(\mathsf{Y}\delta) \rhd\!\!\rhd (\mathsf{Y}\delta)(\mathsf{Y}\delta)$$

and so produce a term which duplicates itself. The same effect can be achieved more directly.

1.17 EXAMPLE. Consider the term

$$\omega = (\lambda x\,.\,(xx)(xx))$$

with empty support. We have

$$\omega s \rhd (ss)(ss)$$

for each term s. In particular, with

$$\Delta = \omega\omega$$

we have

$$\Delta \rhd \Delta\Delta$$

and hence

$$\Delta \rhd\!\!\rhd \Delta\Delta \rhd\!\!\rhd (\Delta\Delta)(\Delta\Delta) \rhd\!\!\rhd ((\Delta\Delta)(\Delta\Delta))((\Delta\Delta)(\Delta\Delta)) \rhd\!\!\rhd \cdots$$

and the term Δ never normalizes. ■

These examples show that reduction in the untyped calculus is more complicated than we might think at first sight. We won't pursue the ramifications of this, but we will make good use of the fixed point combinator Y in the next chapter.

Exercises

1.5 (a) Find a term (with empty support) for which there is at least two reduction paths one of which achieves a normal term in one step whereas the other goes on forever.

(b) Find a term Δ (with empty support) which does normalize (in a few number of steps) but for which $\Delta \rhd\!\!\rhd \Delta$ also holds.

(c) Find a term (with empty support) from which there are infinitely many (in fact continuum many) non-terminating reduction paths.

1.6 Using the confluence of the calculus (describe in the next section) show there is no normal term Z with $\mathsf{Y} \rhd\!\!\rhd Z$ (where Y is the Turing fixed point combinator).

1.7 Let $\mathsf{Y} = \lambda y\,.\,\Omega\Omega$ where $\Omega = \lambda x\,.\,y(xx)$. Show that for each term f there is a term F with $\mathsf{Y}f \rhd\!\!\rhd F \rhd\!\!\rhd fF$. (This Y is the fixed point combinator devised by Curry.)

1.4 Final remarks

There are some questions that should be asked, in fact should have been asked earlier. We will answer some of these but not in full.

How do we know that the process of replacing a redex subterm by its reduct does produce a term? At this stage we don't know that, and it has to be proved that it is so. This is not very difficult, but the proof doesn't give much enlightenment, so we won't bother to do it. Besides, we haven't formally defined 'subterm'.

In a term t which redex should we attack? We can have a go at any of them, which ever we think will be most useful. In other words, the reduction algorithm is non-deterministic. There may be many different reduction paths from a single term.

What happens if a term doesn't have a redex subterm? Nothing, in such a case the term is normal and can't be 'improved' by reduction.

Is it possible to start from a term and generate a reduction path which never ends? Yes, and we have see examples of this in Section 1.3. This is concerned with the normalization problem for the calculus.

Suppose we start from a single term, and generate two different reduction paths both of which lead to normal terms. What is the relationship between these terms?

They are α-variants of each other. This is a consequence of the confluence property of reduction. (Incidentally, a crucial component in the proof of confluence is something called Newman's lemma. Max Newman was an influential figure in the Mathematics Department in Manchester from 1946 to 1964 and was instrumental in bringing Turing to Manchester.)

Is it possible to have two reduction paths from a term one of which reaches a normal term and the other goes on for ever? Yes it is.

Chapter 2
Simulation of arithmetic in $\boldsymbol{\lambda 0}$

In Chapter 1 we set up the syntax and reduction mechanism of the untyped calculus. That mechanism is kind of abstract formalization of 'evaluation via substitution'. In this chapter we make precise this idea. We show how certain terms of the calculus can be used to capture numeric functions. Furthermore, the reduction mechanism provides algorithms for evaluating the captured functions.

Before we begin let's introduce a bit of useful notation. We are going to deal with natural numbers m and the arithmetic of these. There are many places where we have to look at the successor $m+1$ of such a number. The expanded version can make some expressions look a bit cluttered, so we write

$$m' \quad \text{for} \quad m+1$$

that is we use $(\cdot)'$ as the successor function.

2.1 The church numerals

We introduce some notation which generalizes that in the top block of Table 1.1

2.1 DEFINITION. For arbitrary terms t, s we generate

$$t^m s$$

the formal iterates of t on s by

$$t^0 s = s \qquad t^{m+1} s = t(t^m s)$$

for each $m \in \mathbb{N}$. Thus

$$t^m s = t(t(\cdots(t(ts))\cdots))$$

where there are m occurrences of t. ■

This abbreviation is in conflict with the informal convention for omitting brackets. This is not done to be perverse. Both conventions are the best way of doing their particular jobs. Notice that there is no term t^m; there is always a second component s.

A simple exercise shows

$$t^n(t^m s) = t^{m+n} s$$

that is, the two compounds are exactly the same term. We make good use of this.

2.2 DEFINITION. For each $m \in \mathbb{N}$ we set

$$\overline{m} = \lambda y, x \,.\, y^m x$$

to obtain the church numeral for m. ■

For instance

$$\overline{0} = \lambda y, x \,.\, x \quad \overline{1} = \lambda y, x \,.\, yx \quad \overline{2} = \lambda y, x \,.\, y(yx) \quad \overline{3} = \lambda y, x \,.\, y(y(yx))$$

and

$$\overline{4} = \lambda y, x \,.\, y^4 x \qquad \overline{2^4} = \lambda y, x \,.\, y^{16} x$$

and so on. These are the numerals in Table 1.1 and used in Example 1.2.

The term $\overline{m}$ is the canonical description of the number m, and we are going to do some arithmetic with these descriptions. Each numeral is normal (since it does not contain a redex). Notice also that

$$\overline{m}ts \rhd\!\rhd t^m s$$

for each $m \in \mathbb{N}$ and terms s, t. In other words $\overline{m}$ captures the notion of m-fold iteration.

(As an aside the terminology 'church numeral' has nothing to do with a religious organization. It was A. Church, one of the original developers of the λ-calculus, who first used the idea described in this section. There are two ways to encode the natural number m as a numeral, namely

$$\lambda y, x \,.\, y^m x \quad \text{and} \quad \lambda x, y \,.\, y^m x$$

which are the sacred and the profane versions. In the untyped calculus of this and the previous chapter the difference doesn't matter, but it becomes more important in the typed calculus of the next two chapter. It shouldn't surprise you that the profane version has better properties.)

In the central block of Table 1.1 we find four terms of a similar kind. For convenience let's repeat those constructions here.

2.3 DEFINITION. Let

$$\begin{aligned} \mathsf{S} &= \lambda u, y, x \,.\, y(uyx) \\ \mathsf{A} &= \lambda v, u, y, x \,.\, (vy)(uyx) \\ \mathsf{M} &= \lambda v, u, y, x \,.\, v(uy)x \\ \mathsf{E} &= \lambda v, u, y, x \,.\, vuyx \end{aligned}$$

to obtain four terms. ■

The letters chosen might give you a clue to the properties of these terms.

2.4 LEMMA. *For each $m, n \in \mathbb{N}$ the four reductions*

$$\mathsf{S}\,\overline{m} \rhd\!\rhd \overline{m+1} \qquad \mathsf{A}\,\overline{n}\,\overline{m} \rhd\!\rhd \overline{m+n} \qquad \mathsf{M}\,\overline{n}\,\overline{m} \rhd\!\rhd \overline{m \times n} \qquad \mathsf{E}\,\overline{n}\,\overline{m} \rhd\!\rhd \overline{m^n}$$

hold.

Proof. We have

$$\mathsf{S}\,\overline{m} \rhd \lambda y, x \,.\, y(\overline{m}yx) \rhd\!\rhd \lambda y, x \,.\, y(y^m x) = \lambda y, x \,.\, y^{m+1} x = \overline{m+1}$$

to give the first reduction.

Similarly we have

$$\mathsf{A}\,\overline{n}\,\overline{m} \rhd\!\rhd \lambda y, x \,.\, (\overline{n}y)(\overline{m}yx) \rhd\!\rhd \lambda y, x \,.\, y^n(y^m x) = \lambda y, x \,.\, y^{m+n} x = \overline{m+n}$$

to give the second reduction.

The other two reductions are proved in the same way. ■

You are warned about the omitted parts of this proof. They are not as straight forward as you might think. You should work out the missing details.

These four examples are fairly easy to understand. Let's now look at another example where it is almost impossible to fathom out what is happening. The following account makes no effort to explain the secret; that will be done later. The example is chosen to illustrate just how convoluted some reduction manipulations can be.

2.5 EXAMPLE. Let

$$\begin{aligned} \overline{d} &= \lambda x\,.\,x(-3-)(-2-)\overline{1} \\ \text{where}\quad (-3-) &= \lambda w,z,y,x\,.\,z(\lambda u\,.\,W)(yW) \\ W &= w\overline{0}yx \\ (-2-) &= \lambda z,y,x\,.\,z(\lambda u\,.\,x)x \end{aligned}$$

to obtain a term $\overline{d}$ and various subterms. There are multiple uses of some of the identifiers, but notice that the three subterms $(-3-), (-2-), \overline{1}$ are combinators. Also the subterm W does not contain the identifier u, and hence

$$(\lambda u\,.\,W)s \rhd\!\rhd W$$

for each term s.

This term $\overline{d}$ capture a 1-placed function. Which one? Let's do a few calculations.

For each $m \in \mathbb{N}$ we have

$$\overline{d}\,\overline{m} \rhd\!\rhd \overline{m}(-3-)(-2-)\overline{1} \rhd\!\rhd (-3-)^m(-2-)\overline{1}$$

which looks easy enough to handle.

For $m = 0$ we have

$$\begin{aligned} \overline{d}\,\overline{0} &\rhd\!\rhd (-3-)^0(-2-)\overline{1} \\ &= (-2-)\overline{1} \\ &\rhd\!\rhd \lambda y,x\,.\,\overline{1}(\lambda u\,.\,x)x \\ &= \lambda y,x\,.\,(\lambda u\,.\,x)x \\ &\rhd\!\rhd \lambda y,x\,.\,x \qquad\qquad = \overline{0} \end{aligned}$$

which is perhaps a bit of a surprise.

For $m = 1$ we have

$$\begin{aligned} \overline{d}\,\overline{1} &\rhd\!\rhd (-3-)^1(-2-)\overline{1} \\ &= (-3-)(-2-)\overline{1} \\ &\rhd\!\rhd \lambda y,x\,.\,\overline{1}(\lambda u\,.\,A)(yA) \\ &\rhd\!\rhd \lambda y,x\,.\,(\lambda u\,.\,A)(yA) \\ &\rhd\!\rhd \lambda y,x\,.\,A \end{aligned}$$

where

$$A = (-2-)\overline{0}yx \rhd\!\rhd \overline{0}(\lambda u\,.\,x)x \rhd\!\rhd x$$

so that

$$\overline{d}\,\overline{1} \rhd\!\rhd \lambda y,x\,.\,A \rhd\!\rhd \lambda y,x\,.\,x = \overline{0}$$

which is perhaps more of a surprise.

For $m = 2$ we have

$$\begin{array}{rcl} \overline{d}\,\overline{2} & \rhd\!\!\rhd & (-3-)^2(-2-)\overline{1} \\ & = & (-3-)((-3-)(-2-))\overline{1} \\ & \rhd\!\!\rhd & \lambda y, x\,.\,\overline{1}(\lambda u\,.\,B)(yB) \\ & \rhd\!\!\rhd & \lambda y, x\,.\,(\lambda u\,.\,B)(yB) \\ & \rhd\!\!\rhd & \lambda y, x\,.\,B \end{array}$$

where this time

$$B = (-3-)(-2-)\overline{0}yx \rhd\!\!\rhd \overline{0}(\lambda u\,.\,A)(yA) \rhd\!\!\rhd yA$$

where

$$A = (-2-)\overline{0}yx \rhd\!\!\rhd x$$

as above. This gives

$$\overline{d}\,\overline{2} \rhd\!\!\rhd \lambda y, x\,.\,B \rhd\!\!\rhd \lambda y, x\,.\,yA \rhd\!\!\rhd \lambda y, x\,.\,yx = \overline{1}$$

which doesn't make it crystal clear what is going on. ■

We could carry on in this way and calculate $\overline{d}\,\overline{3}$ (to get $\overline{2}$) and then $\overline{d}\,\overline{4}$ (to get $\overline{3}$) and so on. Eventually we might spot the trick.

2.6 LEMMA. *Using the notation of Example 2.5, we have*

$$(-3-)^m(-2-)\overline{0}yx \rhd\!\!\rhd y^m x$$

for each $m \in \mathbb{N}$.

Proof. We proceed by induction on m.

For the base case, $m = 0$, we have

$$(-3-)^0(-2-)\overline{0}yx = (-2-)\overline{0}yx \rhd\!\!\rhd \overline{0}(\lambda u\,.\,x)x \rhd\!\!\rhd x = y^0 x$$

as required.

For the induction step, $m \mapsto m'$, we have

$$(-3-)^{m'}(-2-)\overline{0}yx = (-3-)((-3-)^m(-2-))\overline{0}yx \rhd\!\!\rhd \overline{0}(\lambda u\,.\,M)(yM) \rhd\!\!\rhd yM$$

where

$$M = (-3-)^m(-2-)\overline{0}yx \rhd\!\!\rhd y^m x$$

by the induction hypothesis, to give the required result. ■

We can see a semblance of structure in this calculation, but not a lot. However, it does enable us to describe the behaviour of the strange term $\overline{d}$.

2.7 LEMMA. *Using the notation of Example 2.5, we have*

$$\overline{d}\,\overline{0} \rhd\!\!\rhd \overline{0} \qquad \overline{d}\,\overline{m'} \rhd\!\!\rhd \overline{m}$$

for each $m \in \mathbb{N}$.

Proof. Example 2.5 gives the left hand reduction. For the right hand reduction we have

$$\begin{array}{rcll} \overline{d}\,\overline{m'} & \rhd\!\!\rhd & (-3-)^{m'}(-2-)\overline{1} & \\ & = & (-3-)((-3-)^{m}(-2-))\overline{1} & \\ & \rhd\!\!\rhd & \lambda y, x\,.\,\overline{1}(\lambda u\,.\,M)(yM) & \\ & \rhd\!\!\rhd & \lambda y, x\,.\,(\lambda u\,.\,M)(yM) & \rhd\!\!\rhd \quad \lambda y, x\,.\,M \end{array}$$

where

$$M = (-3-)^{m}(-2-)\overline{0}yx \rhd\!\!\rhd y^{m}x$$

by Lemma 2.6, to give the required result. ■

This term $\overline{d}$ is a variant of the one used by Kleene to capture the predecessor function. Later we will see an explanation of how it works.

Exercises

2.1 The omitted parts of the proof of Lemma 2.4 requires a couple of inductions which are not as obvious as you might think. Set up and prove these inductions. Make sure you state the induction hypothese correctly.

2.2 Numerical negation N is the 1-placed function given by

$$N0 = 1 \qquad Nx' = 0$$

for $x \in \mathbb{N}$. Find a term $\overline{N}$ such that

$$\overline{N}\,\overline{m} \rhd\!\!\rhd \overline{Nm}$$

for each $m \in \mathbb{N}$.

2.3 Natural number substraction D is given by

$$Dxy = \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{if } x < y \end{cases}$$

for $x, y \in \mathbb{N}$. Find a term $\overline{D}$ such that

$$\overline{D}\,\overline{m}\,\overline{n} \rhd\!\!\rhd \overline{Dmn}$$

for all $m, n \in \mathbb{N}$.

2.2 Simulation of functions

If you have never seen results like Lemmas 2.4 and 2.7 before, they can look like magic. If you look a little closer at the proof of Lemma 2.4, you will see that it makes use of the way iterates of functions mesh together. You may then begin to suspect that the result is just a curiosity, or perhaps a small part of a whole family or rather simple descriptions of functions. However, the proof of Lemma 2.7 seems designed to contradict any kind of intuition you have. In this section we set down the general framework in which these manipulations are happening.

2.8 DEFINITION. Let f be an l-placed function on $\mathbb{N}$ and let $\overline{f}$ be an untyped term. We say $\overline{f}$ simulates f (within $\boldsymbol{\lambda}\mathsf{0}$) if the reduction

$$\overline{f}\,\overline{m_l}\cdots\overline{m_1} \twoheadrightarrow \overline{f(m_l,\ldots,m_1)}$$

holds for all $m_1,\ldots,m_l \in \mathbb{N}$.

Let $\mathcal{S}im$ be the class of functions that are simulated in this way. ■

In this terminology Lemma 2.4 says that the terms $\mathsf{S},\mathsf{A},\mathsf{M},\mathsf{E}$ simulate the Successor operation, Addition, Multiplication, and Exponentiation, respectively. There seems to be some reason within these terms. However, Lemma 2.7 says that the predecessor function can be simulated by a term in which it is hard to see any rhyme or reason. We need to investigate this notion further.

What other functions can be simulated? Let's look at a few more examples. The first one continues the sequence Successor, Addition, Multiplication, Exponentiation in a natural way.

2.9 DEFINITION. Let $\beth$ be the 3-placed function specified by

$$\beth(0,y,z) = z \qquad \beth(x',y,z) = y^{\beth(x,y,z)}$$

for $x,y,z \in \mathbb{N}$. This is the stacking function. ■

Pictorially we have

$$\beth(x,y,z) = y^{y^{\cdot^{\cdot^{\cdot^{y^z}}}}}$$

where there is a stack of x number of ys topped off with z. This explains the terminology. Often the central input is fixed at $y=2$ and the right hand input is fixed at $z=0$ or $z=1$. However, it is more convenient to have these inputs at our disposal.

If you completed the proof of Lemma 2.3 then you will have prove the next result already. You should make sure you can prove this, for it is not as easy as it looks.

2.10 LEMMA. *For all $m,k \in \mathbb{N}$ and all terms t,s we have $\overline{m}^k ts \twoheadrightarrow t^{m^k}s$.*

With this we can show that we have a simulation of the stacking function.

2.11 LEMMA. *Consider the term*

$$\mathsf{B} = \lambda u,v,w\,.\,uVw \quad \textit{where} \quad V = \lambda z,y,x\,.\,zvyx$$

as used in Example 1.13. Then

$$\mathsf{B}\,\overline{l}\,\overline{m}\,\overline{n} \twoheadrightarrow \overline{\beth(l,m,n)}$$

for all $l,m,n \in \mathbb{N}$, and the term B simulates the stacking function $\beth$.

Proof. We have

$$\mathsf{B}\,\overline{l}\,\overline{m}\,\overline{n} \rhd\!\rhd \overline{l}M\overline{n} \rhd\!\rhd M^l\overline{n} \quad \text{where} \quad M = \lambda z, y, x\,.\,z\overline{m}yx$$

so it suffices to show that

$$M^l\overline{n} \rhd\!\rhd\!\rhd \overline{\beth(l,m,n)}$$

hold. We proceed by induction on l.

For the base case, $l = 0$, we have

$$M^0\overline{n} = \overline{n} = \overline{\beth(l,m,n)}$$

(and this is the reason why the reflexive version $\rhd\!\rhd\!\rhd$ of $\rhd\!\rhd$ is used here).

For the induction step, $l \mapsto l'$, let $k = \beth(l,m,n)$. Then we have

$$\begin{array}{lcll} M^{l'}\overline{n} = M(M^l\overline{n}) & \rhd\!\rhd & \lambda y, x\,.\,(M^l\overline{n})\overline{m}yx & \\ & \rhd\!\rhd\!\rhd & \lambda y, x\,.\,\overline{k}\overline{m}yx & \\ & \rhd\!\rhd & \lambda y, x\,.\,\overline{m}^k yx & \\ & \rhd\!\rhd & \lambda y, x\,.\,y^{m^k}x & = \quad \overline{m^k} \end{array}$$

as required. The second reduction uses the induction hypothesis and the last uses Lemma 2.10. ■

The idea of simulation need not be restricted to numeric functions. We can also capture some boolean gadgetry.

2.12 EXAMPLE. Let

$$\mathsf{true} = \lambda y, x\,.\,y \qquad \mathsf{false} = \lambda y, x\,.\,x \qquad \mathsf{Zero} = \lambda x\,.\,x(\lambda u\,.\,\mathsf{false})\mathsf{true}$$

and

$$\mathsf{Cond} = \lambda z, y, x\,.\,zyx$$

to obtain four terms.

We have

$$\mathsf{Zero}\,\overline{0} \rhd\!\rhd \overline{0}(\lambda y\,.\,\mathsf{false})\mathsf{true} \rhd\!\rhd (\lambda y\,.\,\mathsf{false}))^0\mathsf{true} \rhd\!\rhd \mathsf{true}$$

and

$$\begin{array}{lcll} \mathsf{Zero}\,\overline{m'} & \rhd\!\rhd & \overline{m'}(\lambda y\,.\,\mathsf{false})\mathsf{true} & \\ & \rhd\!\rhd & (\lambda y\,.\,\mathsf{false}))^{m'}\mathsf{true} \rhd\!\rhd \mathsf{true} & \\ & = & (\lambda y\,.\,\mathsf{false})(-) & \rhd\!\rhd \quad \mathsf{false} \end{array}$$

where $(-)$ is some term (whose details don't matter). Thus Zero is a test for zero.

Next observe that

$$\mathsf{Cond}\,\mathsf{true}\,yx \rhd\!\rhd \mathsf{true}\,yx \rhd\!\rhd y \qquad \mathsf{Cond}\,\mathsf{false}\,yx \rhd\!\rhd \mathsf{false}\,yx \rhd\!\rhd x$$

so that Cond can be read

$$\text{if } z \text{ then } y \text{ else } x \text{ fi}$$

that is the term acts as a conditional. ■

It is a common practice to code such boolean gadgetry by numeric functions. This is not a good idea for any kind of coding tends to obscure what is going on. Nevertheless, it is instructive to see how it can be done. We use 0 as a code for true and 1 as a code for false, and re-work Example 2.12 as follows.

2.13 EXAMPLE. Let

$$\mathsf{true} = \overline{0} = \lambda y, x \,.\, x \qquad \mathsf{false} = \overline{1} = \lambda y, x \,.\, yx \qquad \mathsf{Zero} = \lambda x \,.\, x(\lambda u \,.\, \overline{1})\overline{0}$$

and

$$\mathsf{Cond} = \lambda z, y, x \,.\, z(\lambda u \,.\, x)y$$

and to obtain four terms. We find that Zero is a test for zero and Cond simulates the function

$$\text{if } z = 0 \text{ then } y \text{ else } x \text{ fi}$$

that is, the numeric conditional. ■

We have seen examples of simple functions that can be simulated in a simple way (namely, Successor, Addition, Multiplication, and Exponentiation). We have seen at least one example of a simple function that is simulated in a complicated way (namely, the Predecessor function). Also we have had an indication that at least one rather fast growing function can be simulated in a reasonable way (namely, the stacking function). How extensive is the class $\mathcal{S}im$ of simulated functions?

There is an obvious result to get out of the way.

2.14 LEMMA. *The class $\mathcal{S}im$ is closed under composition.*

Proof. Consider the composition

$$f = h \circ (g_k, \ldots, g_1)$$

where h is k-placed, as indicated, and each g_i is l-placed. In particular, f is l-placed. Suppose $g_1, \ldots, g_k, h$ are simulated by terms $\overline{g_1}, \ldots, \overline{g_k}, \overline{h}$, respectively. Thus

$$\overline{g_i}\,\overline{m_l} \cdots \overline{m_1} \rhd\!\rhd \overline{g_i(m_l, \ldots, m_1)} \qquad \overline{h}\overline{n_k} \cdots \overline{n_1} \rhd\!\rhd \overline{h(n_k, \ldots, n_1)}$$

for all $m_1, \ldots, m_l, n_1, \ldots, n_k \in \mathbb{N}$. It is now routine to show that the term

$$\overline{f} = \lambda u_l, \ldots, u_1 \,.\, \overline{h}(\overline{g_k}u_l \cdots, u_1) \cdots (\overline{g_1}u_l \cdots, u_1)$$

simulates f. ■

There is an obvious and rather rather crude upper bound on the class of simulated functions. Suppose the l-placed function f is simulated by a term $\overline{f}$. Then there is an effective algorithm by which we can evaluate f. To determine the value $f(m_l, \ldots, m_1)$ we use the term

$$t = \overline{f}\,\overline{m_l} \cdots \overline{m_1}$$

which we know must normalize to the value $\overline{f(m_l, \ldots, m_1)}$. Thus we start to reduce t in all possible ways, that is we generate all possible reduction paths which start at t. We know that at least one of these paths must terminate, and any path that does terminate will finish with the same numeral. It is possible that some paths do not terminate. However, we generate all possible paths in the sure knowledge that one will be successful and give us the required value.

This proves the following.

2.15 THEOREM. *Each function in $\mathcal{S}im$ is Turing computable, that is general recursive.*

Strictly speaking we have *not* proved this. What we have done is show that each such function has an effective evaluation algorithm. To prove the Theorem we should turn that informal algorithm into a formal description of a general recursive function. However, we don't learn much by doing that, so we won't bother.

A more important question is whether we can lower this upper bound, and locate within the general recursive functions the precise class of simulated functions.

Exercises

2.4 Consider the 3-placed function $\aleph$ specified by

$$\aleph(z, y, 0) = z \qquad \aleph(z, y, x') = \aleph(y^z, y, x)$$

for $x, y, z \in \mathbb{N}$. Describe this function (in terms of some other well known function) and give a formal proof of your claim.

2.5 Find terms which simulate each of the functions

$$(x + y)^z \qquad x^{y+z} \qquad x^{y \times z}$$

(for $x, y, z \in \mathbb{N}$).

2.6 Consider the two terms

$$\begin{array}{lll} \mathsf{B}_1 = \lambda u, v, w \,.\, uVw & \text{where} & V = \lambda z, y, x \,.\, zvyx \\ \mathsf{B}_2 = \lambda u, v, w \,.\, \lambda y, x \,.\, uVwyx & \text{where} & V = \lambda u \,.\, uv \end{array}$$

(where you should recognize the first). Show that each term represents the stacking functions. Compare the two proofs. Can you say which is the 'better' representation?

2.3 Closure under primitive recursion

How big is the class $\mathcal{S}im$ of functions that can be simulated in the untyped calculus? We have seen two extremes. On the one hand we know that some rather simple functions belong to $\mathcal{S}im$, and on the other hand we know that $\mathcal{S}im \subseteq \mathcal{GR}$, the class of general recursive functions. We know also that $\mathcal{S}im$ is a clone, that is it contains all projections functions and is closed under composition. In this and the next section we consider other closure properties of $\mathcal{S}$. We show that if the function f is built in a certain way from functions known to be in $\mathcal{S}im$, then $f \in \mathcal{S}im$. More importantly, we obtain results of the following kind.

> Suppose the function f is specified in terms of function $g, h, k \ldots$ in $\mathcal{S}im$. Let $\overline{g}, \overline{h}, \overline{k}, \ldots$ be terms which simulate these data functions. Then we have $f \in \mathcal{S}im$. Furthermore, the specification of f provides a recipe by which the terms $\overline{g}, \overline{h}, \overline{k}, \ldots$ can be used to construct a term $\overline{f}$ which simulates f.

As an example of this kind of construction we will see how the silly term $\overline{d}$ of Example 2.5 can be produced. The details for this are outlined in Exercises 2.7 – 2.10.

In this section we consider the specification method of primitive recursion. Let's review how this works.

We are given two functions

$$g : \mathbb{N}^l \longrightarrow \mathbb{N} \qquad h : \mathbb{N}^l \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

which we conveniently refer to as the data functions. We wish to produce a function

$$f : \mathbb{N}^l \times \mathbb{N} \longrightarrow \mathbb{N}$$

in a certain way. These three functions have typical values

$$g(y_l, \ldots, y_1) \qquad h(y_l, \ldots, y_1, x, t) \qquad f(y_l, \ldots, y_1, x)$$

for $x, y_1, \ldots, y_l, t \in \mathbb{N}$. The input x to f plays a special role, it is the recursion input. The other inputs $y_1, \ldots, y_l$ are the parameters and in this construction play a passive role. To highlight what is going on we collapse the list of parameters to a single input

$$p = [y_l, \ldots, y_1]$$

so that

$$gp \qquad h(p, x, t) \qquad f(p, x)$$

are typical values of the three functions. This is why the source types of the three functions are punctuated as they are.

Here is the formal definition.

2.16 DEFINITION. A $(l+1)$-placed function

$$f : \mathbb{N}^l \times \mathbb{N} \longrightarrow \mathbb{N}$$

is obtained by primitive recursion from the data functions

$$g : \mathbb{N}^l \longrightarrow \mathbb{N} \qquad h : \mathbb{N}^l \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

if

$$f(p, 0) = gp \qquad f(p, x') = h(p, x, f(p, x))$$

holds for all $x \in \mathbb{N}$ and list $p \in \mathbb{N}^l$ of parameters. ■

In this definition the rightmost input to f is the recursion input. There are, of course, other variants of this idea. The recursion input could appear in any position. The inputs to the data functions could be mixed about. However, such permutations are easily handled using projections functions. When analysing the global properties of recursion constructions it is convenient to have the recursion input at one end or the other of the list of inputs. Furthermore, it is often instructive to try both versions; sometimes things are clearer in one version than the other.

The purpose of this section is to show how simulation $\overline{g}, \overline{h}$ of g, h can be converted into a simulation $\overline{f}$ of f.

As with almost all of these conversion results the trick is to first rephrase the given specification in the form of an iteration. We then use church numerals to capture that iteration. In general, such an iteration does not manipulate single numbers but tuples of numbers. This means we need some pairing gadgetry.

2.17 DEFINITION. Three terms

$$\mathsf{Pair} \qquad \mathsf{Left} \qquad \mathsf{Right}$$

act as pairing gadgets if the reductions

$$\mathsf{Left}(\mathsf{Pair}lr) \rhd\!\rhd l \qquad \mathsf{Right}(\mathsf{Pair}lr) \rhd\!\rhd r$$

hold for all terms l, r. ■

There are various ways of producing pairing gadgets within the untyped calculus. We have, more or less, seen one way of doing this already (although this wasn't pointed out at the time). We are not going to select any particular collection of pairing gadget. This is because the inner details of the three terms can confuse the understanding of the overall process (and again we have seen an example of this already).

We use a standard method of converting a primitive recursion into an iteration to prove the following.

2.18 THEOREM. *The class $\mathcal{S}im$ is closed under primitive recursion.*

Proof. Consider a primitive recursive specification

$$f(p,0) = gp \qquad f(p,x') = h(p,x,f(p,x))$$

of a function f in terms of two functions g, h. Here x is the recursion input (and we have chosen to display it as the right-most input of f). The other input p to f is a list of fixed length of parameters each of which is a natural number. In this account we will treat p as a single natural number.

We assume the two data functions g, h are simulated by terms $\overline{g}$ and $\overline{h}$, respectively. Thus

$$\overline{g}\,\overline{p} \rhd\!\rhd \overline{gp} \qquad \overline{h}\,\overline{p}\,\overline{m}\,\overline{t} \rhd\!\rhd \overline{h(p,m,t)}$$

for all parameters p and natural numbers m, t.

Using pairing gadgets let

$$\overline{H} = \lambda p, w\,.\,\mathsf{Pair}(\overline{\mathsf{S}}l)(\overline{h}plr) \quad \text{where } l = \mathsf{Left}w \text{ and } r = \mathsf{Right}w$$

to obtain an auxiliary term. We check that the following hold for all natural numbers m, l, r and parameters p.

$$\begin{array}{lr} (i) & \overline{H}\,\overline{p}\,(\mathsf{Pair}\,\overline{l}\,\overline{r}) \rhd\!\rhd \mathsf{Pair}\,\overline{l'}\,\overline{h(p,l,r)} \\ (ii) & \overline{H}\,\overline{p}\,(\mathsf{Pair}\,\overline{l}\,\overline{f(p,l)}) \rhd\!\rhd \mathsf{Pair}\,\overline{l'}\,\overline{f(p,l')} \\ (iii) & \overline{H}^m\,\overline{p}\,(\mathsf{Pair}\,\overline{l}\,\overline{f(p,l)}) \rhd\!\rhd \mathsf{Pair}\,\overline{l+m}\,\overline{f(p,l+m)} \\ (iv) & \overline{H}^m\,\overline{p}\,(\mathsf{Pair}\,\overline{0}\,\overline{gp}) \rhd\!\rhd \mathsf{Pair}\,\overline{m}\,\overline{f(p,m)} \\ (v) & \overline{m}\,\overline{H}\,\overline{p}\,(\mathsf{Pair}\,\overline{0}\,\overline{gp}) \rhd\!\rhd \mathsf{Pair}\,\overline{m}\,\overline{f(p,m)} \end{array}$$

Using $\overline{H}$ let

$$\overline{f} = \lambda p, x\,.\,\mathsf{Right}(x\overline{H}pw) \quad \text{where } l = \mathsf{Left}\,w(\mathsf{Pair}\overline{0}(\overline{g}p))$$

to obtain a second term. A use of (v) gives

$$\overline{f}\,\overline{p}\,\overline{m} \rhd\!\rhd \mathsf{Right}(\overline{m}\,\overline{H}\,\overline{p}\,(\mathsf{Pair}\,\overline{0}\,\overline{gp}) \rhd\!\rhd \mathsf{Right}(\mathsf{Pair}\,\overline{m}\,\overline{f(p,m)}) \rhd\!\rhd \overline{f(p,m)}$$

to show that $\overline{f}$ simulates f. ■

The class $\mathcal{PR}$ of primitive recursive functions is the least clone which contains certain initial functions and is closed under primitive recursion. It is more or less a triviality that the initial functions are in $\mathcal{S}im$, and we know that $\mathcal{S}im$ is a clone. Theorem 2.18 shows that $\mathcal{PR} \subseteq \mathcal{S}im$.

There are some rather weird functions in $\mathcal{PR}$, and each one of these can be simulated in the untyped calculus. For instance, the function which enumerates the primes in ascending order is primitive recursive, and hence can be simulated in the calculus. Can you imagine what a simulating term might look like?

For another example, consider first order peano arithmetic. The proof predicate of that system can be describe by a primitive recursive function. Thus there is a way of capturing first order number theory within the calculus. Since the vast majority of proofs within mathematics can be rephrase as proofs in first order number theory, we begin to see that the untyped calculus is no pussy cat.

In fact, these two functions (prime enumeration and proof predicate) are Kalmár elementary, a class of functions which form a very small and rather simple part of $\mathcal{PR}$.

Exercises

Thes four exercises explains the silly looking term $\overline{d}$ of Example 2.5 which simulates the predecessor function d. It make use of Kleene's trick which is the topic of the first exercise.

2.7 Consider the function $D : \mathbb{N}^2 \longrightarrow \mathbb{N}^2$ given by

$$D(x, y) = (x', x)$$

for $x, y \in \mathbb{N}$. Show that

$$D(x, dx) = (x', dx') \qquad D^x(0, 0) = (x, dx)$$

(for $x \in \mathbb{N}$) and hence explain how d can be evaluated by iteration.

2.8 (a) Consider the terms

$$\mathsf{P} = \lambda l, r, z \,.\, zlr \qquad \mathsf{L} = \lambda w \,.\, w\mathsf{t} \quad \mathsf{R} = \lambda w \,.\, w\mathsf{f}$$

where

$$\mathsf{t} = \lambda y, x \,.\, y \qquad \mathsf{f} = \lambda y, x \,.\, x$$

are the auxiliary terms. Show that these form pairing gadgets.

(b) Consider the terms

$$\mathsf{P} = \lambda l, r, z, y, x \,.\, z(\lambda u \,.\, ryx)(lyx) \qquad \mathsf{L} = \lambda w \,.\, w\overline{0} \quad \mathsf{R} = \lambda w \,.\, w\overline{1}$$

which use the first two numerals. Do these form pairing gadgets?

2.9 Using a suitable set of pairing gadgets let

$$\overline{(m, n)} = \mathsf{P}\,\overline{m}\,\overline{n}$$

for each $m, n \in \mathbb{N}$, and let

$$\mathsf{D} = \lambda w \,.\, \mathsf{P}(\mathsf{S}(\mathsf{L}w))(\mathsf{L}w) \qquad \mathsf{d} = \lambda x \,.\, \mathsf{R}(x\mathsf{D}\overline{(0,0)})$$

where S simulates the sucessor function. Show that

$$\mathsf{D}\overline{(m,n)} \rhd\!\rhd \overline{(m',m)}$$

(for each $m, \in \mathbb{N}$) and hence show that d simulates d.

2.10 Recall the term $\overline{d}$ of Example 2.5. Using a suitable version of the terms D and d of Exercise 2.9, show that

$$\overline{(0,0)} \rhd\!\rhd (-2-) \qquad \mathsf{D} \rhd\!\rhd (-3-) \qquad \mathsf{d} \rhd\!\rhd \overline{d}$$

and hence explain how $\overline{d}$ simulates d.

2.4 Closure under unbounded search

The technique of the proof of Theorem 2.18 can be used to show that $\mathcal{S}im$ is closed under many other forms of recursion. In each case we take the recursion, rephrase it as an iteration, and then simulate this using church numerals to get at the iteration. In the final analysis, this is the technique we need to know about because it does more than just produce simulation. It converts a recursive specification of a function into an evaluation algorithm which can be performed by reduction. We are not going to pursue these ideas here, not because they are unimportant, they are. But they deserve more time and space than we have available.

In this section we locate the extremes of $\mathcal{S}im$. We know that $\mathcal{PR} \subseteq \mathcal{S}im \subseteq \mathcal{GR}$, and the remarks above indicate that $\mathcal{PR}$ is not a very good lower bound. In this section we show that, in fact $\mathcal{S}im = \mathcal{GR}$.

We use the well known characterization of $\mathcal{GR}$, that is $\mathcal{GR}$ is the smallest clone which include certain initial functions and is closed under unbounded search. The initial functions are certainly contained in $\mathcal{PR}$, so the job of this section is to show how unbounded search can be handled in the calculus.

This raises a problem which so far we have ignored and will be able to ignore later but it is important here. So far we have dealt entirely with total functions, but $\mathcal{GR}$ is a class of *partial* functions. Each member f of $\mathcal{GR}$ has the form

$$f : \mathbb{S} \longrightarrow \mathbb{N} \quad \text{where} \quad \mathbb{S} \subseteq \mathbb{N}^l$$

for some $l \in \mathbb{N}$. Informally we are a little less precise and we say f is a partial function of type $\mathbb{N}^l \longrightarrow \mathbb{N}$. We need to have another look at the notion of simulation. You should compare the following with Definition 2.8

2.19 DEFINITION. Let f be a partial l-placed function on $\mathbb{N}$. Let $\overline{f}$ be an untyped term. We say $\overline{f}$ simulates f (in $\boldsymbol{\lambda}0$) if for all $m_1, \ldots, m_l \in \mathbb{N}$ both

- If $(m_l, \ldots, m_1)$ is in the domain of definition of f then

$$\overline{f}\,\overline{m_l} \cdots \overline{m_1} \rhd\!\rhd \overline{f(m_l, \ldots, m_1)}$$

- If $(m_l, \ldots, m_1)$ is not in the domain of definition of f then the term $\overline{f}\overline{m_l}\cdots\overline{m_1}$ does not normalize.

hold. ■

We may also want to add a further restriction on the term $\overline{f}$. For instance we may want to insist that if $(m_l, \ldots, m_1)$ is in the domain of definition of f then every reduction path from the compound term $\overline{f}\overline{m_l}\cdots\overline{m_1}$ does terminate. We will not worry about that here.

Recall how unbounded search works.

Suppose we have a partial function

$$h : \mathbb{N} \times \mathbb{N}^l \longrightarrow \mathbb{N}$$

(with some domain of definition). Using this we obtain a partial function

$$f : \mathbb{N}^l \longrightarrow \mathbb{N}$$

where

$$f(x_l, \ldots, x_1) = (\mu y)[h(y, x_l, \ldots, x_1) = 0]$$

for all $(x_l, \ldots, x_1)$ in the domain of definition of f. In other words this value is the least natural number y such that

$$h(y, x_l, \ldots, x_1) = 0$$

if there is such a y, otherwise $f(x_l, \ldots, x_1)$ is undefined. Notice that even if h is a total function, the specified function f may not be total (since the required solution y may not exists).

We will show how to translate this unbounded search into a term of the calculus. As we do this the inputs $x_l, \ldots, x_1$ play a rather passive role, so it is convenient to collapse them to one input x and treat this as a single number. This prevents the notation becoming too cluttered with irrelevant subscripts.

Thus we show how to simulate the function f given by

$$fx = (\mu y)[h(y, x) = 0]$$

where h is a (partial) function that can be simulated. Let $\overline{h}$ be any term which simulates h. Thus

- If (n, m) is in the domain of definition of h then $\overline{h}\,\overline{n}\,\overline{m} \rhd\!\rhd \overline{h(n, m)}$.
- Otherwise, the term $\overline{h}\,\overline{n}\,\overline{m}$ does not normalize.

We use the boolean gadgetry of Example 2.12 (which is one reason it was introduced). However, here we will clean it up a little. Thus let

$$k = \lambda H, y, x\,.\,\Big((\overline{h}yx)(\lambda u\,.\,\mathsf{false})\mathsf{true}\Big)y\Big(H(\mathsf{S}y)x\Big)$$

where true and false are the terms of Example 2.12 and S is the simulation of the successor function. Suppose $h(n, m)$ is defined. Then

$$kH\,\overline{n}\,\overline{m} \rhd\!\rhd \Big(\overline{h(n,m)}(\lambda u\,.\,\mathsf{false})\mathsf{true}\Big)\overline{n}\Big(H\,\overline{n'}\,\overline{m}\Big) \rhd\!\rhd \begin{cases} \mathsf{true}\,\overline{n}\Big(H\,\overline{n'}\,\overline{m}\Big) & \text{if } h(n,m) = 0 \\ \mathsf{false}\,\overline{n}\Big(H\,\overline{n'}\,\overline{m}\Big) & \text{if } h(n,m) \neq 0 \end{cases}$$

and hence

$$kH\,\overline{n}\,\overline{m} \rhd\!\rhd \begin{cases} \overline{n} & \text{if } h(n,m) = 0 \\ H\,\overline{n'}\,\overline{m} & \text{if } h(n,m) \neq 0 \end{cases}$$

using the selection properties of true and false. Thus, informally

$$kHyx$$

can be read as

$$\text{if } h(y,x) = 0 \text{ then } y \text{ else } H(y',x)$$

provided we take a bit of care.

Next we remember the Turing fixed point combinator Y of Definition 1.14 and Lemma 1.15. With this let

$$F = \mathsf{Y}k$$

so that

$$F \rhd\!\rhd kF$$

and hence

$$F\,\overline{n}\,\overline{m} \rhd\!\rhd \begin{cases} \overline{n} & \text{if } h(n,m) = 0 \\ F\,\overline{n'}\,\overline{m} & \text{if } h(n,m) \neq 0 \end{cases}$$

for all $m, n \in \mathbb{N}$ where $h(n,m)$ is defined.

Now suppose

$$h(0,m), h(1,m), h(2,m), \ldots$$

are defined, and the first few are not zero. Then

$$F\,\overline{0}\,\overline{m} \rhd\!\rhd F\,\overline{1}\,\overline{m} \rhd\!\rhd F\,\overline{2}\,\overline{m} \rhd\!\rhd \cdots$$

and this reduction continues until it finds $n \in \mathbb{N}$ with $h(n,m) = 0$. Thus, with

$$\overline{f} = F\overline{0}$$

we have more or less proved the following.

2.20 THEOREM. *The class $\mathcal{S}im$ is closed under unbounded search.*

In fact, we have not quite proved this. In the calculations above we did not consider what happens if fm is undefined. For that case we ought to show that the term $\overline{f}\,\overline{m} = F\,\overline{0}\,\overline{m}$ does not normalize. This is easy enough when h is total, but more complicated when h is partial. In fact, with F as constructed this required non-termination can not be guaranteed. A slight modification of F does ensure the required property, but the proof depends on a rather detailed analysis of reduction properties in the calculus. For what we need here that is not necessary.

Exercises

2.11 Show that the natural number function $x \longmapsto \lfloor \log x \rfloor$ can be simulated in the calculus (where the logarithm is to base 2). Have you any idea what the simulating term looks like?

2.5 Final remarks

The basic syntax and reduction mechanism of the untyped λ-calculus is deceptively simple. Nevertheless the calculus is very powerful. It is Turing complete in the sense that it captures every general recursive function. It can compute everything that anything can compute. In fact, we have seen an indication of something better. Each recursive specification of a function f can be converted into a simulating term $\overline{f}$, and the construction of $\overline{f}$ follows quite closely the given specification. Because of this the evaluation algorithm using $\overline{f}$ (and the reduction mechanism) has a relationship with the algorithm embedded in the specification. There are some differences, but these seem to arise from various coding tricks that are needed (for instance, to obtain pairing gadgets, boolean gadgets, and the like).

As it turns out the untyped calculus is a little (in fact, a lot) too free and easy. It is too undisciplined. We are going to correct this with a more refined calculus. In consequence, we find that an evaluation algorithm using reduction matches very closely the algorithm embedded in the parent specification. Furthermore, we can use the syntax of the refined calculus to organize a fine grained classification of the complexity of certain functions.

Chapter 3

The simply typed calculus $\boldsymbol{\lambda 1}$

The untyped λ-calculus has a deceptively simple structure, and yet can it simulate any Turing computable function. In other words the calculus is a powerful as any Turing machine. It is as powerful as any *universal* Turing machine. The untyped calculus is extremely powerful, but where does this power come from? The same place that makes some aspects of the untyped calculus hard to analyse.

The crucial facilities of the untyped calculus are the fixed point combinators, and these are available because the calculus is rather free and easy with the notion of functionality. We are going regulate this by making the calculus conform to a quite reasonable set of restrictions, and then we find that its power is severely limited.

3.1 Types, terms, and reduction

For the purposes of these notes we think of the untyped calculus as an attempt to capture the notion of 'functionality' and the process of 'evaluation'. Thus we think of an untyped term t as a name of a function. More precisely, we think of a term t as a description of the behaviour of a function, a method by which it converts its input into its output. In particular, for a term s, which we think of as a name of an input, the compound ts is a name of the output (which might be simplified by reduction). However, there is clearly something missing here, and we need to rectify that.

Recall that a function f is determined by three facets, its source $\mathbb{S}$, its target $\mathbb{T}$, and its behaviour

$$\begin{array}{ccc} \mathbb{S} & \xrightarrow{\quad f \quad} & \mathbb{T} \\ s & \longmapsto & \text{expression in } s \end{array}$$

a description of the way it converts an input into an output. The source $\mathbb{S}$ is the place where it takes it inputs from (and we assume that the function is total, that is any member of $\mathbb{S}$ is an acceptable input). The target $\mathbb{T}$ is the place the function sends it outputs to (and it could be that some members of $\mathbb{T}$ can not be achieved in this way). Together the source and target (in that order) form the type of the function.

The behaviour of the function is what happen between the source and the target, the process of converting the input into the output. The terms of the untyped calculus give us a way of expressing such a behaviour. They are like polynomials or other standard descriptions of functions, except they are much more flexible and can describe many more functions. Often we think of the behaviour of a function as happening instantaneously. However, when we begin to analyse the complexity of a function, we try to think of the behaviour as some kind of process or procedure which uses up some resources. The reduction mechanism of the calculus is a way of getting at this process.

The terms of the untyped calculus provide a formal way of writing down the behaviour of a function, but they omit some crucial information. Where can we find the

source and the target? The simply typed λ-calculus is a refinement of the untyped calculus in which the source and target of a term are made explicit. To do that we use a new family of syntactic expressions which describe these gadgets.

3.1 DEFINITION. The types of the simply typed λ-calculus are generated from an unlimited stock of variables using two rules of construction (a base and a step).

- Each variable is a type.
- If ρ, σ are types, then so is $(\sigma \to \rho)$.

There are no other types. ■

It is convenient to let

$$\begin{array}{ll} X, Y, Z, \ldots & \text{range over variables} \\ \rho, \sigma, \tau \ldots & \text{range over types} \end{array}$$

but we can break this informal convention if it suits us.

The brackets in a type are part of its official construction. They are there to ensure that it is uniquely parsed. However, to help make some types more legible we often leave out some brackets. We will discuss the informal conventions for the later on (just after Definition 3.2).

You should not confuse variables and identifiers. The first are the building blocks of types and the second are (and will be) the building blocks of terms. This is the reason why 'identifier' is used rather than the more common 'term variable'.

The remainder of this section will follow quite closely the development in Sections 1.1 and 1.2. In fact, you will begin to wonder what use these types are since, in this section, they play no significant role. Their job in life and how they do it will be explained in the next section.

Compare the following with Definition 1.1.

3.2 DEFINITION. The raw terms of the simply typed λ-calculus are generated from an unlimited stock of identifiers using three rules of construction (a base and two steps).

- Each identifier is a raw term.
- If q, p are raw terms, then the application (qp) is a raw term.
- If x is an identifier, σ is a type, and r is a raw term, then the abstraction $(\lambda x : \sigma \,.\, r)$ is a raw term.

There are no other terms. ■

At first you might not spot the difference. Each abstraction contains a type. Thus the untyped and the typed abstractions are

$$(\lambda x \,.\, r) \qquad (\lambda x : \sigma \,.\, r)$$

respectively. If we take a typed raw term and erase all the types then we get an untyped term. Conversely if we take an untyped term and scatter various types throughout it at the abstractions then we get a typed raw term. Clearly by choosing different types we can obtain many raw terms from the same untyped term.

$\overline{0}_\bullet$	$(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,x))$	$\lambda y:\eta, x:\zeta\,.\,x$
$\overline{1}_\bullet$	$(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,(yx)))$	$\lambda y:\eta, x:\zeta\,.\,yx$
$\overline{2}_\bullet$	$(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,(y(yx))))$	$\lambda y:\eta, x:\zeta\,.\,y(yx)$
$\overline{3}_\bullet$	$(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,(y(y(yx)))))$	$\lambda y:\eta, x:\zeta\,.\,y(y(yx))$
$\mathsf{S}_\bullet$	$(\lambda u:\theta\,.\,(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,(y((uy)x)))))$	$\lambda u:\theta, y:\eta, x:\zeta\,.\,y(uyx)$
$\mathsf{A}_\bullet$	$(\lambda v:\beta\,.\,(\lambda u:\theta\,.\,(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,((vy)((uy)x))))))$	$\lambda v:\beta, u:\theta, y:\eta, x:\zeta\,.\,(vy)(uyx)$
$\mathsf{M}_\bullet$	$(\lambda v:\beta\,.\,(\lambda u:\theta\,.\,(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,(v(uy))x))))$	$\lambda v:\beta, u:\theta, y:\eta, x:\zeta\,.\,v(uy)x$
$\mathsf{E}_\bullet$	$(\lambda v:\beta\,.\,(\lambda u:\theta\,.\,(\lambda y:\eta\,.\,(\lambda x:\zeta\,.\,(((vu)y)x)))))$	$\lambda v:\beta, u:\theta, y:\eta, x:\zeta\,.\,vuyx$

Table 3.1: Some examples of terms (combinators)

The point of this will become clear in the next section.

As with untyped terms it is convenient to let

$x, y, z, \ldots$	range over identifiers
$p, q, r, s, t \ldots$	range over raw terms

and we will use the same conventions for omitting some brackets.

The simply typed λ-calculus uses types and raw terms. Both of these syntactic gadgets contain brackets (to ensure that each is uniquely parsed). However, in practice some of these brackets are omitted (to improve the legibility). The informal conventions for omitting brackets from raw terms are the same as for untyped terms. The informal conventions for omitting brackets from types are slightly different.

Consider the following.

Type	Term
$\tau \to \sigma \to \rho$	tsr
$(\tau \to (\sigma \to \rho))$	$((ts)r)$

On the bottom line we have a type and a raw term built according to the official rules. Above that we have each with the brackets omitted. The crucial difference is that for types the brackets are reinstated from the right whereas for raw terms the brackets are reinstated from the left. This convention is not chosen to be perverse but because it matches well with currying.

To illustrate this think informally for a moment. Suppose we have a function

$$f : \tau \to \sigma \to \rho \to \zeta$$

in curried form. Thus

$$(\tau \to (\sigma \to (\rho \to \zeta)))$$

is the type of this function with the brackets reinstated. Now let's evaluate this function at successive arguments

$$t : \tau \quad s : \sigma \quad r : \rho$$

to get

$$(ft) \quad ((ft)s) \quad (((ft)s)r)$$

in turn. Here we have inserted brackets in the natural order. When these are omitted we see that

$$ftsr$$

is the eventual output of f at these inputs.

In short, the two informal conventions complement each other in a natural way.

Let's look at some examples of raw terms. In fact, we can use typed versions of some of the terms in Table 1.1.

3.3 EXAMPLES. Table 3.1 gives some examples of raw terms using four types $\zeta, \eta, \theta, \beta$ (which need not be distinct). Each raw term has be given a subscript $(\cdot)_\bullet$ to distinguish it from its type erased version (which appears in Table 1.1).

As in that table the official construction is given in the middle column, and then the informal, more legible version is given in the right column (using the conventions for omitted brackets). ■

We carry over some of the notions and notations from the untyped calculus. Thus the support ∂t of a raw term is just the finite set of identifiers which occur free in the raw term. This can be generated as in Definition 1.6. A typed combinator is a raw term with empty support. Two raw terms are α-equivalent if they differ only in the choice of bound identifiers.

The notions of substitution and reduction for raw terms can be described almost word for word as for untyped terms.

3.4 DEFINITION. (Not quite) For all raw terms r, s and each identifier x

$$r[x := s]$$

is the term obtained from r by replacing all free occurrences of x by s, and taking due precautions not to catch anything. ■

This can be made more precise as in Definition 1.9 using any of the common substitution algorithms. As with untyped terms, in these notes we need not get into the details of this.

We set up the reduction mechanism in the same way.

3.5 DEFINITION. A redex or reducible expression of the simply typed λ-calculus is a compound raw term

$$(\lambda x : \sigma \,.\, r)s$$

formed by an outer application and an inner abstraction.

The immediate reduct of the redex

$$r[x := s]$$

is obtained by performing the indicated substitution.

A 1-step reduction

$$(\lambda x : \sigma \,.\, r)s \rhd r[x := s]$$

converts a redex into its reduct. ■

This is just the same as Definition 1.10 with a couple of types appearing. The notion of reduction is exactly the same.

3.6 DEFINITION. For raw terms t^- and t^+ we write

$$t^- \rhd\!\!\rhd t^+$$

and say t^- reduces to t^+ if we can move from t^- to t^+ in a finite number of steps where at each step we replace a redex subterm of the current term by its immediate reduct (or take an α-variant).

A term is normal if no subterm is a redex.

Let $\rhd\!\!\rhd\!\!\rhd$ be the reflexive closure of $\rhd\!\!\rhd$. Thus

$$t^- \rhd\!\!\rhd\!\!\rhd t^+$$

holds (for terms t^-, t^+) precisely when either t^- and t^+ are the same term (up to α-equivalence) or $t^- \rhd\!\!\rhd t^+$. ■

What is the point of these types? As yet they seem no more than a way of cluttering up the notation without any benefits. This is illustrated by the Exercises 3.2 and 3.3. Of course, the types are there to impose a typing discipline on the raw terms. Using this we see that some raw terms don't come up to scratch, and need to be eliminated from the contest.

Exercises

3.1 For each typed term t there is an untyped term t^ϵ obtained by removing all the embedded types. This process $(\cdot)^\epsilon$ is called type erasure. Write down a recursive description of $(\cdot)^\epsilon$.

3.2 Consider the untyped reductions in Lemma 2.3. Show that exactly the same reductions can be perfored using the raw terms of Table 3.1 no matter what types are inserted.

Do the same for a typed version of the term B of Lemma 2.11.

3.3 Find typed versions of the untyped terms S, A, M, E which 'make sense'.

What about the term B (of Example 1.13 and later) and the barmy term $\overline{d}$?

3.2 The typing discipline

The untyped calculus is too undisciplined in the sense that each term which is trying to describe the behaviour of a function completely ignores the source and target of that function. The raw terms of the simply typed calculus do contain some typing information, but this seems to achieved nothing (except make the syntax more cluttered). Clearly there is still something missing. What is it?

Remember that when we have a function

$$f : \mathbb{S} \longrightarrow \mathbb{T}$$

before we can apply f to a potential input s we *must* check that s is in $\mathbb{S}$. When this is the case we can form fs to obtain the output in $\mathbb{T}$. This typing aspect is missing from

the raw terms. Given any pair q, p of raw terms the compound (qp) is allowed as a raw term. This is too liberal, and we must somehow restrict the use of application so that q and p have a certain compatibility.

The kind of restriction we need is fairly obvious. However, once we try to organize it we find that it can get a bit messy. We need to take a step or two back, and introduce a few more syntactic notions. So here we go.

3.7 DEFINITION. A statement is a pair

$$t : \tau$$

where t, the subject, is a raw term and τ, the predicate, is a type.

A declaration is a statement

$$x : \sigma$$

where the term is an identifier.

A context is a list

$$\Gamma = x_l : \sigma_l, \ldots, x_1 : \sigma_1$$

of declarations. The context is legal if the declared identifiers are distinct.

A judgement

$$\Gamma \vdash t : \tau$$

is a statement in context.

Given such a judgement we say t inhabits τ (within the context Γ) or τ is the housing type of t (within the context Γ). ■

A judgement is a piece of syntax with, as yet, no meaning. The symbols '$\vdash$' and ':' are merely punctuation devices to organize the various parts. They are chosen because the whole judgement looks a bit like some other notation you know very well, and this is more than just a superficial resemblance.

We wish to read the displayed judgement as

> Within the *legal* context Γ,
> the raw term t is *well-formed*,
> to inhabit the *acceptable* type τ

but to do that it must conform to certain conditions. The legality of a context is explained above, the declared identifiers are distinct. For us every type is acceptable. (In more sophisticated systems the situation is more complex.) The crunch comes when we try to extract those raw terms that are well-formed.

We need some way of certifying that a judgement is correct (that is, can be read in the manner above). To do that we show it can be generated in certain way using a derivation system. This produces a more sophisticated version of a parsing tree.

3.8 DEFINITION. A derivation is a finite rooted tree of judgements grown according to the rules of Table 3.2 and subject to two provisos.

At a Projection the context Γ must be legal and the extracted statement $x : \sigma$ must be a component of Γ.

At a Weakening the inserted identifier x must be fresh, that is it must not occur in Γ. (There are no restrictions on the housing type σ.) ■

Rule	Shape
Projection	$\Gamma \vdash x : \sigma$
Weakening	$\dfrac{\Gamma \vdash t : \tau}{\Gamma, x : \sigma \vdash t : \tau}$
Introduction	$\dfrac{\Gamma, x : \sigma \vdash r : \rho}{\Gamma \vdash (\lambda x : \sigma \,.\, r) : (\sigma \to \rho)}$
Elimination	$\dfrac{\Gamma \vdash q : \pi \to \tau \quad \Gamma \vdash p : \pi}{\Gamma \vdash qp : \tau}$

Table 3.2: The construction rules for derivations

Quite soon we will look at some examples of derivations. You might want to look at those first or you might want to read the following 'explanation' first. Whatever you do we suggest you read these parts several times to pick up all the subtleties you might miss first time round.

What is a derivation trying to do? Consider a judgement

$$\Gamma \vdash t : \tau \quad \text{where} \quad \Gamma = x_l : \sigma_l, \ldots, x_1 : \sigma_1$$

as above. This is trying to describe an l-placed function with 'source'

$$\sigma_l \times \cdots \times \sigma_1$$

and 'target' τ and where

$$\begin{array}{c} \sigma_l \times \cdots \times \sigma_1 \longrightarrow \tau \\ (x_l, \ldots, x_1) \longmapsto t \end{array}$$

is the 'behaviour'. (We can make this precise and take the quotes off 'source', 'target', and 'behaviour' by developing a semantics for the calculus. However, that is not for these notes.) For this idea the required legality of the context is quite natural. We want to use different place-holders for the different input positions.

Now take another look at Definition 3.8 and remember this informal idea.

Each derivation is either a Leaf or is compound and built from smaller derivations in one of three ways.

Each Leaf is a Projection of the form

$$\Gamma \vdash x : \sigma$$

where Γ is a legal context and $x : \sigma$ is one of the declarations in Γ. This judgement is describing the obvious projection function, and the correctness of the judgement needs no more than a one line derivation.

The Weakening rule modifies a function by inserting an extra, dummy, input. Naturally, when we do this the inserted place-holder should be fresh.

The Introduction rule is the syntactic analogue of currying. Rather than view one component of the context as an input we move it across to form part of a more complex target.

The Elimination rule is the crucial trick. This is the symbolic analogue of applying a function q to an input p. To do that we *must* know that the type π of p matches the input position of q.

Let's look at some examples of derivations to see how this typing discipline works. For this and many later calculations we introduce a bit of notation.

For each type ζ we let

$$\zeta' = (\zeta \to \zeta)$$

to obtain a convenient abbreviation. This construction can be iterated. Thus

$$\begin{aligned}
\zeta' &= (\zeta \to \zeta) \\
\zeta'' &= (\zeta' \to \zeta') = ((\zeta \to \zeta) \to (\zeta \to \zeta)) \\
\zeta''' &= (\zeta'' \to \zeta'') = ((\zeta' \to \zeta') \to (\zeta' \to \zeta')) = \cdots \\
&\vdots
\end{aligned}$$

where the higher iterates begin to show why the abbreviation is helpful.

3.9 EXAMPLES. Consider the raw terms

$$\begin{aligned}
\overline{0}_\bullet &= \lambda y : \eta, x : \zeta \,.\, x \\
\overline{1}_\bullet &= \lambda y : \eta, x : \zeta \,.\, yx \\
\overline{2}_\bullet &= \lambda y : \eta, x : \zeta \,.\, y^2 x \\
\overline{3}_\bullet &= \lambda y : \eta, x : \zeta \,.\, y^3 x
\end{aligned}$$

of Table 3.1. We use derivations to extract some restriction on the embedded types ζ, η. To do that we use the context

$$\Gamma = y : \eta, x : \zeta$$

which we assume is legal (that is the two identifiers x and y are different).

(0) The derivation

$$\dfrac{\dfrac{\Gamma \vdash x : \zeta}{y : \eta \vdash (\lambda x : \zeta \,.\, x) : \zeta'}}{\vdash \overline{0}_\bullet : \eta \to \zeta'}$$

is formed from a Projection followed by two Introductions. This imposes no restrictions on η and ζ.

(1) We wish to use an Elimination

$$\dfrac{\Gamma \vdash y : \eta \quad \Gamma \vdash x : \zeta}{\Gamma \vdash yx : ?}$$

but to do that we must have

$$\eta = (\zeta \to \chi)$$

for some type χ. When this is the case

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash y : \eta \quad \Gamma \vdash x : \zeta}{\Gamma \vdash yx : \chi}}{y : \eta \vdash (\lambda x : \zeta \,.\, yx) : \eta}}{\vdash \overline{1}_\bullet : \eta'}$$

is a derivation.

(2) With $\eta = (\zeta \to \chi)$ we wish to use a derivation

$$\dfrac{\dfrac{}{\Gamma \vdash y : \eta} \quad \dfrac{\Gamma \vdash y : \eta \quad \Gamma \vdash x : \zeta}{\Gamma \vdash yx : \chi}}{\Gamma \vdash y^2 x : ?}$$

but for the second Elimination we must have

$$\eta = (\zeta \to \chi) = (\chi \to \xi)$$

for some type ξ. In other words

$$\xi = \chi = \zeta$$

is required, and hence $\eta = \zeta'$. When this is the case

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash y : \zeta'} \quad \dfrac{\Gamma \vdash y : \zeta' \quad \Gamma \vdash x : \zeta}{\Gamma \vdash yx : \zeta}}{\Gamma \vdash y^2 x : \zeta}}{y : \zeta' \vdash (\lambda x : \zeta \,.\, y^2 x) : \zeta'}}{\vdash \overline{2}_\bullet : \zeta''}$$

is a derivation.

(3) With

$$\overline{3_\zeta} = \lambda y : \zeta', x : \zeta \,.\, y^3 x$$

we have

$$\vdash \overline{3_\zeta} : \zeta''$$

by a slight extension of part (2). ■

As in these examples, a context may be empty and then the term is a combinator (of the simply typed calculus).

It should be clear how this short list of examples can be extended, and what role these terms will play. We begin the formal development of these ideas in the next section.

For now let's look at some more examples of derivations. This time we use the raw terms in the second block of Table of Table 3.1.

3.10 EXAMPLES. Consider the raw terms

$$\begin{aligned} \mathsf{S}_\zeta &= \lambda u : \zeta'', y : \zeta', x : \zeta \,.\, y(uyx) \\ \mathsf{A}_\zeta &= \lambda v, u : \zeta'', y : \zeta', x : \zeta \,.\, (vy)(uyx) \\ \mathsf{M}_\zeta &= \lambda v, u : \zeta'', y : \zeta', x : \zeta \,.\, v(uy)x \\ \mathsf{E}_{\text{Jones}} &= \lambda v, u : \zeta'', y : \zeta', x : \zeta \,.\, vuyx \end{aligned}$$

which are variants of those in Table 3.1. Are these well formed?

To deal with S_ζ let

$$\Gamma = u : \zeta'', y : \zeta', x : \zeta$$

to produce a legal context (provided x, y, u are different). Then

$$\dfrac{\dfrac{\Gamma \vdash u : \zeta'' \quad \Gamma \vdash y : \zeta'}{\Gamma \vdash uy : \zeta'} \qquad \Gamma \vdash x : \zeta}{\dfrac{\Gamma \vdash y : \zeta' \qquad \Gamma \vdash uyx : \zeta}{\Gamma \vdash y(uyx) : \zeta}}$$

followed by three Introductions gives

$$\vdash \mathsf{S}_\zeta : (\zeta'' \to \zeta'')$$

as required.

To deal with $\mathsf{A}_\zeta, \mathsf{M}_\zeta$ let

$$\Gamma = v : \zeta'', u : \zeta'', y : \zeta', x : \zeta$$

to produce a legal context (provided x, y, u, v are different). Then

$$\dfrac{\dfrac{\Gamma \vdash v : \zeta'' \quad \Gamma \vdash y : \zeta'}{\Gamma \vdash vy : \zeta'} \qquad \dfrac{\Gamma \vdash u : \zeta'' \quad \Gamma \vdash y : \zeta'}{\Gamma \vdash uy : \zeta'} \qquad \Gamma \vdash x : \zeta}{\Gamma \vdash (vy)(uyx)} \qquad \dfrac{\Gamma \vdash v : \zeta'' \qquad \dfrac{\Gamma \vdash u : \zeta'' \quad \Gamma \vdash y : \zeta'}{\Gamma \vdash uy : \zeta'}}{\dfrac{\Gamma \vdash v(yu) : \zeta' \qquad \Gamma \vdash x : \zeta}{\Gamma \vdash v(yu)x : \zeta}}$$

each followed by four Introductions give

$$\vdash \mathsf{A}_\zeta : \zeta'' \to \zeta'' \to \zeta'' \qquad \vdash \mathsf{M}_\zeta : \zeta'' \to \zeta'' \to \zeta''$$

respectively. (At this point you can see that displaying in full anything but the smallest derivation is impossible.) ■

These calculations suggest that E_J can be dealt with in the same way. However, it can't. To form vu within the body the identifiers v and u can not have the same type. We are beginning to see how the typing discipline works.

Exercises

3.4 (a) Show that if a judgement

$$\Gamma \vdash t : \tau$$

is derivable then the context Γ is legal.

(b) Show that if a judgement

$$\Gamma \vdash t : \tau$$

is derivable then $\partial t \subseteq \partial\Gamma$ where $\partial\Gamma$ is the set of identifiers declared in Γ.

(c) Show that if two judgements

$$\Gamma \vdash t : \tau_1 \qquad \Gamma \vdash t : \tau_2$$

are derivable then $\tau_1 = \tau_2$.

3.5 The term E_J of Examples 3.10 appears to be a typed version of the untyped term E of Examples 2.4. However, E_J is not well-formed. Can you find a well-formed typed term whose type erasure is E?

3.6 Show that the Turing fixed point combinator is not the type erasure of any well-formed typed combinator.

3.7 The formal composition of two well-formed terms

$$\Gamma \vdash g : (\theta \to \zeta) \qquad \Gamma \vdash f : (\eta \to \theta)$$

is the term

$$g \circ f = \lambda x : \eta \,.\, g(fx)$$

where x is fresh form Γ. Show that

$$\Gamma \vdash g \circ f : (\eta \to \zeta)$$

is derivable.

3.3 Simulation of arithmetic

Just as with the untyped calculus we are going to use the simply typed calculus to do some arithmetic. To begin we need the analogue of the untyped church numerals. Some of these typed version were used in Examples 3.9.

3.11 DEFINITION. For each $m \in \mathbb{N}$ and type ζ we set

$$\overline{m_\zeta} = \lambda y : \zeta', x : \zeta \,.\, y^m x$$

to obtain the typed church numeral for m over ζ. ■

The type erased version of $\overline{m_\zeta}$ is just $\overline{m}$. Put another way, each untyped numeral ramifies into a whole hierarchy of typed numerals. The consequences of this will soon become clear.

The calculations of Examples 3.9 indicate how the following is proved.

3.12 LEMMA. *For each $m \in \mathbb{N}$ and type ζ we have* $\vdash \overline{m_\zeta} : \zeta''$

This with Examples 3.10 and typed versions of the reductions in the proof of Lemma 2.4) more or less proves the following.

3.13 LEMMA. *For an arbitrary type ζ consider the terms $\mathsf{S}_\zeta, \mathsf{A}_\zeta, \mathsf{M}_\zeta$ of Examples 3.10. For each $m, n \in \mathbb{N}$ we have*

$$\vdash \mathsf{S}_\zeta\, \overline{m_\zeta} : \zeta'' \qquad \vdash \mathsf{A}_\zeta\, \overline{n_\zeta}\, \overline{m_\zeta} : \zeta'' \qquad \vdash \mathsf{M}_\zeta\, \overline{n_\zeta}\, \overline{m_\zeta} : \zeta''$$

and the reductions

$$\mathsf{S}_\zeta\, \overline{m_\zeta} \rhd\!\rhd \overline{(m+1)_\zeta} \qquad \mathsf{A}_\zeta\, \overline{n_\zeta}\, \overline{m_\zeta} \rhd\!\rhd \overline{(m+n)_\zeta} \qquad \mathsf{M}_\zeta\, \overline{n_\zeta}\, \overline{m_\zeta} \rhd\!\rhd \overline{(m \times n)_\zeta}$$

hold.

Notice that this result does not include the term E. That is because things are not as simple as you might think. The following definition should be compared with Definition 2.19, and you should wonder why it has been made so complicated.

3.14 DEFINITION. Let f be an l-placed function on $\mathbb{N}$ and let $\overline{f}$ be a raw term with

$$\vdash \overline{f} : \zeta(l)'' \to \cdots \to \zeta(1)'' \to \zeta(0)''$$

for some types $\zeta(0), \ldots, \zeta(l)$. We say $\overline{f}$ simulates f (in the simply typed λ-calculus) if the reduction

$$\overline{f}\,\overline{m(l)_{\zeta(l)}} \cdots \overline{m(1)_{\zeta(1)}} \rhd\!\!\rhd\!\!\rhd \overline{m(0)_{\zeta(0)}}$$

holds for all $m(0), m(1), \ldots, m(l) \in \mathbb{N}$ with $f(m(l), \ldots, m(1)) = m(0)$. ■

The underlying idea here is the same as that using untyped numerals, but notice that the types controlling the inputs and the output need not be the same. Lemma 3.13 shows that, as with the untyped calculus, the terms $\mathsf{S}_\zeta, \mathsf{A}_\zeta, \mathsf{M}_\zeta$ simulate successor, addition, and multiplication. However, these cases are not typical since the controlling types are all the same.

Exponentiation illustrates why different types for different inputs can be useful.

3.15 LEMMA. *For each type ζ the term*

$$\mathsf{E}_\zeta = \lambda v : \zeta''', u : \zeta'', y : \zeta'.x : \zeta \,.\, vuyx$$

satisfies

$$\vdash \mathsf{E}_\zeta : \zeta''' \to \zeta'' \to \zeta'' \qquad \vdash \mathsf{E}_\zeta : \zeta^{(iv)}$$

and simulates exponentiation, that is

$$\mathsf{E}_\zeta \overline{n_{\zeta'}}\,\overline{m_\zeta} \rhd\!\!\rhd\!\!\rhd \overline{m_\zeta^n}$$

for each $m, n \in \mathbb{N}$.

Proof. Using the legal context

$$\Gamma = v : \zeta''', u : \zeta'', y : \zeta'.x : \zeta$$

we have

$$\frac{\Gamma \vdash v : \zeta''' \quad \Gamma \vdash u : \zeta''}{\Gamma \vdash vu : \zeta''}$$

and then the remainder of the required derivation is straight forward. The reduction calculations are essentially the same as for the proof of Lemma 2.4). ■

This simulation uses two different input types, namely ζ and ζ'. You might wonder if there is a 'better' simulation of exponentiation for which only one input type is needed. There isn't, but we're not going to tell you why.

As you can see the development so far follows that of untyped simulation, except now we have to take some care to get the typing correct. However, notice that once the typing has been done it can be forgotten. The reductions are exactly the same as with the type erased versions.

Let's continue the parallel development and look the predecessor function d where

$$d0 = 0 \qquad dx' = x$$

for $x \in \mathbb{N}$. In Example 2.5 and Lemma 2.6 we cobbled together an untyped simulation of d, and Exercises 2.7 – 2.10 give an explanation of that construction. There is a typed version, and for that we need some pairing gadgets. The proof of the following is left as an exercise.

3.16 LEMMA. *Consider the terms*

$$\mathsf{P}_\zeta = \lambda l, r, z : \zeta'', y : \zeta', x : \zeta \,.\, z(\lambda u : \zeta \,.\, ryx)(lyx)$$

$$\mathsf{L}_\zeta = \lambda w : \zeta''' \,.\, w\overline{0_\zeta} \qquad \mathsf{R}_\zeta = \lambda w : \zeta''' \,.\, w\overline{1_\zeta}$$

for an arbitrary type ζ. Then

$$\vdash \mathsf{P}_\zeta : \zeta'' \to \zeta'' \to \zeta''' \qquad \vdash \mathsf{L}_\zeta : \zeta''' \to \zeta'' \qquad \vdash \mathsf{R}_\zeta : \zeta''' \to \zeta''$$

and the terms act as numeral-wise pairing gadgets, that is

$$\mathsf{L}_\zeta(\mathsf{P}_\zeta\,\overline{m_\zeta}\,\overline{n_\zeta}) \rhd\!\rhd \overline{m_\zeta} \qquad \mathsf{R}_\zeta(\mathsf{P}_\zeta\,\overline{m_\zeta}\,\overline{n_\zeta}) \rhd\!\rhd \overline{n_\zeta}$$

for all $m, n \in \mathbb{N}$.

For convenience for each $m, n \in \mathbb{N}$ let

$$(m, n)_\zeta = \mathsf{P}_\zeta\,\overline{m_\zeta}\,\overline{n_\zeta}$$

to produce a term $\vdash (m, n)_\zeta : \zeta'''$ such that

$$\mathsf{L}_\zeta(m, n)_\zeta \rhd\!\rhd \overline{m_\zeta} \qquad \mathsf{R}_\zeta(m, n)_\zeta \rhd\!\rhd \overline{n_\zeta}$$

hold. With these we have the following.

3.17 LEMMA. *For an arbitrary type ζ let*

$$\mathsf{D}_\zeta = \lambda w : \zeta''' \,.\, \mathsf{P}_\zeta(\mathsf{S}_\zeta(\mathsf{L}_\zeta w))(\mathsf{L}_\zeta w) \qquad \mathsf{d}_\zeta = \lambda x : \zeta^{(v)} \,.\, \mathsf{R}_\zeta(x\mathsf{D}_\zeta(0, 0)_\zeta)$$

where $\mathsf{P}_\zeta, \mathsf{L}_\zeta, \mathsf{R}_\zeta$ are the pairing gadgets of Lemma 3.16 and S_ζ simulates the successor function. Then we have

$$\vdash \mathsf{D}_\zeta : \zeta^{(iv)} \qquad \vdash \mathsf{d}_\zeta : \zeta^{(v)} \to \zeta''$$

and

$$\mathsf{D}_\zeta(m, n)_\zeta \rhd\!\rhd (m', m)_\zeta \qquad \mathsf{D}_\zeta^m(0, 0)_\zeta \rhd\!\rhd (m, dm)_\zeta \qquad \mathsf{d}_\zeta\overline{m_{\zeta'''}} \rhd\!\rhd \overline{(dm)_\zeta}$$

for all $m, n \in \mathbb{N}$.

Proof. Most of this is straight forward or a re-working of earlier results and exercises. For the final reduction we have

$$\mathsf{d}_\zeta\overline{m_{\zeta'''}} \rhd\!\rhd \mathsf{R}_\zeta(\overline{m_{\zeta'''}}\mathsf{D}_\zeta(0, 0)_\zeta) \rhd\!\rhd \mathsf{R}_\zeta(\mathsf{D}_\zeta^m(0, 0)_\zeta) \rhd\!\rhd \mathsf{R}_\zeta(m, dm)_\zeta \rhd\!\rhd \overline{(dm)_\zeta}$$

as required. ■

What other functions can be simulated? The situation for the typed calculus is not quite as straight forward as you might think.

Let us look at the iteration of the predecessor function d. Thus

$$d^j x = \begin{cases} x - j & \text{if } j \leq x \\ 0 & \text{if } x \leq j \end{cases}$$

for all $x, j \in \mathbb{N}$. When working on $\mathbb{N}$ this substatction is often written $x - j$ with the tacit convention that it takes the value 0 whene $x < j$.

Since d is simulated in the calculus (by d_ζ) and since numerals enable us to capture iteration, it seems 'obvious' that this 2-placed substraction function can be simulated, but there is a problem. The simulating term has type

$$\mathsf{d}_\zeta : \zeta^{(3)\prime\prime} \to \zeta^{\prime\prime}$$

and such a term can *not* be formallly iterated (since its source and target types are different). However, all is not lost.

3.18 EXAMPLE. Let ζ be some fixed type and generate the types $\zeta^{(i)}$ by

$$\zeta^{(0)} = \zeta \qquad \zeta^{(i+1)} = \zeta^{(i)\prime}$$

(for $i < \omega$). Thus $\zeta^{(0)} = \zeta, \zeta^{(1)} = \zeta', \zeta^{(2)} = \zeta'', \zeta^{(3)} = \zeta''', \ldots$ and so on. By Lemma 3.17 we have a term

$$\vdash \mathsf{d}_i : \zeta^{(i+3)\prime\prime} \to \zeta^{(i)\prime\prime}$$

such that

$$\mathsf{d}_i \overline{m_{i+3}} \rhd\!\!\rhd \overline{(m-1)_i}$$

for each $m \in \mathbb{N}$. Throughout the example we write $\overline{m_i}$ for $\overline{m_{\zeta^{(i)}}}$

Using the terms d_i we generate a whole family of terms

$$\vdash \mathsf{d}_i^j : \zeta^{(i+3j)\prime\prime} \to \zeta^{(i)\prime\prime}$$

by recursion on j with variation of the parameter i. Thus

$$\mathsf{d}_i^0 = \lambda x : \zeta(i)^{\prime\prime} . x \qquad \mathsf{d}_i^{j+1} = \lambda x : \zeta^{(i+3+3j)\prime\prime} . \mathsf{d}_{i+3}(\mathsf{d}_{i+3}^j x)$$

for each $i < j, \omega$. It is routine to check that these are well-formed with the indicated types. Also, we may check that

$$\mathsf{d}_i^j \overline{m_{i+3j}} \rhd\!\!\rhd \overline{(m-j)_i}$$

for each $m \in \mathbb{N}$ and $i < \omega$. ■

This doesn't show that the 2-placed subtraction function can be simulated in the calculus, for we can not use the index j as an input. Let's leave that for a while and look at the trick used in the construction, namely composition. We begin with an utterly trivial observation.

3.19 EXAMPLE. Suppose $f = h \circ g$ where all three function f, g, h are 1-placed. Supose the terms

$$\vdash \overline{g} : \theta^{\prime\prime} \to \eta^{\prime\prime} \qquad \vdash \overline{h} : \eta^{\prime\prime} \to \zeta^{\prime\prime}$$

simulate g and h. Then

$$\overline{f} = \lambda x : \theta^{\prime\prime} . \overline{h}(\overline{g} x)$$

satifies

$$\vdash \overline{f} : \theta^{\prime\prime} \to \zeta^{\prime\prime}$$

and simulates f. ■

Suppose we have $f = h \circ g$, as in the Example, and suppose we wish to produce a simulation of f. Suppose also that, for some reason, we want the target type of the simulation to be ζ''. To produce this we first find a simulation of h with type $\eta'' \to \zeta''$ for some η. Notice that we may not have much control over the type η; we have to accept what comes out in the wash. Once we have found η we require a simulation of g with type $\theta'' \to \eta''$ for some type θ, and then we can produce the required simulation with type $\theta'' \to \zeta''$.

The point of these remarks is that there are times when we don't have much control over certain types. In particular, the notion of a simulation (as in Definition 3.14) is not quite what we want.

3.20 DEFINITION. Let f be an l-placed function on $\mathbb{N}$. Then f is simulated throughout the calculus if for each type $\zeta(0) = \zeta$ there are types $\zeta(1), \ldots, \zeta(l)$ and a term

$$\vdash \overline{f} : \zeta(l)'' \to \cdots \to \zeta(1)'' \to \zeta(0)''$$

which simulates f. ■

You may go back and check that all the examples of simulations given in this section show that the parent function is simulated throughout the calculus. This is becase in all cases the target type is completely arbitrary.

You may think this is being over fussy, but a slight extension of Example 3.19 we indicate that there may be a problem.

3.21 EXAMPLE. Suppose $f = h \circ (k, g)$ where the two functions g, k are 1-placed and h is 2-place. Thus f is the 1-placed function given by

$$fm = h(km, gm)$$

for each $m \in \mathbb{N}$. Suppose each of h, g, k is simulated throughout the calculus. Surely, under these circumstances f is simulated throughout the calculus.

Consider an arbitrary type ζ. There is some term

$$\vdash \overline{h} : \xi'' \to \eta'' \to \zeta''$$

which simulates h. Here we have to make do with the types ξ and η we are given. We wish to input values of k, g into h so we use simuations

$$\vdash \overline{k} : \chi'' \to \xi'' \qquad \vdash \overline{g} : \theta'' \to \eta''$$

for some types χ and θ which we have little control over.

How does this lead to a simulation of f? The temptation is to use a term

$$\lambda x : (?) \,.\, \overline{h}(\overline{k}x)(\overline{g}x)$$

for some type $(?)$. But this term is not well-formed unless, by chance, $\chi = \theta$. What if $\chi \neq \theta$?

Suppose for some type π there are terms

$$\vdash K : \pi'' \to \chi'' \qquad \vdash G : \pi'' \to \xi''$$

which convert numerals in the sense that

$$K\,\overline{m_\pi} \rhd\!\rhd \overline{m_\chi} \qquad G\,\overline{m_\pi} \rhd\!\rhd \overline{m_\theta}$$

for each $m \in \mathbb{N}$. Then

$$\overline{f} = \lambda x : \pi'' \,.\, \overline{h}(\overline{k}(Kx))(\overline{g}(Gx))$$

satisfies

$$\vdash \overline{f} : \pi'' \to \zeta''$$

and simulates f. ■

The existence or not of such numeral converters is far from clear. In fact, the calculus as set up in section 3.1 is flawed, and needs to be improved. This we do in the next section.

Exercises

3.8 Complete the proofs of Lemmas 3.16 and 3.17.

3.9 For an arbitrary type ζ let

$$\mathsf{B}_\zeta = \lambda u : \eta', v : \zeta'', w : \zeta''' \,.\, uVw$$

where

$$V = \lambda z : \zeta''', y : \zeta', x : \zeta \,.\, zvyx \qquad \eta = \zeta''' \to \zeta''$$

are the auxiliaries.

Show that B_ζ is well-formed and determine its type.

Does this term simulate the stacking function $\beth$?

3.10 For an arbitrary type ζ let

$$\mathsf{K}_\zeta = \lambda x, u : \zeta \,.\, x \qquad \mathsf{J}_\zeta = \lambda y, v : \zeta', u : \zeta \,.\, y(vu)$$

to obtain two standard combinators. Using these let

$$\mathsf{L}_\zeta = \lambda w : \zeta''', y : \zeta', x : \zeta \,.\, w(\mathsf{J}_\zeta y)(\mathsf{K}_\zeta x)x$$

to obtain a non-standard combinator. Show that

$$\vdash \mathsf{L}_\zeta : \zeta''' \to \zeta''$$

and that this term converts numerals.

3.11 Show that for each $l \in \mathbb{N}$ the 2-placed function $\beth(l, \cdot, \cdot)$ is simulated throughout the calculus.

3.4 Product types and pairing gadgets

On several ocasssions we have seen how pairing gadgets are used in the simulation of functions. So far these have been obained by various coding tricks. However, when we look at the real world calculations that underlie the simulations we see that the manipulation of pairs and tuples of numbers is the crucial idea. Something as fundamental as this should not be obtained by a trick; it should be an explicit part of the calculus. In this section we show how this is done.

We re-work the material of Sections 3.1 and 3.2, but because of our familiarity with the ideas we can now proceed quite quickly.

We use a more general notion of type with explicit products.

3.22 DEFINITION. The types of the simply typed λ-calculus with products are generated from an unlimited stock of variables using three rules of construction (a base and two steps).

- Each variable is a type.
- If ρ, σ are types, then so is $(\sigma \to \rho)$.
- If ρ, σ are types, then so is $(\sigma \times \rho)$.

There are no other types. ■

This calculus has a construct which produces a new type

$$(\sigma \times \rho)$$

out of two previously constructed types σ, ρ. Here we have put in the punctuating brackets whereas we usually leave them out. The idea is that if σ and ρ are descriptions of two concrete sets then $\sigma \times \rho$ is a description of the product of these sets.

In line with this idea there should be syntactic analogues of the pairing gadgets. So far whenever we have needed these we have cobbled together three terms to do the job in hand. For this calculus we used constants. This is a new kind of primitive term.

3.23 DEFINITION. For the simply typed λ-calculus with products, for each ordered pair of types σ, ρ there are three associated constants

$$\mathsf{Pair}_{\sigma,\rho} : \sigma \to \rho \to \sigma \times \rho \qquad \mathsf{Left}_{\sigma,\rho} : \sigma \times \rho \to \sigma \qquad \mathsf{Right}_{\sigma,\rho} : \sigma \times \rho \to \rho$$

the pairing constants. ■

In this definition the three constants are

$$\mathsf{Pair}_{\sigma,\rho} \qquad \mathsf{Left}_{\sigma,\rho} \qquad \mathsf{Right}_{\sigma,\rho}$$

and these are primitive symbols which, as we explain shortly, may be used in the construction of terms. They are indexed by the pair of types, so there is a triple of constants for each *ordered* pair of types.

As well as giving the new symbols the definition gives some more information. For each constant k there is a nominated housing type κ, and then the statement

$$\mathsf{k} : \kappa$$

is an axiom (with a lower case initial letter). These are used to enrich the derivation system.

Rule	Shape
Axiom	$\Gamma \vdash \mathsf{k} : \kappa$
Projection	$\Gamma \vdash x : \sigma$
Weakening	$\dfrac{\Gamma \vdash t : \tau}{\Gamma, x : \sigma \vdash t : \tau}$
Introduction	$\dfrac{\Gamma, x : \sigma \vdash r : \rho}{\Gamma \vdash (\lambda x : \sigma \,.\, r) : (\sigma \to \rho)}$
Elimination	$\dfrac{\Gamma \vdash q : \pi \to \tau \quad \Gamma \vdash p : \pi}{\Gamma \vdash qp : \tau}$

Table 3.3: The construction rules for derivations

3.24 DEFINITION. The raw terms of the simply typed λ-calculus with products are generated from an unlimited stock of identifiers using four rules of construction (two base and two steps).

- Each identifier is a raw term.
- Each constant is a raw term.
- If q, p are raw terms, then the application (qp) is a raw term.
- If x is an identifier, σ is a type, and r is a raw term, then the abstraction $(\lambda x : \sigma \,.\, r)$ is a raw term.

There are no other terms. ■

Some examples will not go amiss. We will build on these later.

3.25 EXAMPLES. Let τ, σ, ρ be arbitrary types and let

$$\pi = ((\tau \times \sigma) \times \rho)$$

so we may condence some syntax. Let

$$\begin{aligned}
\mathsf{Trip}_{\tau,\sigma,\rho} &= \lambda z : \tau, y : \sigma, x; \rho \,.\, \mathsf{Pair}_{\tau\times\sigma,\rho}(\mathsf{Pair}_{\tau,\sigma} zy)x \\
\mathsf{West}_{\tau,\sigma,\rho} &= \lambda w : \pi \,.\, \mathsf{Left}_{\tau,\sigma}(\mathsf{Left}_{\tau\times\sigma,\rho} w) \\
\mathsf{Cent}_{\tau,\sigma,\rho} &= \lambda w : \pi \,.\, \mathsf{Right}_{\tau,\sigma}(\mathsf{Left}_{\tau\times\sigma,\rho} w) \\
\mathsf{East}_{\tau,\sigma,\rho} &= \mathsf{Right}_{\tau\times\sigma,\rho}
\end{aligned}$$

to obtain four raw terms. We will come back to these in later examples. ■

The constants $\mathsf{Pair}_\bullet, \mathsf{Left}_\bullet, \mathsf{Right}_\bullet$ require a modification of both the reduction mechanism and the typing discipline. It is probably more instructive to look at the derivation system first.

3.26 DEFINITION. A derivation (in the simply typed calculus with products) is a finite rooted tree of judgements grown according to the rules of Table 3.3 and subject to three provisos.

At an Axiom the context Γ must be legal and the statment $\mathsf{k} : \kappa$ must be an axiom (that is a constant with its housing type).

At a Projection the context Γ must be legal and the extracted statement $x : \sigma$ must be a component of Γ.

At a Weakening the inserted identifier x must be fresh, that is it must not occur in Γ. (There are no restrictions on the housing type σ.) ■

The new facilities here are the Axioms. These are new kinds of Leaves and are used to immerse the constants in terms.

3.27 EXAMPLES. Consider the raw terms

$$\mathsf{Trip}_{\tau,\sigma,\rho} \qquad \mathsf{West}_{\tau,\sigma,\rho} \qquad \mathsf{Cent}_{\tau,\sigma,\rho} \qquad \mathsf{East}_{\tau,\sigma,\rho}$$

of Example 3.25. We show these are well-formed combinators and exhibit the type of each. To do that we use the constants

$$\mathsf{Pair}_\bullet \qquad \mathsf{Left}_\bullet \qquad \mathsf{Right}_\bullet$$

appropriately indexed, and to save a bit of space we abbreviate these to

$$\mathsf{P}_\bullet \qquad \mathsf{L}_\bullet \qquad \mathsf{R}_\bullet$$

respectively.

Using the context

$$\Gamma = z : \tau, y : \sigma, x : \rho$$

the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathsf{P}_{\tau,\sigma} : \star \quad \Gamma \vdash z : \tau}{\Gamma \vdash \mathsf{P}_{\tau,\sigma} z : \sigma \to \tau \times \sigma} \quad \Gamma \vdash y : \sigma}{\Gamma \vdash \mathsf{P}_{\tau,\sigma} zy : \tau \times \sigma}}{\Gamma \vdash \mathsf{P}_{\tau\times\sigma,\rho} : \star \qquad \Gamma \vdash \mathsf{P}_{\tau\times\sigma,\rho}(\mathsf{P}_{\tau,\sigma} zy) : \rho \to \pi}}{\Gamma \vdash \mathsf{P}_{\tau\times\sigma,\rho}(\mathsf{P}_{\tau,\sigma} zy)x : \pi} \quad \Gamma \vdash x : \rho}{\dfrac{\bullet}{\bullet}}}{\vdash \mathsf{Trip}_{\tau,\sigma,\rho} : \tau \to \sigma \to \rho \to \pi}$$

deals with $\mathsf{Trip}_{\tau,\sigma,\rho}$. Here each of the two $\star$ is an appropriate type, and not the same one.

The derivation

$$\dfrac{\dfrac{w : \pi \vdash \mathsf{Right}_{\tau,\sigma} : \tau \times \sigma \to \sigma \qquad \dfrac{w : \pi \vdash \mathsf{L}_{\tau\times\sigma,\rho} : \pi \to \tau \times \sigma \quad w : \pi \vdash w : \pi}{w : \pi \vdash \mathsf{L}_{\tau\times\sigma,\rho} w : \tau \times \sigma}}{w : \pi \vdash \mathsf{Right}_{\tau,\sigma}(\mathsf{L}_{\tau\times\sigma,\rho} w) : \sigma}}{\vdash \mathsf{Cent}_{\tau,\sigma,\rho} : \pi \to \sigma}$$

deals with $\mathsf{Cent}_{\tau,\sigma,\rho}$.

The other two terms $\mathsf{West}_{\tau,\sigma,\rho}, \mathsf{East}_{\tau,\sigma,\rho}$ are dealt with in a similar way. ■

The reduction mechanism of this calculus is more sophisticated than before. We still have a rule

$$(\lambda x : \sigma \,.\, r)s \rhd r[x := s]$$

as before, but now there are also some rules involving the pairing gadgets.

3.28 DEFINITION. An abstraction redex is a term

$$(\lambda x : \sigma \,.\, r)s$$

and then

$$r[x := s]$$

is its immediate reduct.

A pairing redex is a term of one of the two forms

$$\mathsf{Left}_\bullet(\mathsf{Pair}_\bullet lr) \qquad \mathsf{Right}_\bullet(\mathsf{Pair}_\bullet, lr)$$

where l and r are terms. The imediate reduct if this term is

$$l \qquad r$$

respectively.

A 1-step reduction

$$t^- \rhd t^+$$

converts a redex t^- (of some kind) into its immediate redex t^+. ■

The reduction relation $t^- \rhd\!\!\rhd t^+$ on terms is now generated in the same way as before. Thus we move from t^- to t^+ using a sequence of replacements. At each step we replace either an abstraction redex or a pairing redex by the corresponding reduct. We need not give a formal definition of this here (for we are going to look at a more general idea in the next chapter), but some examples will help.

3.29 EXAMPLES. Consider the set up of Examples 3.25 and 3.27. We show that

$$\mathsf{West}_\bullet(\mathsf{Trip}_\bullet\, w\, c\, e) \rhd\!\!\rhd w \qquad \mathsf{Cent}_\bullet(\mathsf{Trip}_\bullet\, w\, c\, e) \rhd\!\!\rhd c \qquad \mathsf{East}_\bullet(\mathsf{Trip}_\bullet\, w\, c\, e) \rhd\!\!\rhd e$$

for all terms w, c, e.

For the west case observe that

$$\mathsf{Trip}_\bullet\, w\, c\, e \rhd\!\!\rhd \mathsf{Pair}_\bullet(\mathsf{Pair}_\bullet\, w\, c)e$$

by the removal of three abstraction redexes. With this we have

$$\mathsf{W}_\bullet(\mathsf{T}_\bullet\, w\, c\, e) \rhd\!\!\rhd \mathsf{W}_\bullet(\mathsf{P}_\bullet(\mathsf{P}_\bullet\, w\, c)e) \rhd\!\!\rhd \mathsf{L}_\bullet(\mathsf{L}_\bullet(\mathsf{P}_\bullet(\mathsf{P}_\bullet\, w\, c)e)) \rhd\!\!\rhd \mathsf{L}_\bullet(\mathsf{P}_\bullet\, w\, c) \rhd\!\!\rhd w$$

where we have abbreviated each term by its initial latter. Each of the last two steps use the removal of a pairing gadget.

The central case is similar, and the east case is immediate. ■

The use of explicit pairing constants makes the simply typed calculus far more flexible, but it still has some flaws. For instance, we still can not show that the class of functions simulated throught the calculus is closed under composition. This is because of a rather silly problem (that not all types are inhabited). We won't attempt to solve this problem here, for there is a better, more honest, way of doing things.

Exercises

3.12 In this exercise we write $\mathsf{P}, \mathsf{L}, \mathsf{R}$ for the pairing gadgets and omit the typing subscripts σ, ρ. Notice that in general

$$\mathsf{P}(\mathsf{L}w)(\mathsf{R}w) \rhd\!\rhd w$$

need not hold. This exercise indicates how this can be partially rectified.

Let $\theta = \sigma \times \rho$, so that

$$\mathsf{P} : \sigma \to \rho \to \theta \qquad \mathsf{L} : \theta \to \sigma \qquad \mathsf{R} : \theta \to \rho$$

are the typing restrictions. Let

$$\mathsf{Id} = \lambda w : \theta \,.\, \mathsf{P}(\mathsf{L}w)(\mathsf{R}w)$$

to obtain a well formed term $\vdash \mathsf{Id} : \theta'$.

Show how $\mathsf{P}, \mathsf{L}, \mathsf{R}$ may be modified to terms $\tilde{\mathsf{P}}, \tilde{\mathsf{L}}, \tilde{\mathsf{R}}$ which do form pairing gadets for σ, ρ and for which

$$\tilde{\mathsf{L}} \circ \mathsf{Id} \rhd\!\rhd \tilde{\mathsf{L}} \qquad \tilde{\mathsf{R}} \circ \mathsf{Id} \rhd\!\rhd \tilde{\mathsf{R}} \qquad \tilde{\mathsf{P}} \circ (\tilde{\mathsf{L}}, \tilde{\mathsf{R}}) \rhd\!\rhd \mathsf{Id}$$

where the last reduction uses the obvious formal composition.

3.5 Final remarks

In this final section we consider some of the questions which ought to be asked about the simply typed calculus (with products). Although we will give the answers to these questions, we do not give proofs. Most of the proofs are rather complicated and are rather unenlightening when first seen.

The questions are essentially the same as those asked about the untyped calculus but now there are nicer answers.

3.30 PROPOSITION. (Confluence) *The simply typed calculus (with products) is confluent. That is, for each divergent reduction wedge*

$$t^- \rhd\!\rhd\!\rhd l \qquad t^- \rhd\!\rhd\!\rhd r$$

from a common source there is a convergent reduction wedge

$$l \rhd\!\rhd\!\rhd t^+ \qquad r \rhd\!\rhd\!\rhd t^+$$

to a common sink.

In this respect this typed calculus is similar to the untyped calculus. However, when it comes to normalization the typed calculus is very different.

3.31 PROPOSITION. (Strong normalization) *Suppose the judgement*

$$\Gamma \vdash t^- : \tau$$

is derivable. Then there is a normal term t^- where $t^- \rhd\!\rhd\!\rhd t^+$ and

$$\Gamma \vdash t^+ : \tau$$

is derivable. Furthermore, this term t^+ is unique (up to α-equivalence) and every reduction path from t^- eventually leads to t^+.

This with the next result show that the typing discipline gives some very strong safeguards.

3.32 PROPOSITION. (Subject reduction) *Suppose the judgement*

$$\Gamma \vdash t^- : \tau$$

is derivable, and suppose $t^- \rhd\!\rhd t^+$. *Then*

$$\Gamma \vdash t^+ : \tau$$

is derivable.

This shows that once we have verified that a term is well-formed, we can mess about with it as much as we like. In fact, a stronger result holds. Once we have verified that a term is well-formed, we can erase all types and work in the untyped calculus. Every reduction we perform will lead to a term that can be typed and, in fact, we can use the original derivation and the reduction to reinstate the types.

These three results show that the typed calculus (with products) is very much nicer than the untyped calculus. However, there is a price to pay.

Let us say a 1-placed function f outstrips a multiplaced function g if eventually for comparable inputs the value of f is greater then the value of g. This intuitive idea can be made precise (in terms of eventual domination), but we need not worry about the precise details here.

By Exercise 3.11 we see that stacked exponentiation $\beth(0,\cdot,\cdot)$, $\beth(1,\cdot,\cdot)$, $\beth(2,\cdot,\cdot)$, ... to any fixed stacking level can be simulated.

3.33 PROPOSITION. *Each function that can be simulated in the simply typed calculus (with products) is outstripped by a function* $\beth(l,2,\cdot)$ *for some* l.

In particularly the 3-placed function function $\beth$ can not be simulated. This is a very severe restriction on the class of simulated functions. The class is a very small part of the class of primitive recursive functions which itself is a very small class of the Turing computable functions, and each one of these can be simulated in the untyped calculus.

In short, by imposing a typing discipline and moving from untyped terms to typed terms, we have very severely weakened the system. Is there any way we can retrieved at least some of this strength? We see how to do this in the next chapter.

Chapter 4
An applied calculus, λG

We have seen the two extremes. The untyped calculus is capable of handling all Turing computable functions, whereas the simply typed calculus can handle far fewer functions. It would be nice if we can find ways of handling classes of functions somewhere in between.

The way to do this is to recognize that so far functions have been handled in a not too honest fashion. We have got at the natural numbers via the church numerals, and in the final reckoning this is nothing more than a trick. We use it for no other reason than it works. Perhaps there are other kinds of numerals which, in some ways, are better.

In this chapter we set up a calculus which is specifically designed to handle natural number gadgets. We modify the simply typed calculus in two ways. Firstly, we drop the use of variables as undetermined types, and introduce an atom $\mathcal{N}$ which is used as the name of $\mathbb{N}$. All other types are built out of this. Secondly, we use constants to name specific numeric gadgets. The names of all other functions are built out of these.

4.1 The basic facilities

In this section we set up the basic facilities of the calculus; the types, terms, derivation system, and the reduction mechanism. Because of our familiarity with several other calculi we can get through this basic material quite quickly.

4.1 DEFINITION. The types of the calculus are generated from a single atom $\mathcal{N}$.

- $\mathcal{N}$ is a type.
- If ρ, σ are types, then so is $(\sigma \to \rho)$.
- If ρ, σ are types, then so is $(\sigma \times \rho)$.

There are no other types. In particular, there are no variables. ■

The new feature here is that there are no variables and all the types are generated from a single atom. Thus the first few types are

$$\begin{array}{c} \mathcal{N} \\ \mathcal{N}' \quad \mathcal{N} \times \mathcal{N} \\ \mathcal{N}'' \quad \mathcal{N}' \times \mathcal{N} \quad \mathcal{N} \times \mathcal{N}' \quad \mathcal{N}' \times \mathcal{N}' \quad (\mathcal{N} \times \mathcal{N})' \\ \mathcal{N}' \to \mathcal{N} \quad \mathcal{N} \to \mathcal{N}' \quad \mathcal{N} \times \mathcal{N} \to \mathcal{N} \quad \mathcal{N} \to \mathcal{N} \times \mathcal{N} \quad \mathcal{N} \times \mathcal{N} \to \mathcal{N} \times \mathcal{N} \\ \vdots \qquad \vdots \end{array}$$

and so on. We continue with the abbreviation conventions of the earlier chapters. We let

$$\tau' \quad \text{abbreviate} \quad (\tau \to \tau)$$

and iterate to obtain $\tau, \tau', \tau'', \tau'''$, and so on. As before we let $\rho, \sigma, \tau \ldots$ range over types.

The idea is that $\mathcal{N}$ is the name of the set $\mathbb{N}$ of natural numbers, and each type is the name of the corresponding space generated from $\mathbb{N}$. The well-formed terms of the calculus (discussed shortly) name functions between these spaces. This can be made precise using the ***Set***-based semantics or, more generally, a categorical semantics.

The raw terms of the calculus are generated as in Definition 3.24 but with more constants.

4.2 DEFINITION. The constants of the calculus fall into three classes.

- For each types σ, ρ there are the three pairing constants

$$\mathsf{Pair}_{\sigma,\rho} : \sigma \to \rho \to \sigma \times \rho \qquad \mathsf{Left}_{\sigma,\rho} : \sigma \times \rho \to \sigma \qquad \mathsf{Right}_{\sigma,\rho} : \sigma \times \rho \to \rho$$

  with the usual properties.

- There are two constants

$$\mathsf{0} : \mathcal{N} \qquad \mathsf{S} : \mathcal{N}'$$

  which serve as the name of 0 and the successor operation.

- For a selected family Σ of types σ there is a constant

$$\mathsf{I}_\sigma : \mathcal{N} \to \sigma''$$

  called the iterator over σ.

There are no other constants. ■

The pairing constants are just as in Section 3.4 and do the same job. The two constants

$$\mathsf{0} : \mathcal{N} \qquad \mathsf{S} : \mathcal{N}'$$

are, respectively, the name of the zero natural number and the name of the successor function on natural numbers.

We will use each iterator

$$\mathsf{I}_\sigma : \mathcal{N} \to \sigma''$$

to obtain certain recursion. Exactly how this is done is the topic in the next three sections. Notice that we have some freedom over the choice of the family Σ. This will give us a way of controlling the strength of the calculus. When Σ is empty we get a very weak calculus. When Σ contains all types we get a very strong calculus (but nowhere near as strong as the untyped calculus). When $\Sigma = \{\mathcal{N}, \mathcal{N} \times \mathcal{N}\}$ we get something like primitive recursive arithmetic.

4.3 DEFINITION. The raw terms of the calculus are generated from an unlimited stock of identifiers and the constants of Definition 4.2 in the manner of Definition 3.24. ■

We carry over all the previous notions such as the support ∂t of a term t. We also use the notion of α-equivalent terms, that is terms that differ only in the choice of the bound identifiers. We will look at some particular examples of terms in the following sections.

As with the simply typed calculus we use a derivation system to extract the well-formed terms (in context).

4.4 DEFINITION. A derivation in λG is a finite rooted tree of judgements grown according to the rules of Table 3.3 with the usual provisos, as in Definition 3.26. ■

The difference here, of course, is that we now have many more Axioms, that is Leaves where the subject is a constant. Previously the only Axioms we have seen involve the pairing constants. We now have $\mathsf{0}, \mathsf{S}$ and the iterators I_σ. We will look at some examples of derivations involving these in the next section.

The most intricate part of this applied calculus is the reduction mechanism. We still have the removal of abstraction redexes (as in the simply typed calculus) and the removal of paring redexes (as in the calculus with products). We now also have another kind of redex removal involving the iterators. At first sight this aspect might seem a bit of a mystery, but all will be explained by a few examples. Let's get the basic definitions out of the way.

4.5 DEFINITION. For λG there are three kinds of 1-step reductions.

- For each redex $(\lambda x : \sigma \,.\, r)s$ there is a redex removal

$$(\lambda x : \sigma \,.\, r)s \rhd r[x := s]$$

  using the immediate reduct. More precisely, an abstraction redex is converted into its abstraction reduct.

- For terms l, r we have

$$\mathsf{Left}(\mathsf{Pair}lr) \rhd l \qquad \mathsf{Right}(\mathsf{Pair}lr) \rhd r$$

  (where the subscripts on the pairing constants have been omitted). That is a pairing redex is converted into its pairing reduct.

- For terms r, s, t we have

$$\mathsf{I}0ts \rhd s \quad \mathsf{I}(\mathsf{S}r)ts \rhd t(\mathsf{I}rts)$$

  (where the subscript on the iterator has been omitted). That is an iterator redex is converted into its iterator reduct.

There are no other 1-step reductions. ■

The iterator reductions are new. This is the important, almost characteristic, property of the calculus. Let's look at an informal explanation of this.

Consider an iterator I_σ over some type σ. Think of σ as the name of some concrete set $\mathbb{S}$. Consider some function $f : \mathbb{S} \longrightarrow \mathbb{S}$. Such a function can be iterated. Thus we set

$$f^0 = id_{\mathbb{S}} \quad f^1 = f \quad f^2 = f \circ f \quad f^3 = f \circ f^2 \quad \cdots \quad f^{r+1} = f \circ f^r \quad \cdots$$

to produce the family

$$(f^r \mid r \in \mathbb{N})$$

of finite iterates of f. Each one of these can be evaluated at an arbitrary $s \in \mathbb{S}$. Thus

$$f^0 s = s \quad f^{r+1} s = f(f^r s)$$

for each $r \in \mathbb{N}$. Now consider the high order function

$$\mathbb{I}_\sigma : \mathbb{N} \longrightarrow \mathbb{S}' \longrightarrow \mathbb{S} \longrightarrow \mathbb{S}$$

in curried form which first take a natural number $m \in \mathbb{N}$, then takes a function $f \in \mathbb{S}'$, and finally takes an argument $s \in \mathbb{S}$, to return the eventual value

$$\mathbb{I}_\sigma m f s = f^m s$$

the m^{th} iterate of f on s. This function $\mathbb{I}_\sigma$ can be specified by

$$\mathbb{I}_\sigma 0 f s = s \qquad \mathbb{I}_\sigma m' f s = f(\mathbb{I}_\sigma m f s)$$

for each $m \in \mathbb{N}, f \in \mathbb{S}', s \in \mathbb{S}$.

The idea is that I_σ is the canonical name in the calculus of $\mathbb{I}_\sigma$, and the two 1-step reductions are direct translations of the two components of the specification of $\mathbb{I}_\sigma$.

You can probably guess what the reduction relation $\rhd\!\!\rhd$ is, but let's formally define it.

4.6 DEFINITION. For terms t^- and t^+ we write

$$t^- \rhd\!\!\rhd t^+$$

and say t^- reduces to t^+ if we can move from t^- to t^+ in a finite number of steps where at each step we do one of three things.

- We replace an abstraction redex subterm of the current term by its abstraction reduct.
- We replace a pairing redex subterm of the current term by its pairing reduct.
- We replace an iterator redex subterm of the current term by its iterator reduct.

At each step we are allowed to take an α-variant of the resulting term.

A term is normal if no subterm is a redex of any kind.

Let $\rhd\!\!\rhd\!\!\rhd$ be the reflexive closure of $\rhd\!\!\rhd$. Thus for terms t^-, t^+ we have

$$t^- \rhd\!\!\rhd\!\!\rhd t^+$$

when either t^- and t^+ are the same term (up to α-equivalence) or $t^- \rhd\!\!\rhd t^+$. ■

This is the same idea as before but the existence and removal of iterator redexes bring in some quite new properties. As before we have introduced the reflexive version $\rhd\!\!\rhd\!\!\rhd$ of the reduction relation, but now we will really use it. It is this reduction mechanism which makes the calculus what it is, so we should spend some time looking at some examples.

Exercises

4.1 *Need some*

4.2 Capturing numeric gadgets

The constants $\mathsf{0}, \mathsf{S}$ are used to produce the canonical name for each natural number.

4.7 DEFINITION. For each $m \in \mathbb{N}$ we set

$$\ulcorner m \urcorner = \mathsf{S}^m\mathsf{0}$$

to obtain the numeral for m. ■

In other words we have

$$\ulcorner 0 \urcorner = \mathsf{0} \qquad \ulcorner m+1 \urcorner = \mathsf{S}\ulcorner m \urcorner$$

for each $m \in \mathbb{N}$. You should not confuse these numerals with the church numerals. The two kinds of numerals are related, but they are not the same.

4.8 EXAMPLE. For each $m \in \mathbb{N}$ the judgement to the left is derivable.

$$\vdash \ulcorner m \urcorner : \mathcal{N} \qquad\qquad \vdash \ulcorner 0 \urcorner : \mathcal{N} \qquad\qquad \dfrac{\vdash \mathsf{S} : \mathcal{N}' \qquad \dfrac{\vdots}{\vdash \ulcorner m \urcorner : \mathcal{N}}}{\vdash \ulcorner m' \urcorner : \mathcal{N}}$$

The templates to the right generate the required derivations.

For all terms s, t, the reduction

$$\mathsf{I}_\sigma \ulcorner m \urcorner ts \rhd\!\rhd t^m s$$

follows by an induction over m. For $m = 0$ we have

$$\mathsf{I}_\sigma \ulcorner 0 \urcorner ts = \mathsf{I}_\sigma \mathsf{0} ts \rhd s = t^0 s$$

for the base of the reduction. For the step $m \mapsto m'$ we have $\ulcorner m' \urcorner = \mathsf{S}\ulcorner m \urcorner$ and hence

$$\mathsf{I}_\sigma \ulcorner m' \urcorner ts = \mathsf{I}_\sigma (\mathsf{S}\ulcorner m \urcorner) ts \rhd t(\mathsf{I}_\sigma \ulcorner m \urcorner ts) \rhd\!\rhd (t^m s) = t^{m'} s$$

using first the 1-step reduction and then the induction hypothesis. ■

This shows that the compound $\mathsf{I}_\sigma \ulcorner m \urcorner$ behaves something like the church numeral $\overline{m_\sigma}$. In particular, it suggests that anything we can do with church numerals we can do with genuine numerals, provided we have access to the appropriate iterator.

The whole point of this calculus is to get at numeric functions in a more honest fashion. We have seen how we do this for a natural number m (that is we use the numerals $\ulcorner m \urcorner$ rather than the battery of numerals $\overline{m_\zeta}$). What about functions themselves?

A simple exercise shows that

$$\mathsf{S}^m \ulcorner n \urcorner = \ulcorner n + m \urcorner$$

and with this idea we can get at addition and other simple functions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{u:\mathcal{N} \vdash \mathsf{I}:\mathcal{N}\to\mathcal{N}'' \quad u:\mathcal{N}\vdash u:\mathcal{N}}{u:\mathcal{N}\vdash \mathsf{I}u:\mathcal{N}''} \qquad u:\mathcal{N}\vdash \mathsf{S}:\mathcal{N}'}{u:\mathcal{N}\vdash \mathsf{I}u\mathsf{S}:\mathcal{N}'}}{\boxed{\vdash \mathsf{A}:\mathcal{A}}}}{v:\mathcal{N}\vdash \mathsf{A}:\mathcal{A}}}{\Gamma\vdash \mathsf{A}:\mathcal{A}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\vdash \mathsf{I}:\mathcal{N}\to\mathcal{N}'' \quad \Gamma\vdash u:\mathcal{N}}{\Gamma\vdash \mathsf{I}u:\mathcal{N}''} \qquad \dfrac{\Gamma\vdash \mathsf{A}:\mathcal{A} \qquad \Gamma\vdash v:\mathcal{N}}{\Gamma\vdash \mathsf{A}v:\mathcal{N}'}}{\Gamma\vdash \mathsf{I}u(\mathsf{A}v):\mathcal{N}'} \qquad \Gamma\vdash 0:\mathcal{N}}{\Gamma\vdash \mathsf{I}u(\mathsf{A}v)0:\mathcal{N}}}{v:\mathcal{N}\vdash \lambda u:\mathcal{N}\,.\,\mathsf{I}u(\mathsf{A}v)0:\mathcal{N}'}$$

$$\dfrac{\dfrac{v:\mathcal{N}\vdash \lambda u:\mathcal{N}\,.\,\mathsf{I}u(\mathsf{A}v)0:\mathcal{N}'}{\boxed{\vdash \mathsf{M}:\mathcal{A}}}}{v:\mathcal{N}\vdash \mathsf{M}:\mathcal{A}} \qquad \dfrac{v:\mathcal{N}\vdash \mathsf{M}:\mathcal{A}}{\Gamma\vdash \mathsf{M}:\mathcal{A}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\vdash \mathsf{I}:\mathcal{N}\to\mathcal{N}'' \quad \Gamma\vdash u:\mathcal{N}}{\Gamma\vdash \mathsf{I}u:\mathcal{N}''} \qquad \dfrac{\Gamma\vdash \mathsf{M}:\mathcal{A} \qquad \Gamma\vdash v:\mathcal{N}}{\Gamma\vdash \mathsf{M}v:\mathcal{N}'}}{\Gamma\vdash \mathsf{I}u(\mathsf{M}v):\mathcal{N}'} \qquad \dfrac{\Gamma\vdash \mathsf{S}:\mathcal{N}' \quad \Gamma\vdash 0:\mathcal{N}}{\Gamma\vdash \ulcorner 1\urcorner:\mathcal{N}}}{\Gamma\vdash \mathsf{I}u(\mathsf{M}v)\ulcorner 1\urcorner:\mathcal{N}}}{v:\mathcal{N}\vdash \lambda u:\mathcal{N}\,.\,\mathsf{I}u(\mathsf{M}v)0:\mathcal{N}'}$$

$$\dfrac{v:\mathcal{N}\vdash \lambda u:\mathcal{N}\,.\,\mathsf{I}u(\mathsf{M}v)0:\mathcal{N}'}{\boxed{\vdash \mathsf{E}:\mathcal{A}}}$$

Table 4.1: An example derivation

4.9 EXAMPLES. Let $\mathsf{I}=\mathsf{I}_{\mathcal{N}}$ and let

$$\mathsf{A}=\lambda u:\mathcal{N}\,.\,\mathsf{I}u\mathsf{S} \qquad \mathsf{M}=\lambda v,u:\mathcal{N}\,.\,\mathsf{I}u(\mathsf{A}v)\ulcorner 0\urcorner \qquad \mathsf{E}=\lambda v,u:\mathcal{N}\,.\,\mathsf{I}u(\mathsf{M}v)\ulcorner 1\urcorner$$

to produce three raw terms. Let

$$\Gamma = v:\mathcal{N}, u:\mathcal{N} \qquad \mathcal{A}=\mathcal{N}\to\mathcal{N}\to\mathcal{N}=\mathcal{N}\to\mathcal{N}'$$

to produce a legal context (assuming u, v are distinct) and a type. With these the derivation of Table 4.1 shows that

$$\vdash \mathsf{A}:\mathcal{A} \qquad \vdash \mathsf{M}:\mathcal{A} \qquad \vdash \mathsf{E}:\mathcal{A}$$

and hence the three terms are well-formed.

For each $m, n \in \mathbb{N}$ we have

$$\mathsf{A}\ulcorner m\urcorner\ulcorner n\urcorner \mathrel{\rhd\!\!\rhd} \mathsf{I}\ulcorner m\urcorner\mathsf{S}\ulcorner n\urcorner \mathrel{\rhd\!\!\rhd} \mathsf{S}^{m}\ulcorner n\urcorner = \ulcorner n+m\urcorner$$

to show how A captures addition.

Similarly we have

$$\mathsf{M}\ulcorner n\urcorner\ulcorner m\urcorner \mathrel{\rhd\!\!\rhd} \mathsf{I}\ulcorner m\urcorner(\mathsf{A}\ulcorner n\urcorner)\ulcorner 0\urcorner \mathrel{\rhd\!\!\rhd} (\mathsf{A}\ulcorner n\urcorner)^{m}\ulcorner 0\urcorner \mathrel{\rhd\!\!\rhd} \ulcorner n\times m\urcorner$$

where the last step requires an auxiliary argument.

Finally we have

$$\mathsf{E}\ulcorner n\urcorner\ulcorner m\urcorner \rhd\!\rhd \mathsf{I}\ulcorner m\urcorner(\mathsf{M}\ulcorner n\urcorner)\ulcorner 1\urcorner \rhd\!\rhd (\mathsf{M}\ulcorner n\urcorner)^m\ulcorner 1\urcorner \rhd\!\rhd \ulcorner n^m\urcorner$$

where the last step requires an auxiliary argument. ■

These examples are very like, but not the same as, the earlier simulations of addition, multiplication, and exponentiation. The use of numerals rather then church numerals means we should give the idea a different name.

4.10 DEFINITION. Let f be an l-placed function on $\mathbb{N}$ and let $\ulcorner f\urcorner$ be a λ-term with

$$\vdash \ulcorner f\urcorner : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N}$$

where there are $l+1$ occurrences of $\mathcal{N}$, that is l occurrences in an input positions and one in the eventual target position. We say $\ulcorner f\urcorner$ represents f (in the calculus) if the reduction

$$\ulcorner f\urcorner\ulcorner m_l\urcorner \cdots \ulcorner m_1\urcorner \rhd\!\rhd\!\rhd \ulcorner m_0\urcorner$$

holds for all $m_0, m_1, \ldots, m_l \in \mathbb{N}$ with $f(m_l, \ldots, m_1) = m_0$. ■

Notice that it is the reflexive version of the reduction relation that is used here. This is to cut out some silly problems, and enables us to prove the following.

4.11 LEMMA. *The constant* S *and the three terms* $\mathsf{A}, \mathsf{M}, \mathsf{E}$ *(of Examples 4.9) represent the successor function, addition, multiplication, and exponentiation, respectively.*

Proof. For each $m \in \mathbb{N}$ we have

$$\mathsf{S}\ulcorner m\urcorner \rhd\!\rhd\!\rhd \ulcorner m+1\urcorner \quad \text{in fact} \quad \mathsf{S}\ulcorner m\urcorner = \ulcorner m+1\urcorner$$

to give the representation of successor. The other three representations (for which we can use $\rhd\!\rhd$) are verified by Example 4.9. ■

This is the reason that we use $\rhd\!\rhd\!\rhd$ in Definition 4.10. We do *not* have

$$\mathsf{S}\ulcorner m\urcorner \rhd\!\rhd \ulcorner m+1\urcorner$$

but clearly we do want to use S as the canonical name of the successor function.

The Examples 4.9 can be continued in a quite natural way.

4.12 EXAMPLES. Let

$$\mathsf{B} = \lambda u, v, w : \mathcal{N} \,.\, \mathsf{I}u(\mathsf{E}v)w$$

where $\mathsf{I} = \mathsf{I}_{\mathcal{N}}$ and E is the term of Examples 4.9. Then

$$\vdash \mathsf{B} : \mathcal{N} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}$$

and

$$\mathsf{B}\ulcorner T\urcorner\ulcorner m\urcorner\ulcorner n\urcorner \rhd\!\rhd (\mathsf{E}\ulcorner m\urcorner)^l\ulcorner n\urcorner \rhd\!\rhd \ulcorner \beth(l, m, n)\urcorner$$

for all $l, m, n \in \mathbb{N}$.

Let

$$\beth(x) = \beth(x, 2, 1)$$

to obtain a 1-placed function $\beth$ which outstrips all functions that can be simulated in the simply typed calculus. The term

$$\mathsf{D} = \lambda u : \mathcal{N} \,.\, \mathsf{I}u(\mathsf{E}\ulcorner 2\urcorner)\ulcorner 1\urcorner$$

represents $\beth$.

The function

$$\daleth(x) = \beth^x(2)$$

quickly outstrips the entire universe (for try calculating $\daleth(4)$). The term

$$\mathsf{G} = \lambda u : \mathcal{N} \,.\, \mathsf{I}u\mathsf{D}\ulcorner 2\urcorner$$

represents $\daleth$. ■

This trick can be continued for some time, which gives us an inkling of the strength of the calculus. All this is achieved using no more than the iterator $\mathsf{I}_{\mathcal{N}}$. What can be done with the other iterators? In fact, all the functions considered so far are primitive recursive.

Exercises

4.2 Show that the class of functions represented in the applied calculus is a clone.

4.3 The notion of 'simulation' still makes sense for the applied calculus. Show that if a term

$$\vdash \overline{f} : \zeta(l)'' \to \cdots \to \zeta(1)'' \to \mathcal{N}''$$

simulates an l-placed function f, then the function is also represented in the calculus.

Does a similar result hold if the target of the simulation is ζ'' for any arbitrary type ζ?

4.3 Closure under recursions

The Examples 4.12 show that even with just the one iterator $\mathsf{I}_{\mathcal{N}}$ this applied calculus $\boldsymbol{\lambda}\mathsf{G}$ is stronger than the simply typed calculus (since it is easy to represent a function which outstrips the stacking functions $\beth$ and this outstrips every function which is simulated in the simply typed calculus). How strong is $\boldsymbol{\lambda}\mathsf{G}$? The answer, of course, depends on which iterators we have available. We will look at this question in more detail in the next section, but for now let's look at some closure properties that the calculus has.

We consider first at how primitive recursion can be handled.

In Theorem 2.18 we showed that the class of functions simulated in the untyped calculus is closed under primitive recursion. That proof can be lifted to obtain a similar result for $\boldsymbol{\lambda}\mathsf{G}$.

4.13 EXAMPLE. Let

$$g : \mathbb{N}^l \longrightarrow \mathbb{N} \qquad h : \mathbb{N}^l \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

be two data functions with typical values

$$g(y_l, \ldots, y_1) \qquad\qquad h(y_l, \ldots y_1, x, z)$$

for $x, y_1, \ldots, y_l, z \in \mathbb{N}$. We use x as the recursion variable, with $y_1, \ldots, y_l$ as the parameters, and z as an auxiliary. Consider the function

$$f : \mathbb{N}^l \times \mathbb{N} \longrightarrow \mathbb{N}$$

given by the primitive recursion

$$f(y_l, \ldots, y_1, 0) = g(y_l, \ldots, y_1) \qquad f(y_l, \ldots, y_1, x') = h(y_l, \ldots, y_1, x, f(y_l, \ldots, y_1, x))$$

for $x, y_1, \ldots, y_l \in \mathbb{N}$. We suppose the two functions g, h are represented by terms

$$\vdash \ulcorner g \urcorner : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N} \qquad \vdash \ulcorner h \urcorner : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}$$

and we wish to produce a representation

$$\vdash \ulcorner f \urcorner : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}$$

of f.

To do that we first rework the construction of f into an iteration. Let

$$H : \mathbb{N}^l \longrightarrow \mathbb{N}^2 \longrightarrow \mathbb{N}^2$$

be given by

$$H(y_l, \ldots, y_1)(x, z) = (x', h(y_l, \ldots, y_1, x, z)$$

for the usual $x, y_1, \ldots, y_l, z$. The crucial observation is that

$$H\mathsf{y}(x, f(\mathsf{y}, x)) = (x', f(\mathsf{y}, x'))$$

for each x and $\mathsf{y} = (y_l, \ldots, y_1)$.

Now let

$$F : \mathbb{N}^l \times \mathbb{N} \longrightarrow \mathbb{N}^2$$

be given by

$$F(\mathsf{y}, x) = (H\mathsf{y})^x(0, g\mathsf{y})$$

so that

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

by a simple induction over x. In particular, the required function is obtained by extracting the right component from F.

To obtain $\ulcorner f \urcorner$ we mimic the construction in the calculus. To do this we make use of the pairing gadgets.

Let $\ulcorner H \urcorner$ be the term

$$\lambda y_l, \ldots, y_1 : \mathcal{N}, w : \mathcal{N}^2 \,.\, \mathsf{Pair}\big(\mathsf{S}(\mathsf{Left}w)\big)\big(\ulcorner h \urcorner y_l \cdots y_1(\mathsf{Left}w)(\mathsf{Right}w)\big)$$

so that

$$\vdash \ulcorner H \urcorner : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N}^2 \to \mathcal{N}^2$$

and $\ulcorner H \urcorner$ represents H.

The recursion is obtained using the iterator $\mathsf{I}_{\mathcal{N}^2}$. Thus, let $\ulcorner F \urcorner$ be the term

$$\lambda y_l, \ldots, y_1 : \mathcal{N}, x : \mathcal{N} \,.\, (\mathsf{I}_{\mathcal{N}^2} x)(\ulcorner H \urcorner \mathsf{y})\big(\mathsf{Pair}\ulcorner 0 \urcorner(\ulcorner g \urcorner \mathsf{y})\big)$$

so that

$$\vdash \ulcorner F \urcorner : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}^2$$

and $\ulcorner F \urcorner$ represents F. In this term we have condensed the parameters to y.

Finally let $\ulcorner f \urcorner$ be the term

$$\lambda y_l, \ldots, y_1 : \mathcal{N}, x : \mathcal{N} \,.\, \mathsf{Right}(\ulcorner F \urcorner \mathsf{y} x)$$

to obtain the representation of f. ■

There is still a bit of work to be done to complete this example, but the details are essentially the same as for In Theorem 2.18. With these we have the following result.

4.14 THEOREM. *Using only the iterators for the types $\Sigma = \{\mathcal{N}, \mathcal{N}^2\}$ the class of functions that can be represented in the calculus is closed under primitive recursion.*

Just in case you have been fooled it should be pointed out that a slight fast one has been pulled in this proof. There are claims that

$$\ulcorner H \urcorner \text{ represents } H \qquad \ulcorner F \urcorner \text{ represents } F$$

for two constructed terms $\ulcorner H \urcorner$ and $\ulcorner F \urcorner$. What do these claims mean? Look at Definition 4.10. Neither of the function H, F can be represented because neither has the right form of type. However, this is not a serious flaw and is easily corrected.

Even with just the two iterators based on $\Sigma = \{\mathcal{N}, \mathcal{N}^2\}$ the calculus is very much stronger than the simply typed calculus. But how much stronger? We still have all the other iterators, but perhaps these give us nothing new. Let's nail that one straight away.

In Chapter 2 we saw that the untyped calculus is Turing complete in the sense that it can simulate every general recursive function. The trick to this is to find a way to handle unbounded search. This applied calculus of this chapter is certainly not Turing complete, but it can get way beyond the primitive recursive functions. To give an illustration of this we need a form of recursion which is stronger than primitive recursion but is still reasonably simple.

4.15 EXAMPLE. For an arbitrary 1-placed function $f : \mathbb{N} \longrightarrow \mathbb{N}$ let $F : \mathbb{N}^3 \longrightarrow \mathbb{N}$ be the 3-placed function specified by

$$F(0,0,x) = fx \qquad F(i',0,x) = F(i,x,x) \qquad F(i,r',x) = F(i,0,F(i,r,x))$$

for $i, r, x \in \mathbb{N}$. This F is the Ackermann modification of f. ■

When f is the successor function this f is Ackermann's function which outstrips every primitive recursive function. In particular, this F is not primitive recursive. (As an aside, most textbook use a 2-place 'Ackermann function' which is reputedly simpler than the 3-placed function. This 2-paced function is not due to Ackermann, and is certainly not simpler to analyse in the way usually described.)

The specification of F uses a recursion over two arguments and a nested call on itself. It is this aspect which make the function F complicated. However, there is a much simpler way to describe F. This is due to Hilbert. The trick is to use a higher order function.

4.16 DEFINITION. Let

$$Ack : \mathbb{N}' \longrightarrow \mathbb{N}'$$

be the higher order function given by

$$Ackfx = f^{x+1}x$$

for each $f \in \mathbb{N}'$ and $x \in \mathbb{N}$. This function Ack is the Ackermann jump operator. ■

What is the connection between the Ackermann jump and the Ackermann modification? They are the same process.

4.17 LEMMA. *Let f be a 1-placed function with Ackermann modification F. Then*

$$F(i, r, x) = (Ack^i f)^{r+1}x$$

for each $i, r, x \in \mathbb{N}$.

Proof. This follows by a nested induction over i and r. ■

If you can not fill in the details of this proof then your mathematical education has a least one gap that ought to be filled.

The point of this construction is that the Ackermann jump is easily captured in the applied calculus.

4.18 DEFINITION. Using the iterator $\mathsf{I}_{\mathcal{N}}$ let

$$\ulcorner Ack \urcorner = \lambda f : \mathcal{N}', x : \mathcal{N} \,.\, \mathsf{I}_{\mathcal{N}}(\mathsf{S}x)fx$$

to obtain a term

$$\vdash \ulcorner Ack \urcorner : \mathcal{N}''$$

in the applied calculus. ■

Notice that the construction of the term $\ulcorner Ack \urcorner$ follows exactly the construction of the operator Ack, and uses no more than the lowest level operator $\mathsf{I}_{\mathcal{N}}$. The following result shows that $\ulcorner Ack \urcorner$ 'names' Ack in a sense not made precise here.

4.19 LEMMA. *Suppose the 1-placed function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is represented by the term $\vdash \ulcorner f \urcorner : \mathcal{N}'$. Then the term*

$$\ulcorner F \urcorner = \lambda i, r : \mathcal{N} \,.\, \mathsf{I}_{\mathcal{N}}(\mathsf{S}r)(\mathsf{I}_{\mathcal{N}'} i \ulcorner Ack \urcorner \ulcorner f \urcorner)$$

satisfies

$$\vdash \ulcorner F \urcorner : \mathcal{N} \to \mathcal{N} \to \mathcal{N} \to \mathcal{N}$$

and represents the Ackermann modification F of f.

Proof. For each $i, r, m \in \mathbb{N}$ we have

$$\begin{aligned} \ulcorner F \urcorner \ulcorner i \urcorner \ulcorner r \urcorner \ulcorner m \urcorner &\rhd\!\rhd \mathsf{I}_{\mathcal{N}}(\mathsf{S}\ulcorner r \urcorner)(\mathsf{I}_{\mathcal{N}'} \ulcorner i \urcorner \ulcorner Ack \urcorner \ulcorner f \urcorner) \ulcorner m \urcorner \\ &\rhd\!\rhd \mathsf{I}_{\mathcal{N}}(\ulcorner r + 1 \urcorner)(\ulcorner Ack \urcorner^{\ulcorner i \urcorner} \ulcorner f \urcorner) \ulcorner m \urcorner \\ &\rhd\!\rhd (\ulcorner Ack \urcorner^{\ulcorner i \urcorner} \ulcorner f \urcorner)^{r+1} \ulcorner m \urcorner \qquad \rhd\!\rhd \ulcorner (Ack^i f)^{r+1} m \urcorner \end{aligned}$$

where the last reduction follows by a double induction. ■

This gives us a function that is representable but is not primitive recursive, and to do that we need no more than the iterators based on $\Sigma = \{\mathcal{N}, \mathcal{N}^2, \mathcal{N}'\}$. There is a lot more to come.

Exercises

4.4 The applied calculus is closed under many forms of recursion, some much more powerful that primtive recursion. This exercise deals with a quite modest extension.

Let

$$g : \mathbb{N} \longrightarrow \mathbb{N} \qquad h : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \qquad k : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

be three data functions and suppose the function

$$f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

is specified by

$$f(y,0) = gy \qquad f(y,x') = h(y,x,f(w,x)) \quad \text{where } w = k(y,x)$$

for $x, y \in \mathbb{N}$. (This is a form of primitive recursion with variation of parameters.) Show that the three functions g, h, k are representable then so is f.

4.5 Organize a proof of Lemma 4.17 and of the final reduction in the proof of Lemma 4.19.

4.6 The Robinson-Péter jump is the higher order function $Rob : \mathbb{N}''$ given by

$$Robfx = f^{x+1}1$$

for each $f : \mathbb{N}, x : \mathbb{N}$.

For an arbitrary 1-placed function $f : \mathbb{N} \longrightarrow \mathbb{N}$ the Robinson-Péter modification of f is the 2-placed function F specified by

$$F(0,x) = fx \qquad F(i',0) = F(i,1) \qquad F(i',x') = F(i,F(i',x))$$

for $i, x \in \mathbb{N}$.

(a) Show that

$$F(i,x) = Rob^i fx$$

for each $i, x \in \mathbb{N}$.

(b) Write down a term $\ulcorner Rob \urcorner$ which names Rob.

(c) Using your term $\ulcorner Rob \urcorner$ show how to convert a name $\ulcorner f \urcorner$ for f into a name $\ulcorner F \urcorner$ of the RP-modification of f.

(d) Verify directly that

$$\ulcorner F \urcorner \ulcorner n \urcorner \ulcorner m \urcorner \rhd\!\rhd \ulcorner Rob^n fm \urcorner$$

for $m, n \in \mathbb{N}$.

4.7 The Grzegorczyk jump is the higher order function $Grz : \mathbb{N}''$ given by

$$Grzfx = f^x 2$$

for each $f : \mathbb{N}, x : \mathbb{N}$.

For an arbitrary 1-placed function $f : \mathbb{N} \longrightarrow \mathbb{N}$ the Grzegorczyk modification of f is the 2-placed function F specified by

$$F(0,x) = fx \qquad F(i',0) = 2 \qquad F(i',x') = F(i,F(i',x))$$

for $i, x \in \mathbb{N}$.

(a) Show that

$$F(i,x) = Grz^i fx$$

for each $i, x \in \mathbb{N}$.

(b) Write down a term $\ulcorner Grz \urcorner$ which names Grz.

(c) Using your term $\ulcorner Grz \urcorner$ show how to convert a name $\ulcorner f \urcorner$ for f into a name $\ulcorner F \urcorner$ of the G-modification of f.

(d) Verify directly that

$$\ulcorner F \urcorner \ulcorner n \urcorner \ulcorner m \urcorner \rhd\!\rhd \ulcorner Rob^n fm \urcorner$$

for $m, n \in \mathbb{N}$.

4.4 Way beyond

We have seen that with no more than the two iterators based on $\mathcal{N}$ and $\mathcal{N}'$ we can capture in the calculus a function that outstrips every primitive recursive function. What can be done using iterators on more complicated types? To motivate the answer to this let's remember a version of Ackermann's original result. To state this it is convenient to have a bit of serpentine terminology.

A snake is a 1-placed function f which is strictly inflationary and monotone, that is

$$x < fx \qquad x \leq y \Longrightarrow fx \leq fy$$

for all $x, y \in \mathbb{N}$.

4.20 PROPOSITION. *For each snake f and each function which is primitive recursive in f, there is some $i < \omega$ such that the 1-placed function $Ack^i f$ outstrips that function.*

This result has an immediate consequence, which is precisely why Ackermann produced his function.

4.21 COROLLARY. *For each snake f the function*

$$x \longmapsto Ack^{x+1} fx$$

is not primitive recursive in f.

With some notation we can get a bit more out of this result. The following should be compared with Definition 4.16.

4.22 DEFINITION. Let

$$Bck : \mathbb{N}'' \longrightarrow \mathbb{N}''$$

be the higher order function given by

$$BckFfx = F^{x+1} fx$$

for each $F \in \mathbb{N}''$, $f \in \mathbb{N}'$ and $x \in \mathbb{N}$. This is the Ackermann superjump operator. ■

$$\begin{array}{llll} f & Ack\, f & Ack^2 f & \cdots & Ack^i f & \cdots \\ f & Bck\, Ack\, f & Bck^2 Ack f & \cdots & Bck^j Ack f & \cdots \\ f & Cck\, Bck\, Ack\, f & Cck^2 Bck Ack f & \cdots & Cck^k Bck Ack f & \cdots \\ f & Dck\, Cck\, Bck\, Ack\, f & Dck^2 Cck Bck Ack f & \cdots & Dck^l Cck Bck Ack f & \cdots \\ & \vdots & & & & \end{array}$$

Table 4.2: A hierarchy of complexities

Using this Corollary 4.21 says that for each snake f the function $Bck\, Ack f$ is not primitive recursive in f. Furthermore, it is not too hard to show that $Bck\, Ack f$ is a snake (whenever f is) so we can produce a hierarchy of snakes

$$f \quad Bck\, Ack f \quad Bck^2 Ack f \quad Bck^3 Ack f \quad \ldots \quad Bck^j Ack f \quad \ldots$$

where each later one is not primitive recursive in each earlier one.

It is easy to produce even bigger jumps in complexity.

4.23 DEFINITION. Let

$$Cck : \mathbb{N}''' \longrightarrow \mathbb{N}''' \qquad Dck : \mathbb{N}^{(4)} \longrightarrow \mathbb{N}^{(4)}$$

be the higher order functions given by

$$Cck\phi F f x = \phi^{x+1} F f x \qquad Dck\Phi\phi F f x = \Phi^{x+1}\phi F f x$$

for each $\Phi \in \mathbb{N}^{(4)}, \phi \in \mathbb{N}''', F \in \mathbb{N}'', f \in \mathbb{N}'$ and $x \in \mathbb{N}$. These are two of the higher order Ackermann jump operators. ■

Using these we can produce a whole array of functions, as in Table 4.2 where the jump in complexity for each step along a row is bigger than the jump along the whole of previous row. The hierarchy continues in the obvious way.

What is happening here is that the higher level functions are being combined into operators of type $\mathbb{N}' \longrightarrow \mathbb{N}'$ which jump up the rate of growth (and hence complexity) of a snake. For instance

$$Ack^2 \qquad (Bck^3 Ack)^4 \qquad ((Cck^5 Bck)^6 Ack)^7 \qquad (((Dck^8 Cck)^9 Bck)^{10} Ack)^{11}$$

are four such operators.

All of the higher order functions can be captured in the calculus. To describe how this is done it is convenient to have some abbreviations.

As before we set

$$\mathcal{N}^{(0)} = \mathcal{N} \qquad \mathcal{N}^{(l+1)} = \mathcal{N}^{(l)\prime}$$

(for $l < \omega$) to generate an ascending chain of types. Let

$$\mathsf{I}_l = \mathsf{I}_{\mathcal{N}^{(l)}}$$

so that $\mathsf{I}_0, \mathsf{I}_1, \mathsf{I}_2, \ldots$ are progressive stronger iterators.

4.24 DEFINITION. For each $l < \omega$ we set

$$\ulcorner Ack_l \urcorner = \lambda x_{l+1} : \mathcal{N}^{(l+1)}, x_l : \mathcal{N}^{(l)}, \ldots, x_0 : \mathcal{N}^{(0)} \,.\, \mathsf{I}_l(\mathsf{S}x) x_{l+1} x_l \cdots x_0$$

to obtain a term such that

$$\vdash \ulcorner Ack_l \urcorner : \mathcal{N}^{(l+2)}$$

holds. ■

It is not too hard to see that

$$\begin{array}{lll} \ulcorner Ack_0 \urcorner & \text{captures} & Ack \\ \ulcorner Ack_1 \urcorner & \text{captures} & Bck \\ \ulcorner Ack_2 \urcorner & \text{captures} & Cck \\ \ulcorner Ack_3 \urcorner & \text{captures} & Dck \\ & \vdots & \end{array}$$

and so on. Clearly, this is a basis for a more systematic notation. In particular, $\ulcorner Ack_0 \urcorner$ is just the term $\ulcorner Ack \urcorner$, and the terms

$$\ulcorner Ack \urcorner^2 \quad (\ulcorner Bck \urcorner^3 \ulcorner Ack \urcorner)^4 \quad ((\ulcorner Cck \urcorner^5 \ulcorner Bck \urcorner)^6 \ulcorner Ack \urcorner)^7 \quad (((\ulcorner Dck \urcorner^8 \ulcorner Cck \urcorner)^9 \ulcorner Bck \urcorner)^{10} \ulcorner Ack \urcorner)^{11}$$

capture the four operators given above. (Strictly speaking these are not terms, but are easily converted into terms which do the required job.)

This is the beginnings of a quite intricate classification and measure of complexity of those functions that are represented in the calculus.

The finest (least course) such measure is the very syntax used to represent the function. This is quite a good measure for the syntax tracks quite closely the given recursive specification of the function. So not only is the function captured, but we also have an indication of the complexity of the evaluation algorithm.

A courser measure is obtained from the iterators used in the syntax. We locate the iterator with the most complex base type, and then we count the number of nestings used in the representation.

Finally, we can convert this into an ordinal measure using the ordinals

$$0, 1 \cdots, \omega, \cdots, \omega^\omega, \cdots, \omega^{\omega^\omega}, \cdots, \omega^{\omega^{\omega^\omega}}, \cdots$$

up to a certain ordinal ϵ_0. For instance, the ordinals

$$2 \qquad \omega^3 \cdot 4 \qquad \omega^{\omega^5 \cdot 6} \cdot 7 \qquad \omega^{\omega^{\omega^8 \cdot 9} \cdot 10} \cdot 11$$

are the measures of the operators given above.

However, we do not have time to look at such things.

Part II

Solutions

What follows is a fairly full set of solutions, but there are some ommissions. Further solutions will be added in due course.

A
The solutions

A.1 For chapter 1

A.1.1 For section 1.1

1.1 (a) The parsing tree of each of these shouldn't give you too much trouble. Ask if you are in doubt. Note that each term is a combinator so that the eventual support is empty.

(b) This term really is daft. It uses x is four different ways. Here is its parsing tree and some expanded information.

$$\begin{array}{lllll}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(-1-)}(x)}{(-2-)}x}{(-3-)}(x)}{(-4-)}x}{(-5-)}x}{(-6-)}(x)}{(-7-)}(x)}{(-8-)}(x)
& (-1-) = & (\lambda x\,.\,x) = & (\lambda x\,.\,x) & \partial\emptyset \\
& (-2-) = & ((-1-)x) = & ((\lambda x\,.\,x)x) & \partial\{x\} \\
& (-3-) = & (\lambda x\,.\,(-2-)) = & (\lambda x\,.\,((\lambda x\,.\,x)x)) & \partial\emptyset \\
& (-4-) = & ((-3-)x) = & ((\lambda x\,.\,((\lambda x\,.\,x)x))x) & \partial\{x\} \\
& (-5-) = & ((-4-)x) = & (((\lambda x\,.\,((\lambda x\,.\,x)x))x)x) & \partial\{x\} \\
& (-6-) = & ((-5-)x) = & ((((\lambda x\,.\,((\lambda x\,.\,x)x))x)x)x) & \partial\{x\} \\
& (-7-) = & (\lambda x\,.\,(-6-)) = & (\lambda x\,.\,((((\lambda x\,.\,((\lambda x\,.\,x)x))x)x)x)) & \partial\emptyset \\
& (-8-) = & (\lambda x\,.\,(-7-)) = & (\lambda x\,.\,(\lambda x\,.\,((((\lambda x\,.\,((\lambda x\,.\,x)x))x)x)x))) & \partial\emptyset
\end{array}$$

The alphabetic variant

$$(\lambda u\,.\,(\lambda v\,.\,((((\lambda w\,.\,((\lambda x\,.\,x)w))v)v)v)))$$

gives a better idea of the term. Here is the parsing tree and expanded information of that term.

$$\begin{array}{lllll}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(-1-)}(x)}{(-2-)}w}{(-3-)}(w)}{(-4-)}v}{(-5-)}v}{(-6-)}(v)}{(-7-)}(v)}{(-8-)}(u)
& (-1-) = & (\lambda x\,.\,x) = & (\lambda x\,.\,x) & \partial\emptyset \\
& (-2-) = & ((-1-)w) = & ((\lambda x\,.\,x)w) & \partial\{w\} \\
& (-3-) = & (\lambda w\,.\,(-2-)) = & (\lambda w\,.\,((\lambda x\,.\,x)w)) & \partial\emptyset \\
& (-4-) = & ((-3-)v) = & ((\lambda w\,.\,((\lambda x\,.\,x)w))v) & \partial\{v\} \\
& (-5-) = & ((-4-)v) = & (((\lambda w\,.\,((\lambda x\,.\,x)w))v)v) & \partial\{v\} \\
& (-6-) = & ((-5-)v) = & ((((\lambda w\,.\,((\lambda x\,.\,x)w))v)v)v) & \partial\{v\} \\
& (-7-) = & (\lambda v\,.\,(-6-)) = & (\lambda v\,.\,((((\lambda w\,.\,((\lambda x\,.\,x)w))v)v)v)) & \partial\emptyset \\
& (-8-) = & (\lambda u\,.\,(-7-)) = & (\lambda u\,.\,(\lambda v\,.\,((((\lambda w\,.\,((\lambda x\,.\,x)w))v)v)v))) & \partial\emptyset
\end{array}$$

Notice that the outside abstraction is vacuous. ■

1.2 This kind of thing is messy. We want to show that the two iterated substitution operators

$$[y := s][x := r] \qquad [x := r][y := s']$$

are 'equivalent' where $s' = s[x := r]$. We show

$$t[y := s][x := r] = t[x := r][y := s']$$

by induction over the structure of t. However, the induction hypothesis is not exactly straight forward. The crucial case is the step across an abstraction

$$t = (\lambda u \,.\, p)$$

which proceeds something like the following.

$$\begin{array}{ll}
(\lambda u \,.\, p)[y := s][x := r] & (\lambda u \,.\, p)[x := r][y := s'] \\
(\lambda v \,.\, (p[u := v][y := s]))[x := r] & (\lambda v \,.\, (p[u := v][x := r]))[y := s'] \\
(\lambda w \,.\, p[u := v][y := s][v := w][x := r]) & (\lambda w \,.\, p[u := v][x := r][v := w][y := s'])
\end{array}$$

The two substitutions are done in parallel moving downwards. Here v, w are 'suitably fresh' identifiers. In other words we choose them so that they can not interfere with anything. It suffices to show that

$$p[u := v][y := s][v := w][x := r] = p[u := v][x := r][v := w][y := s']$$

holds. However, our choices of v, w ensures that

$$\begin{aligned}
p[u := v][y := s][v := w][x := r] &= p[u := w][y := s][x := r] \\
p[u := v][x := r][v := w][y := s'] &= p[u := w][x := r][y := s']
\end{aligned}$$

holds! The result follows by applying the induction hypothesis to the term $p[u := w]$.

There are several dodgy steps here. Luckily we don't need to get embroiled in this kind of thing here. ■

A.1.2 For section 1.2

1.3 The term tx contains three redexes, one associated with each abstraction. Any of these can be removed to produce a term with two redexes. Either of these can be removed to produce a term which is a redex. Finally, this can be removed. This gives six different reduction paths, all of which end with x.

The situation can be seen better if we first rename the bound identifiers in the term t. Thus consider

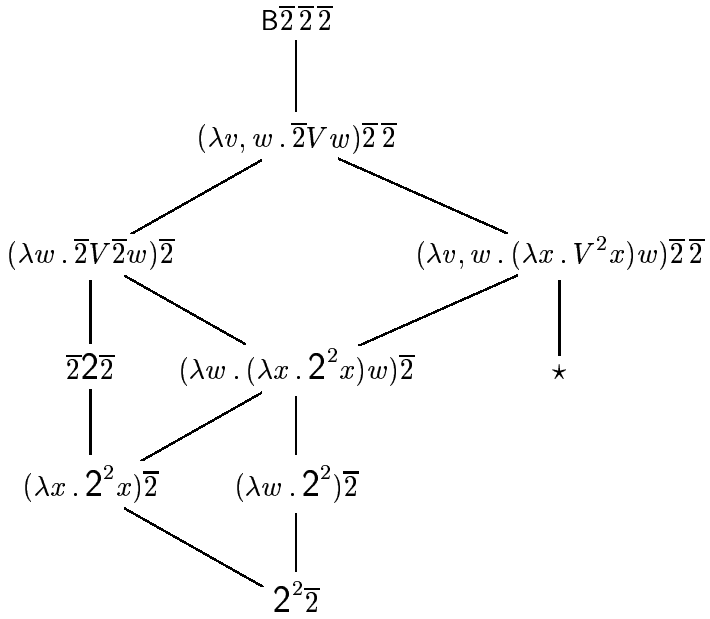
$$t = (\lambda u \,.\, ((\lambda v \,.\, ((\lambda w \,.\, w)v))u))$$

where u, v, w, x are distinct. The diagram

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x}}{((\lambda u \,.\, u)x)} \quad \dfrac{\dfrac{}{x}}{((\lambda v \,.\, v)x)}}{((\lambda u \,.\, ((\lambda v \,.\, v)u)x)} \quad \dfrac{\dfrac{\dfrac{}{x}}{((\lambda u \,.\, u)x)} \quad \dfrac{\dfrac{}{x}}{((\lambda w \,.\, w)x)}}{((\lambda u \,.\, ((\lambda w \,.\, w)u)x)} \quad \dfrac{\dfrac{\dfrac{}{x}}{((\lambda v \,.\, v)x)} \quad \dfrac{\dfrac{}{x}}{((\lambda w \,.\, w)x)}}{((\lambda v \,.\, ((\lambda w \,.\, w)v)x)}}{((\lambda u \,.\, ((\lambda v \,.\, ((\lambda w \,.\, w)v))u))x)}$$

shows the six reduction paths from tx. ■

1.4 Here are several initial reduction paths.



The $\star$ is the result of unravelling the inner redex V^2x.

I have no idea how many reductions paths there are, but I'm pretty sure they don't all have the same length. However, all paths terminate in $\overline{2^4}$. ■

A.1.3 For section 1.3

1.5 (a) Consider the combinators K and I of Table 1.1. We have

$$\mathsf{K}\mathsf{I}t \rhd \mathsf{I}$$

for each term t, and I is normal. Consider any combinator Δ for which

$$\Delta \rhd \Delta$$

(say that of Example 1.17). Let

$$\Omega = \mathsf{K}\mathsf{I}\Delta$$

so that

$$\Omega \rhd \mathsf{I} \qquad \Omega \rhd \Omega \rhd \Omega \rhd \cdots$$

by attacking either the K or the Δ.

(b) Consider any term

$$\Delta = (\lambda x\,.\,\overline{0})\delta$$

where δ is any term (with empty support) with at least one infinite reduction path. We can either attach the whole term (which is a redex) or contually attack δ to get one of

$$\Delta \rhd \overline{0} \qquad \Delta = (-)\delta \rhdd (-)\delta_1 \rhdd (-)\delta_2 \rhdd \cdots$$

to give the required result.

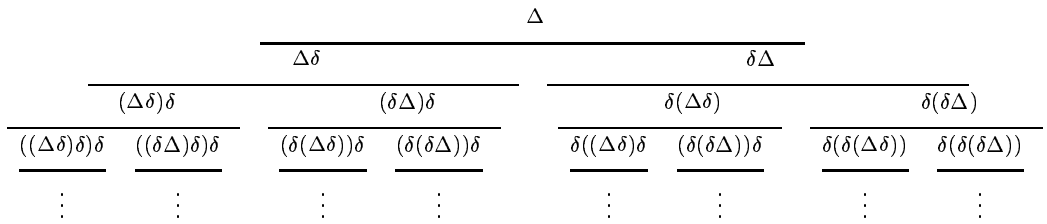
(c) Consider any term δ (with empty support) with

$$\delta \rhd\!\!\rhd \Delta$$

where $\Delta = (\delta\delta)$. We have

$$\Delta \rhd\!\!\rhd \Delta\delta \qquad \Delta \rhd\!\!\rhd \delta\Delta$$

and by repeating this we can generate many paths.



This will produce continuum many paths. ■

1.6 The following argument is typical of many used in the λ-calculus.

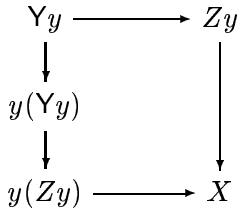
By way of contradiction suppose

$$\mathsf{Y} \rhd\!\!\rhd Z$$

for some normal term Z. Then, for each identifier y we have

$$\mathsf{Y}y \rhd\!\!\rhd Zy \qquad \mathsf{Y}y \rhd\!\!\rhd y(\mathsf{Y}y) \rhd\!\!\rhd y(\mathsf{Z}y)$$

and hence the confluence gives some term X with

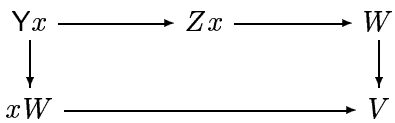


where each arrow indicates a reduction path (that is an instance of $\rhd\!\!\rhd\!\!\rhd$). The two terms Zy and $y(Zy)$ certainly are not the same (for they contain a different number of symbols), so at least one of them is reducible, not normal, (to achieve X). If Zy is not normal, then neither is $y(Zy)$. In other words $y(Zy)$ is not normal, and so must contain a redex.

What can this redex be? There are three possibilities, two of which we can rule out. Perhaps the whole term is a redex? This can't be since y is not an abstraction. Thus the redex must be contained in Zy. Perhaps the redex is contained in Z? This can't be since Z is assumed to be normal. Thus the term Zy itself must be a redex.

This shows that Z is an abstraction, and hence has the form $\lambda x\,.\,W$ for some term W. Notice that W is normal (otherwise Z is not normal). Similarly xW is normal.

Finally we have reductions



where V exists by a second use of confluence. But W and xW are both normal, so we can only have

$$xW = V = W$$

which is patent nonsense (by comparing the lengths of the two terms). ■

1.7 For an arbitrary term f let

$$F = \phi\phi \quad \text{where} \quad \phi = \lambda x\,.\,f(xx)$$

so that

$$\mathsf{Y}f \rhd\!\!\rhd \phi\phi = F \rhd\!\!\rhd f(\phi\phi) = fF$$

to give the required result. ■

A.2 For chapter 2

A.2.1 For section 2.1

2.1 (m) For each $m, n \in \mathbb{N}$ we have

$$\mathsf{M}\overline{n}\,\overline{m} \rhd\!\!\rhd \lambda y, x\,.\,\overline{n}(\overline{m}y)x \rhd\!\!\rhd \lambda y, x\,.\,(\overline{m}y)^n x$$

so that a reduction

$$[n] \quad (\overline{m}y)^n x \rhd\!\!\rhd\!\!\rhd y^{m\times n}x$$

will suffices. We proceed by induction. More precisely, we fix m, y, x and verify $(\forall n)[n]$ by a step-wise induction over n.

Since

$$(\overline{m}y)^0 x = x = y^{m\times 0}x$$

we have the base case [0]. This is the reason for the use of the reflexive relation in $[n]$.

For the induction step $n \mapsto n'$ we have

$$\begin{array}{rcll} (\overline{m}y)^{n+1}x & = & (\overline{m}y)((\overline{m}y)^n x) & \\ & \rhd\!\!\rhd\!\!\rhd & (\overline{m}y)(y^{m\times n}x) & \\ & \rhd\!\!\rhd\!\!\rhd & y^m(y^{m\times n}x) & \\ & = & y^{m\times n+m}x & = \quad y^{m\times(n+1)}x \end{array}$$

as required. Here the second step (the first reduction) uses the induction hypothesis.

(e) For each $m, n \in \mathbb{N}$ we have

$$\mathsf{E}\overline{n}\,\overline{m} \rhd\!\!\rhd \lambda y, x\,.\,\overline{n}\,\overline{m}yx \rhd\!\!\rhd \lambda y, x\,.\,\overline{m}^n yx$$

so that a reduction

$$\overline{m}^n yx \rhd\!\!\rhd\!\!\rhd y^{m^n}x$$

will suffices. We might try to prove this by induction (on n?), but things are not so simple.

Fix $m \in \mathbb{N}$ and a term t and we consider

$$[n, k] \quad (\forall s)[(\overline{m}^n t)^k s \rhd\!\!\rhd\!\!\rhd t^{m^n\times k}s]$$

as a property of a pair n, k of natural numbers. We will show

$$(\forall n, k)[n, k]$$

and then instantiate $t := y, s := x, k := 1$ to get the required result. We prove this result by a double induction over n, k. Notice that we have to allow the parameter s to vary in the induction. (We could have fixed t as y, but this way it looks a bit neater.)

Let

$$[n, \cdot] \text{ abbreviate } (\forall k)[n, k]$$

and observe that

$$[n, 0] \qquad [0, 1]$$

hold trivially. Two simple calculations give

$$\left.\begin{matrix}[n, k] \\ [n, 1]\end{matrix}\right\} \Longrightarrow [n, k'] \qquad [n, \cdot] \Longrightarrow [n', 1]$$

and it is the first of these that requires the variation of s.

We combine these two implications into the double induction. From the first and $[n, 0]$ we have

$$(\star) \quad [n, 1] \Longrightarrow [n, \cdot]$$

and hence

$$[n, 1] \Longrightarrow [n', 1]$$

using the second. Thus $[0, 1]$ gives

$$(\forall n)[n, 1]$$

and hence

$$(\forall n, k)[n, k]$$

follows by a second use of $(\star)$. ■

2.2 Let

$$\overline{N} = \lambda u \,.\, u(\lambda v \,.\, \overline{0})\overline{1}$$

so that

$$\overline{N}\,\overline{0} \rhd\!\rhd (\lambda v \,.\, \overline{0})^0\overline{1} = \overline{1} \qquad \overline{N}\,\overline{m+1} \rhd\!\rhd (\lambda v \,.\, \overline{0})^{m+1}\overline{1} \rhd\!\rhd (\lambda v \,.\, \overline{0})((--)^m\overline{1}) \rhd\!\rhd \overline{0}$$

to give the required result. ■

2.3 Let

$$\overline{D} = \lambda x, y \,.\, y\overline{d}x$$

where $\overline{d}$ is the term of Example 2.5 and Lemmas 2.6 and 2.7. For each $m, n \in \mathbb{N}$ we have

$$\overline{D}\overline{m}\,\overline{n} \rhd\!\rhd \overline{d}^n\overline{m} \rhd\!\rhd \overline{m-n}$$

where the last reduction follows by a simple induction over n. ■

A.2.2 For section 2.2

2.4 The two functions $\aleph$ and $\beth$ are essentially the same. More precisely, we have

$$\aleph(z, y, x) = \beth(x, y, z)$$

for all $x, y, z \in \mathbb{N}$. To see this fix y, z and consider

$$[x] \quad (\forall u)[\aleph(\beth(u, y, z), y, x) = \beth(x + u, y, x)]$$

as a property of $x \in \mathbb{N}$. We verify $\forall x)[x]$ and then take the particular case $u = 0$. We proceed by induction over x and for this we need to allow the parameter u to vary.

The base case, $x = 0$, is immediate.

For the induction step, $x \mapsto x'$, we have

$$\begin{aligned} \aleph(\beth(u, y, z), y, x') &= \aleph(u^{\beth(u,y,z)}, y, x) \\ &= \aleph(\beth(u', y, z), y, x) \\ &= \beth(x + u', y, z) \qquad = \beth(x' + u, y, z) \end{aligned}$$

as required. Here the third equality uses the induction hypothesis. ■

2.5 Consider the terms

$$\mathsf{F} = \lambda u, v, w\,.\,\mathsf{E}w(\mathsf{A}vu) \qquad \mathsf{G} = \lambda u, v, w\,.\,\mathsf{E}(\mathsf{A}wv)u \qquad \mathsf{H} = \lambda u, v, w\,.\,\mathsf{E}(\mathsf{M}wv)u$$

where $\mathsf{A}, \mathsf{M}, \mathsf{E}$ are the terms of Definition 2.3. For each $l, m, n \in \mathbb{N}$ we have

$$\begin{aligned} \mathsf{F}\overline{l}\,\overline{m}\,\overline{n} &\rhd\!\rhd \mathsf{E}\overline{n}(\mathsf{A}\overline{m}\,\overline{l}) \rhd\!\rhd \overline{(l+m)^n} \\ \mathsf{G}\overline{l}\,\overline{m}\,\overline{n} &\rhd\!\rhd \mathsf{E}(\mathsf{A}\overline{n}\,\overline{m})\overline{l} \rhd\!\rhd \overline{l^{m+n}} \\ \mathsf{H}\overline{l}\,\overline{m}\,\overline{n} &\rhd\!\rhd \mathsf{E}(\mathsf{M}\overline{n}\,\overline{m})\overline{l} \rhd\!\rhd \overline{l^{m\times n}} \end{aligned}$$

to show that $\mathsf{F}, \mathsf{G}, \mathsf{H}$ provide the required simulations.

These terms are not normal and can be reduced. Thus

$$\begin{aligned} \mathsf{F} &\rhd\!\rhd \lambda u, v, w, y, x\,.\,w(\mathsf{A}vu)yx && \rhd\!\rhd \lambda u, v, w, y, x\,.\,w(\lambda y, z\,.\,(vy)(uyx))yx \\ \mathsf{G} &\rhd\!\rhd \lambda u, v, w, y, x\,.\,(\mathsf{A}wv)uyx \\ &\rhd\!\rhd \lambda u, v, w, y, x\,.\,((\lambda y, x\,.\,(wy)(vyx))uyx) \\ &\rhd\!\rhd \lambda u, v, w, y, x\,.\,((\lambda x\,.\,(wu)(vux))yx) && \rhd\!\rhd \lambda u, v, w, y, x\,.\,(wu)(vuy)x \\ \mathsf{H} &\rhd\!\rhd \lambda u, v, w, y, x\,.\,(\mathsf{M}wv)uyx \\ &\rhd\!\rhd \lambda u, v, w, y, x\,.\,(\lambda y, x\,.\,w(vy)x)uyx \\ &\rhd\!\rhd \lambda u, v, w, y, x\,.\,(\lambda x\,.\,w(vu)x)yx && \rhd\!\rhd \lambda u, v, w, y, x\,.\,w(vu)yx \end{aligned}$$

to produce three normal, but unreadable, terms. (There are some fools in this world who believe that normal terms are in some sense 'better'.) ■

2.6 Both these terms $\mathsf{B}_1, \mathsf{B}_2$ simulate the stacking functions, The term B_1 is just the term B of Lemma 2.10. Let's look at B_2.

For $l, m, n \in \mathbb{N}$ we have

$$\mathsf{B}_2\overline{l}\,\overline{m}\,\overline{n} \rhd\!\rhd \lambda y, x\,.\,M^l\overline{n}yx \quad \text{where} \quad M = (\lambda u\,.\,u\overline{m})$$

and hence a reduction

$$M^l\overline{n}yx \rhd\!\rhd y^{\beth(l,m,n)}x$$

will give the required result. This is proved by induction on l. The induction step is something like

$$M^{l+1}\overline{n}yx = M(M^l\overline{n})yx \rhd\!\rhd M^l\overline{n}\,\overline{m}yx \rhd\!\rhd \overline{m}^{\beth(l,m,n)}yx$$

which will lead to the required result. Here the last reduction uses the induction hypothesis, but notice that this requires a variation of a parameter. ■

A.2.3 For section 2.3

2.7 Since

$$D(x, dx) = (x', x)$$

the first result

$$D(x, dx) = (x', dx')$$

follows since $dx' = x$.

With this

$$D^x(0, 0) = (x, dx)$$

follws by induction on x. For the base case, $x = 0$, we have

$$D^0(0, 0) = (0, 0) = (0, d0)$$

since $d0 = 0$. For the induction step, $x \mapsto x'$, we have

$$D^{x+1}(0, 0) = D(D^x(0, 0)) = D(x, dx) = (x', dx')$$

as required. Here the second equality uses the induction hypothesis, and the third uses the observation above.

To calculate dx we manipulate two registers, both of which hold a natural number. Starting from the pair $(0, 0)$ we repeatedly update the pair by hitting it with D. We continue until the first component reaches the required input x. At that point the second component holds the required value dx. ■

2.8 (a) For terms l, r we have

$$\begin{aligned}\mathsf{L}(\mathsf{P}lr) &\rhd\!\rhd \mathsf{P}lr\mathsf{t} \rhd\!\rhd \mathsf{t}lr \rhd\!\rhd l\\ \mathsf{R}(\mathsf{P}lr) &\rhd\!\rhd \mathsf{P}lr\mathsf{f} \rhd\!\rhd \mathsf{f}lr \rhd\!\rhd r\end{aligned}$$

to give the required result.

(b) For terms l, r we have

$$\begin{aligned}\mathsf{L}(\mathsf{P}lr) &\rhd\!\rhd \mathsf{P}lr\overline{0} \rhd\!\rhd \lambda y, x\,.\,\overline{0}(-)(lyx) &&\rhd\!\rhd \lambda y, x\,.\,lyx\\ \mathsf{R}(\mathsf{P}lr) &\rhd\!\rhd \mathsf{P}lr\overline{1} \rhd\!\rhd \lambda y, x\,.\,\overline{1}(\lambda u\,.\,ryx)(-) &&\rhd\!\rhd \lambda y, x\,.\,ryx\end{aligned}$$

to show that these do not satisfy the requirements of Definition 2.17.

However, for numerals we have

$$\mathsf{L}(\mathsf{P}\overline{l}\overline{r}) \rhd\!\rhd \lambda y, x\,.\,\overline{l}yx \rhd\!\rhd \lambda y, x\,.\,y^l x = \overline{l}$$
$$\mathsf{R}(\mathsf{P}\overline{l}\overline{r}) \rhd\!\rhd \lambda y, x\,.\,\overline{r}yx \rhd\!\rhd \lambda y, x\,.\,y^r x = \overline{r}$$

to show that these terms form 'numeralwise' pairing gadgets. Often these are good enough to do a certain job. ■

2.9 Consider $m, n \in \mathbb{N}$ and let $W = \overline{(m,n)}$. The pairing reductions give

$$\mathsf{L}W = \mathsf{L}(\mathsf{P}\overline{m}\,\overline{n}) \rhd\!\rhd \overline{m}$$

so that

$$\mathsf{D}\overline{(m,n)} = \mathsf{D}W \rhd\!\rhd \mathsf{P}(\mathsf{S}(\mathsf{L}W))(\mathsf{L}W) \rhd\!\rhd \mathsf{P}(\mathsf{S}\overline{m})(\overline{m}) \rhd\!\rhd \mathsf{P}\overline{m'}\overline{m} = \overline{(m',m)}$$

as required.

A simple induction show that

$$\mathsf{D}^m\overline{(0,0)} \rhd\!\rhd \overline{(m,dm)}$$

for each m, and hence

$$\mathsf{d}\overline{m} \rhd\!\rhd \mathsf{R}(\overline{m}\mathsf{D}\overline{(0,0)}) \rhd\!\rhd \mathsf{R}(\mathsf{D}^m\overline{(0,0)}) \rhd\!\rhd \mathsf{R}\overline{(m,dm)} \rhd\!\rhd \overline{dm}$$

to show that d simulates d. ■

2.10 We use the 'numeralwise' pairing gadgets of Exercise 2.8(b) to form the terms D, d. With these we have

$$\overline{(0,0)} = \mathsf{P}\overline{0}\,\overline{0} \rhd\!\rhd \lambda z, y, x\,.\,(\lambda u\,.\,\overline{0}yx)(\overline{0}yx) \rhd\!\rhd \lambda z, y, x\,.\,(\lambda u\,.\,x)x = (-2-)$$

and

$$\begin{aligned}\mathsf{D} = \lambda w\,.\,\mathsf{P}(\mathsf{S}(\mathsf{L}w))(\mathsf{L}w) &\rhd\!\rhd \lambda w\,.\,\mathsf{P}(\mathsf{S}(w\overline{0}))(w\overline{0}) \\ &\rhd\!\rhd \lambda w, z, y, x\,.\,z(\lambda u\,.\,w\overline{0}yx)(\mathsf{S}(w\overline{0})yx) \\ &\rhd\!\rhd \lambda w, z, y, x\,.\,z(\lambda u\,.\,W)(yW) \qquad = (-3-)\end{aligned}$$

as required.

With these we have

$$\mathsf{d} = \lambda x\,.\,\mathsf{R}(x\mathsf{D}\overline{(0,0)}) \rhd\!\rhd \lambda x\,.\,x\mathsf{D}\overline{(0,0)}\overline{1} \rhd\!\rhd \lambda x\,.\,x(-3-)(-2-)\overline{1} = \overline{d}$$

and hence (by the confluence property) the two terms d and $\overline{d}$ simulate the same function, namely d. ■

A.2.4 For section 2.4

2.11 For $x, y \in \mathbb{N}$ we have

$$y = \lfloor \log x \rfloor \iff y \leq \log x < y+1 \iff 2^y \leq x < 2^{y+1}$$

and hence

$$fx = (\mu y)[x < 2^{y+1}]$$

is the required function. Note also that

$$x < z \iff x + 1 \leq z \iff (x + 1) - z = 0$$

and so the function f can be obtained by an unbounded search applied to a quite simple function. That search can be bounded (since $fx \leq x$) which enables us to show that $f \in \mathcal{PR}$. In fact, f is Kalmár elementary, but that should be dealt with elsewhere. ■

A.3 For chapter 3

A.3.1 For section 3.1

3.1 For each typed term t the erasure t^ϵ is generated by recursion over t where

$$x^\epsilon = x \quad (qp)^\epsilon = (q^\epsilon)(p^\epsilon) \quad (\lambda x : \sigma \,.\, r)^\epsilon = (\lambda x \,.\, r)$$

are the appropriate clauses. ■

3.3 The terms $\mathsf{S}_\zeta, \mathsf{A}_\zeta, \mathsf{M}_\zeta$ of Example 3.10 are well-typed versions of $\mathsf{S}, \mathsf{A}, \mathsf{M}$, respectively.

The term E_J of that example is not well-typed. The term E_ζ of Lemma 3.15 is a well-typed version of E.

For an arbitrary type ζ let

$$\eta = \zeta''' \to \zeta''$$

and set

$$\mathsf{B}_\zeta = \lambda u : \eta', v : \zeta'', w : \zeta''' \,.\, uVw \quad \text{where} \quad V = \lambda z : \zeta''', y : \zeta', x : \zeta \,.\, zvyx$$

is the auxiliary term. It is routine to derive

$$\vdash \mathsf{B}_\zeta : \eta' \to \zeta'' \to \zeta''' \to \zeta''$$

but you should notice something odd about this type. It will come to haunt you later.

A typed version of the term $\overline{d}$ is dealt with in Lemma 3.17. ■

A.3.2 For section 3.2

3.4 Each of these and similar results are proved by induction over the derivations involved, or sometimes over the height of the derivations involved. The base cases of the induction deal with a Leaf, which in this case is a Projection. Each induction step deals with one of the derivation rules, which in this case is a Weakening, a use of Introduction, or an Elimination. Let's look at the pertinent bits of each of these proofs.

(a) For the base case we observe that, by the proviso, for each Leaf the context must be legal.

We deal with the three induction steps in turn (in reverse order).

A use of Elimination does not alter the context, so the legality is not altered.

A use of Introduction shortens the context and this preserves legality.

Consider a use of Weakening

$$\frac{\Gamma \vdash t : \tau}{\Gamma, x : \sigma \vdash t : \tau}$$

where, by the induction hypothesis, we know that Γ is legal. However, by the proviso, the identifier must be fresh, that is it must not occur in Γ. But this ensures that extended context $\Gamma, x : \sigma$ is legal.

(b) The base case and the induction steps across Weakening and Introduction are straight forward.

Consider a use of Introduction

$$\frac{\Gamma, x : \sigma \vdash r : \rho}{\Gamma \vdash t : \tau}$$

where

$$t = (\lambda x : \sigma \,.\, r) \qquad \tau = (\sigma \to \rho)$$

are the root subject and predicate. The induction hypothesis gives

$$\partial r \subseteq \partial\Gamma \cup \{x\}$$

and we know $x \notin \partial\Gamma$ by part (a). But now

$$\partial t = \partial r - \{x\} \subseteq (\partial\Gamma \cup \{x\}) - \{x\} = \partial\Gamma$$

as required.

(c) This is not quite as straight foprward as it looks. Before we get into the proof we make an observation.

Consider a derivation of

$$(\nabla) \quad \Gamma \vdash t : \tau$$

and think what the root rule can be. It must be either a use of Weakening or the rule attached to the subject t according to the following table.

t	Rule
x	Projection
qp	Elimination
$\lambda x : \sigma \,.\, r$	Introduction

Thus any derivation ∇, as above, arises from a shorter or equal derivation

$$(\nabla^-) \quad \Gamma^- \vdash t : \tau$$

where the root rule of this is the one attached to t. Here Γ^- is some initial part of Γ. This derivation ∇^- is obtained from ∇ by unravelling all the root uses of Weakening. In other words ∇ is obtained from ∇^- by a series of Weakenings.

Let's now look at the proof of the result. We have two derivations

$$(\nabla_1) \quad \Gamma \vdash t : \tau_1 \qquad (\nabla_2) \quad \Gamma \vdash t : \tau_2$$

in the same context and with the same subject. To show $\tau_1 = \tau_2$ we unravel both derivations to obtain

$$(\nabla_1^-) \quad \Gamma_1 \vdash t : \tau_1 \qquad (\nabla_2^-) \quad \Gamma_2 \vdash t : \tau_2$$

where each of Γ_1, Γ_2 is an initial part of Γ, and the root rule of neither derivation is Weakening. By the observation above the root rule of each of ∇_1^-, ∇_2^- is the same, namely that determined by t. We can now proceed by induction using one base case and two induction steps.

For the induction step across $t = qp$ we have two derivations

$$\frac{\Gamma_1 \vdash q : \pi_1 \to \tau_1 \quad \Gamma_1 \vdash p : \pi_1}{\Gamma_1 \vdash t : \tau_1} \qquad \frac{\Gamma_2 \vdash q : \pi_2 \to \tau_2 \quad \Gamma_2 \vdash p : \pi_2}{\Gamma_2 \vdash t : \tau_1}$$

using auxiliary types π_1, π_2. Each of Γ_1 and Γ_2 is an inital part of Γ so we may re-instate the Weakenings to obtain derivations

$$\Gamma_1 \vdash q : \pi_1 \to \tau_1 \quad \Gamma_1 \vdash p : \pi_1 \qquad \Gamma_2 \vdash q : \pi_2 \to \tau_2 \quad \Gamma_2 \vdash p : \pi_2$$

each of which is strictly shorter than the longer of the two original derivations. The induction hypothesis gives

$$(\pi_1 \to \tau_1) = (\pi_2 \to \tau_2) \qquad \pi_1 = \pi_2 = \pi \quad \text{(say)}$$

so that

$$(\pi \to \tau_1) = (\pi \to \tau_2)$$

to give $\tau_1 = \tau_2$, as required.

For the induction step across $t = (\lambda x : \sigma \,.\, r)$ we have two derivations

$$\frac{\dfrac{\Gamma_1, x : \sigma \vdash r : \rho_1}{\Gamma_1 \vdash t : \tau_1}}{\Gamma \vdash t : \tau_1} \qquad \frac{\dfrac{\Gamma_2, x : \sigma \vdash r : \rho_2}{\Gamma_2 \vdash t : \tau_2}}{\Gamma \vdash t : \tau_2}$$

where

$$\tau_1 = (\sigma \longrightarrow \rho_1) \qquad \tau_2 = (\sigma \longrightarrow \rho_2)$$

are the two root types. Here Γ_1, Γ_2 are initial parts of Γ and the dots indicate a series of Weakenings. We can now apply the induction hypothesis to the top derivations. [Can we really!] This gives

$$\rho_1 = \rho_2$$

and hence $\tau_1 = \tau_2$, as required.

In this proof we have been a bit relaxed over the the precise nature of the induction hypothesis. You may like to work out what it should be. ■

3.5 The term E_J of that example is not well-typed. The term E_ζ of Lemma 3.15 is a well-typed version of E. ■

3.6 Consider the compound xx which occurs embeddded in the component Ω of the term Y. If Y can be typed then somewhere within the witnessing derivation we must have

$$\frac{\Gamma \vdash x : \pi \quad \Gamma \vdash x : \pi \to \tau}{\Gamma \vdash xx : \tau}$$

for some types π and τ. But now, by Exercise 3.4(c) we must have

$$\pi = (\pi \to \tau)$$

which is impossible.

A similar kind of argument will show that any parrticular fixed point combinator can not be typed. ■

3.7 The derivation

$$\dfrac{\dfrac{\Gamma \vdash g : (\theta \to \zeta)}{\Gamma, x : \eta \vdash g : (\theta \to \zeta)} \qquad \dfrac{\dfrac{\Gamma \vdash f : (\eta \to \theta)}{\Gamma, x : \eta \vdash f : (\eta \to \theta)} \quad \Gamma, x : \eta \vdash x : \eta}{\Gamma, x : \eta \vdash fx : \theta}}{\dfrac{\Gamma, x : \eta \vdash g(fx) : \zeta}{\Gamma \vdash g \circ f : (\eta \to \zeta)}}$$

gives the result. ■

A.3.3 For section 3.3

3.9 Let

$$\Gamma = u : \eta', v : \zeta'', w : \zeta''' \qquad \Sigma = \Gamma, z : \zeta''', y : \zeta', x : \zeta$$

where we assume the idenitifiers u, v, w, z, y, x are distinct. A derivation of the form

$$\dfrac{\Gamma \vdash u : \eta' \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \vdash z : \zeta''' \quad \Sigma \vdash v : \zeta''}{\Sigma \vdash zv : \zeta''} \qquad \Sigma \vdash y : \zeta'}{\Sigma \vdash zvy : \zeta'} \qquad \Sigma \vdash x : \zeta}{\Sigma \vdash zvyx : \zeta}}{\Gamma, z : \zeta''', y : \zeta' \vdash \star : \zeta'}}{\Gamma, z : \zeta''' \vdash \star : \zeta''} \qquad \Gamma \vdash V : \eta}{\Gamma \vdash uV : \eta} \qquad \Gamma \vdash w : \zeta'''}{\dfrac{\dfrac{\dfrac{\Gamma \vdash uVw : \zeta''}{u : \eta', v : \zeta'' \vdash \star : \zeta''' \to \zeta''}}{u : \eta' \vdash \star : \zeta'' \to \zeta''' \to \zeta''}}{\vdash \mathsf{B}_\zeta : \eta' \to \zeta'' \to \zeta''' \to \zeta''}}$$

shows that B_ζ is well-formed with

$$\eta' \to \zeta'' \to \zeta''' \to \zeta''$$

as its type.

This term looks as through it could simulate a function. But observe that the left hand input type η' can *not* house numerals. The term does not simulate a function. ■

3.10 The typing of L_ζ is straight forward.

For the second part we first show

$$(\mathsf{J}_\zeta y)^m(\mathsf{K}_\zeta x)x \rhd\!\rhd y^m x$$

by induction on m. The base case, $m = 0$, is immediate. For the induction step $m \mapsto m'$, we have

$$(\mathsf{J}_\zeta y)^{m+1}(\mathsf{K}_\zeta x)x = (\mathsf{J}_\zeta y)\big((\mathsf{J}_\zeta y)^m(\mathsf{K}_\zeta x)\big)x \rhd\!\rhd y\big((\mathsf{J}_\zeta y)^m(\mathsf{K}_\zeta x)x\big) \rhd\!\rhd y\big(y^m x\big) = y^{m+1}x$$

to give the required result. Here the third step (second reduction) uses the induction hypothesis.

Using this, for each $m \in \mathbb{N}$ we have

$$\begin{aligned} \mathsf{L}_\zeta\overline{m_{\zeta'}} &\rhd\!\rhd \lambda y : \zeta, x : \zeta \,.\, \overline{m_{\zeta'}}(\mathsf{J}_\zeta y)(\mathsf{K}_\zeta x)x \\ &\rhd\!\rhd \lambda y : \zeta, x : \zeta \,.\, (\mathsf{J}_\zeta y)^m(\mathsf{K}_\zeta x)x \\ &\rhd\!\rhd \lambda y : \zeta, x : \zeta \,.\, y^m x \qquad\qquad = \overline{m_\zeta} \end{aligned}$$

to show that L_ζ converts numerals in the required fashion. ■

3.11 Recall that

$$\beth(0, m, n) = n \qquad \beth(l+1, m, n) = n^{\beth(l,m,n)}$$

for all $l, m, n \in \mathbb{N}$. The function $\beth(0, \cdot, \cdot)$ is easy to simulate (since it is a projection). Once we have a simulation of $\beth(l, \cdot, \cdot)$ we can combine this with a simulation of exponentiation to obtain a simulation of $\beth(l+1, \cdot, \cdot)$. However, some of the details need to be organized carefully.

In the following it is important that we have and construct simulations *throughout* $\boldsymbol{\lambda}1$. Several of the constructions proceed by recursion on the level l with allowable variation of a type ζ. The corresponding proofs proceed by induction over l.

For each type ζ we have a term

$$\mathsf{E}_\zeta : \zeta''' \to \zeta'' \to \zeta''$$

with

$$\mathsf{E}_\zeta\overline{n_{\zeta'}}\,\overline{m_\zeta} \rhd\!\rhd \overline{(m^n)_\zeta}$$

for all $m, n \in \mathbb{N}$.

We need to extend the numeral converter

$$\mathsf{L}_\zeta : \zeta''' \to \zeta''$$

of Exercise 3.10 to achieve a greater drop in level. Thus we set

$$\mathsf{L}^0_\zeta = \mathsf{L}_\zeta \qquad \mathsf{L}^{l+1}_\zeta = \mathsf{L}^l_\zeta \circ \mathsf{L}_{\zeta^{(l+1)}}$$

for each l (with variation of ζ). We may check that

$$\mathsf{L}^l_\zeta : \zeta^{(l+3)} \to \zeta''$$

and

$$\mathsf{L}^l_\zeta\overline{m_{\zeta^{(l+1)}}} \rhd\!\rhd \overline{m_\zeta}$$

for each $m \in \mathbb{N}$.

Finally we set

$$\mathsf{B}^0_\zeta = \lambda v, u : \zeta'' \,.\, v \qquad \mathsf{B}^{l+1}_\zeta = \lambda v, u : \zeta^{(l+3)} \,.\, \mathsf{E}_\zeta(\mathsf{B}^l_{\zeta'}vu)(\mathsf{L}^l_\zeta u)$$

to obtain

$$\mathsf{B}^l_\zeta : \zeta^{(l+2)} \to \zeta^{(l+2)} \to \zeta''$$

with

$$\mathsf{B}^l_\zeta \overline{n_{\zeta^{(l)}}}\, \overline{m_{\zeta^{(l)}}} \rhd\!\rhd \overline{\exists(l,m,n)_\zeta}$$

for all $l, m, n \in \mathbb{N}$. ■

A.3.4 For section 3.4

3.12 We show that the terms

$$\tilde{\mathsf{P}} = \lambda u : \sigma, v : \rho \,.\, \mathsf{P}uv \qquad \tilde{\mathsf{L}} = \lambda w : \theta \,.\, \mathsf{L}w \qquad \tilde{\mathsf{R}} = \lambda w : \theta \,.\, \mathsf{R}w$$

will do.

The typing requirements

$$\tilde{\mathsf{P}} : \sigma \to \rho \to \theta \qquad \tilde{\mathsf{L}} : \theta \to \sigma \qquad \tilde{\mathsf{R}} : \theta \to \rho$$

are immediate.

For terms l, r we have

$$\tilde{\mathsf{L}}(\tilde{\mathsf{P}}lr) \rhd\!\rhd \tilde{\mathsf{L}}(\mathsf{P}lr) \rhd\!\rhd \mathsf{L}(\mathsf{P}lr) \rhd\!\rhd l$$

to give one of the reduction requirements. The other one is obtained in a similar fashion.

Next we have

$$\tilde{\mathsf{L}} \circ \mathsf{Id} = \lambda w : \theta \,.\, \tilde{\mathsf{L}}(\mathsf{Id}w) \rhd\!\rhd \lambda w : \theta \,.\, \mathsf{L}(\mathsf{Id}w) \rhd\!\rhd \lambda w : \theta \,.\, \mathsf{L}(\mathsf{P}(\mathsf{L}w)(\mathsf{R}w)) \rhd\!\rhd \lambda w : \theta \,.\, \mathsf{L}w = \tilde{\mathsf{L}}$$

to give the first of the extra reduction requirements. The second one is obtained in a similar fashion. Finally we have

$$\tilde{\mathsf{P}} \circ (\tilde{\mathsf{L}}, \tilde{\mathsf{R}}) = \lambda w : \theta \,.\, \tilde{\mathsf{P}}(\tilde{\mathsf{L}}w)(\tilde{\mathsf{R}}w) \rhd\!\rhd \lambda w : \theta \,.\, \tilde{\mathsf{P}}(\mathsf{L}w)(\mathsf{R}w) \rhd\!\rhd \lambda w : \theta \,.\, \mathsf{P}(\mathsf{L}w)(\mathsf{R}w) = \mathsf{Id}$$

to give the third extra requirement. ■

A.4 For chapter 4

A.4.1 For section 4.1

[*No exercises here*]

A.4.2 For section 4.2

4.3 For each type ζ we have the iterator

$$\mathsf{I}_\zeta : \mathcal{N} \to \zeta''$$

and the term

$$\mathsf{L}_\zeta = \lambda u : \mathcal{N}, y : \zeta', x : \zeta \,.\, \mathsf{I}_\zeta uyx$$

satisfies

$$\vdash \mathsf{L}_\zeta : (\mathcal{N} \to \zeta'') \qquad \mathsf{L}_\zeta \ulcorner m \urcorner \rhd\!\rhd \overline{m_\zeta}$$

for each $m \in \mathbb{N}$. Thus L_ζ converts hones numerals into church numerals.

Using the given simulation

$$\overline{f} : \zeta(l)'' \to \cdots \to \zeta(1)'' \to \mathcal{N}''$$

of the l-placed function f let

$$\ulcorner f \urcorner = \lambda u_l, \cdots, u_l : \mathcal{N} \,.\, \overline{f}(\mathsf{L}_{\zeta(l)} u_l) \cdots (\mathsf{L}_{\zeta(1)} u_1)$$

to obtain a term

$$\vdash \ulcorner f \urcorner : \mathcal{N} \to \cdots \to \mathcal{N} \to \mathcal{N}''$$

which represents f.

We can do a similar trick with the target of a function provided we can find a term of type

$$\zeta'' \to \mathcal{N}$$

which converts church numerals into honest numerals. This can be done but is a little more complicated. ■

A.4.3 For section 4.3

4.4 This is one of the case where it is neater to have the recursion variable in the left-most position. However, we will stick with the given notation.

It is tempting to try to use the trick of Example 4.13, but that won't work precisely because the parameter y is allowed to vary. We must think of the function consuming the parameter last rather than first.

Let's look at the informal method and the formalize it.

We modify the data function h to produce a function

$$H : \mathbb{N} \times \mathbb{N}^{\mathbb{N}} \longrightarrow \mathbb{N} \times \mathbb{N}^{\mathbb{N}}$$

which can be iterated. Thus for $x \in \mathbb{N}, \phi \in \mathbb{N}^{\mathbb{N}}$ we set

$$\begin{gathered} H(x, \phi) = (x', \phi') \\ \text{where } \phi' y = h(y, x, \phi w) \\ \text{where } w = k(y, x) \end{gathered}$$

(for $y \in \mathbb{N}$). In particular if

$$\phi = f(\cdot, x)$$

then

$$\phi' y = h(y, x, f(w, x)) = f(y, x')$$

since $w = k(y, x)$. Thus

$$H(x, f(\cdot, x)) = (x', f(\cdot, x'))$$

(which indicates why it would be neater to have the recursion variable on the left). A simple induiction now gives

$$H^x(0, g) = (x, f(\cdot, x))$$

using the other data function.

In other words we have

$$f(y,x) = \mathrm{Right}(I_\bullet x H(\mathrm{Pair}0g))y$$

using informal pairing gadgets an an appropriate iterator. The function H is defined by

$$Hz = \mathrm{Pair}lr$$

where

$$l = S(\mathrm{Left}z) \qquad ry = h(y, \mathrm{Left}z, \mathrm{Right}zw) \text{ where } w = k(y, \mathrm{Left}z)$$

for each $y \in \mathbb{N}$.

We now convert these into λ-terms. We set

$$\begin{aligned} \ulcorner H \urcorner &= \lambda z : \mathcal{N} \times \mathcal{N}' \,.\, \mathsf{Pair}lr \\ \text{where } l &= \mathsf{S}(\mathsf{Left}z) \\ r &= \lambda y : \mathcal{N} \,.\, \ulcorner h \urcorner y(\mathsf{Left}z)(\mathsf{Right}zw) \\ \text{where } w &= \ulcorner k \urcorner y(\mathsf{Left}z) \end{aligned}$$

and then

$$\ulcorner f \urcorner = \lambda y, x : \mathcal{N} \,.\, \mathsf{Right}(\mathsf{I}_\bullet x \ulcorner H \urcorner (\mathsf{Pair}0 \ulcorner g \urcorner))y$$

is the required representation. Here $\mathsf{I}_\bullet = \mathsf{I}_{\mathcal{N} \times \mathcal{N}'}$. ■

4.5 For the proof of Lemma 4.17 consider the property

$$[i,r] \quad (\forall x)[(i,r,x) = (Ack^i f)^{r+1} x]$$

for $i, r \in \mathbb{N}$. Let

$$[i,\cdot] \text{ abbreviate } (\forall r)[i,r]$$

so we want $(\forall i)[i,\cdot]$. We observe that the three clauses of the specification of F give

$$(0) \quad [0,0] \qquad (1) \quad [i,\cdot] \Longrightarrow [i',0] \qquad (2) \quad [i,0] \& [i,r] \Longrightarrow [i,r']$$

for $i, r \in \mathbb{N}$. These are verified by direct calculation.

Now (2) gives

$$(3) \quad [i,0] \Longrightarrow [i,\cdot]$$

by induction on r, and this gives

$$(4) \quad [i,0] \Longrightarrow [i',0]$$

when combined with (2). But now (0) and (4) give

$$(\forall i)[i,0]$$

by an induction on i, and this with (3) gives ther required result.

For the reduction of Lemma 4.19 consider the property

$$[i,r] \quad (\forall m)[(\ulcorner Ack \urcorner^i \ulcorner f \urcorner)^{r+1} \ulcorner m \urcorner \rhd\!\rhd \ulcorner (Ack^i f)^{r+1} m \urcorner]$$

for $i, r \in \mathbb{N}$. As before let

$$[i,\cdot] \text{ abbreviate } (\forall r)[i,r]$$

so we want $(\forall i)[i,\cdot]$. Again as before we first check that

$$(0)\quad [0,0] \qquad (1)\quad [i,\cdot] \Longrightarrow [i',0] \qquad (2)\quad [i,0]\&[i,r] \Longrightarrow [i,r']$$

hold for $i, r \in \mathbb{N}$. However, here some of these take a little more work.

Clause (0) is the given reduction property of $\ulcorner f \urcorner$.

For both (1) and (2) it is covenient to let

$$\widetilde{g} = \ulcorner Ack \urcorner^i \ulcorner f \urcorner \qquad g = Ack^i f$$

for the fixed i.

For (1) the induction hypothesis gives

$$\widetilde{g}^{r+1} \ulcorner m \urcorner \rhd\!\rhd \ulcorner g^{r+1} m \urcorner$$

for all $r, m \in \mathbb{N}$. With this we have

$$\begin{array}{rcll} (\ulcorner Ack \urcorner^{i+1} \ulcorner f \urcorner) \ulcorner m \urcorner & = & \ulcorner Ack \urcorner \widetilde{g}\, \ulcorner m \urcorner & \\ & \rhd\!\rhd & \mathsf{I}_{\mathcal{N}}(\mathsf{S} \ulcorner m \urcorner) \widetilde{g} \ulcorner m \urcorner & \\ & \rhd\!\rhd & \ulcorner g \urcorner^{m+1} \ulcorner m \urcorner & \\ & \rhd\!\rhd & \ulcorner g^{m+1} m \urcorner & \\ & = & \ulcorner Ack\, g\, m \urcorner \qquad = & \ulcorner Ack^{i+1} f m \urcorner \end{array}$$

as required. Here the fourth step (the third reduction) uses the induction hypothesis as give above.

For (2) the induction hypothesis gives

$$\widetilde{g} \ulcorner m \urcorner \rhd\!\rhd \ulcorner g m \urcorner \qquad \widetilde{g}^{r+1} \ulcorner m \urcorner \rhd\!\rhd \ulcorner g^{r+1} m \urcorner$$

for arbitrary $m \in \mathbb{N}$ and the fixed $r \in \mathbb{N}$. With these we have

$$\begin{array}{rcll} (\ulcorner Ack \urcorner^i \ulcorner f \urcorner)^{r+2} \ulcorner m \urcorner & = & \widetilde{g}^{r+2} \ulcorner m \urcorner & \\ & = & \widetilde{g}(\widetilde{g}^{r+1}) \ulcorner m \urcorner & \\ & \rhd\!\rhd & \widetilde{g}(\ulcorner g^{r+1} m \urcorner & \\ & \rhd\!\rhd & \ulcorner g(g^{r+1} m) \urcorner & \\ & = & \ulcorner g^{r+2} m \urcorner \qquad = & \ulcorner (Ack^i f)^{r+2} m \urcorner \end{array}$$

as required. Here the two reductions use th etwo parts of the induction hypothesis.

This proves (0,1,2) and the proof of $(\forall i)[i,\cdot]$ now follows by the same argument as used in the first part. ■

4.6 (a) Consider the property

$$[i,x] \quad F(i,x) = Rob^i f x$$

for $i, x \in \mathbb{N}$. Let

$$[i,\cdot] \text{ abbreviate } (\forall x)[i,x]$$

so we want $(\forall i)[i,\cdot]$. We first obtain the three clauses

$$(0)\quad [0,\cdot] \qquad (1)\quad [i,1] \Longrightarrow [i',0] \qquad (2)\quad [i,\cdot]\&[i',x] \Longrightarrow [i',x']$$

for $i, x \in \mathbb{N}$.

Clause (0) is immediate. Clause (1) holds since

$$Rob f 0 = f 1$$

for all $f : \mathbb{N}'$. The proof of clause (2) is a bit more complicated.

First of all we have

$$F(i',x) = F(i,F(i',x)) = F(i,Rob^{i+1}fx) = (Rob^i f)(Rob^{i+1}fx)$$

using the hypotheses $[i',x]$ and $[i,\cdot]$ in that order. But now

$$Rob^{i+1}fx = Rob(Rob^i f)x = (Rob^i f)^{x+1}1$$

so that

$$(Rob^i f)(Rob^{i+1}fx) = (Rob^i f)((Rob^i f)^{x+1}1) = (Rob^i f)^{x+2}1 = Rob(Rob^i f)(x+1)$$

and hence

$$F(i',x) = (Rob^i f)(Rob^{i+1}fx) = Rob(Rob^i f)(x+1) = Rob^{i+1}f(x+1)$$

as required.

Now (2) gives

$$(3)\quad [i,\cdot]\&[i',0] \Longrightarrow [i',\cdot]$$

by an induction over x, and hence

$$(4)\quad [i,\cdot] \Longrightarrow [i',\cdot]$$

follows by (1). Thus an induction over i gives the requirted result.

(b) Using the iterator $\mathsf{I} = \mathsf{I}_{\mathcal{N}}$, the term

$$\ulcorner Rob \urcorner = \lambda y : \mathcal{N}', x : \mathcal{N} \,.\, \mathsf{I}(\mathsf{S}x)y\ulcorner 1\urcorner$$

will do.

(c) Using the iterator $\mathsf{J} = \mathsf{I}_{\mathcal{N}'}$, the term

$$\ulcorner F \urcorner = \lambda i, x : \mathcal{N} \,.\, \mathsf{J}i\ulcorner Rob\urcorner\ulcorner f\urcorner x$$

will do.

(d) We first show

$$[n]\quad (\forall m)[\ulcorner Rob\urcorner^n\ulcorner f\urcorner\ulcorner m\urcorner \rhd\!\rhd \ulcorner Rob^n f m\urcorner]$$

by induction on n. The base case [0] is just the given representation of f by $\ulcorner f\urcorner$. The induction step $[n] \Rightarrow [n+1]$ requires an auxiliary result. Assuming $[n]$ we show

$$\langle k\rangle\quad (\ulcorner Rob\urcorner^n\ulcorner f\urcorner)^k\ulcorner 1\urcorner \rhd\!\rhd \ulcorner (Rob^n f)^k 1\urcorner$$

by induction on k.

The base case $\langle 0\rangle$ is trivial. For the induction step $\langle k\rangle \Rightarrow \langle k+1\rangle$ with

$$m = (Rob^n f)^k 1$$

we have

$$\begin{array}{rcl} (\ulcorner Rob\urcorner^n\ulcorner f\urcorner)^{k+1}\ulcorner 1\urcorner & = & (\ulcorner Rob\urcorner^n\ulcorner f\urcorner)\big((\ulcorner Rob\urcorner^n\ulcorner f\urcorner)^k\ulcorner 1\urcorner\big) \\ & \rhd\!\rhd & \ulcorner Rob\urcorner^n\ulcorner f\urcorner\ulcorner m\urcorner \\ & \rhd\!\rhd & \ulcorner Rob^n f m\urcorner \\ & = & \ulcorner (Rob^n f)\big((Rob^n f)^k 1\big)\urcorner \qquad = \quad \ulcorner (Rob^n f)^{k+1}1\urcorner \end{array}$$

as required. Here the first reduction uses local induction hypothesis $\langle k\rangle$ and the second uses the global induction $[n]$.

With this result we have

$$\begin{array}{rcl} \ulcorner Rob\urcorner^{n+1}\ulcorner f\urcorner\ulcorner m\urcorner & = & \ulcorner Rob\urcorner(\ulcorner Rob\urcorner^n\ulcorner f\urcorner)\ulcorner m\urcorner \\ & \rhd\!\rhd & (\ulcorner Rob\urcorner^n\ulcorner f\urcorner)^{m+1}\ulcorner 1\urcorner \\ & \rhd\!\rhd & \ulcorner (Rob^n f)^{m+1}1\urcorner \\ & = & \ulcorner Rob(Rob^n f)m\urcorner \qquad = \quad \ulcorner Rob^{n+1} f m\urcorner \end{array}$$

to verify $[n+1]$. Here the first reduction uses the construction of Rob and the second uses the auxiliary result $\langle m+1\rangle$.

Finally, with $[n]$ we have

$$\ulcorner F\urcorner\ulcorner n\urcorner\ulcorner m\urcorner \rhd\!\rhd \mathsf{I}\ulcorner n\urcorner\ulcorner Rob\urcorner\ulcorner f\urcorner\ulcorner m\urcorner \rhd\!\rhd \ulcorner Rob\urcorner^n\ulcorner f\urcorner\ulcorner m\urcorner \rhd\!\rhd \ulcorner Rob^n f m\urcorner$$

as required to show that $\ulcorner F\urcorner$ represents F. ■

4.7 This is very like Solution 4.6 with a few minor changes.

(a) Consider the property

$$[i,x] \quad F(i,x) = Grz^i f x$$

for $i, x \in \mathbb{N}$. Let

$$[i,\cdot] \text{ abbreviate } (\forall x)[i,x]$$

so we want $(\forall i)[i,\cdot]$. We first obtain the three clauses

$$(0) \quad [0,\cdot] \qquad (1) \quad [i',0] \qquad (2) \quad [i,\cdot]\&[i',x] \Longrightarrow [i',x']$$

for $i, x \in \mathbb{N}$.

Clause (0) is immediate. Clause (1) holds since

$$Grz f 0 = f 2$$

for all $f : \mathbb{N}'$. The proof of clause (2) is a bit more complicated.

First of all we have

$$F(i',x) = F(i,F(i',x)) = F(i, Grz^{i+1} f x) = (Grz^i f)(Grz^{i+1} f x)$$

using the hypotheses $[i', x]$ and $[i, \cdot]$ in that order. But now

$$Grz^{i+1} f x = Grz(Grz^i f)x = (Grz^i f)^x 2$$

so that

$$(Grz^i f)(Grz^{i+1} f x) = (Grz^i f)((Grz^i f)^x 2) = (Grz^i f)^{x+1} 2 = Grz(Grz^i f)(x+1)$$

and hence

$$F(i', x) = (Grz^i f)(Grz^{i+1} f x) = Grz(Grz^i f)(x+1) = Grz^{i+1} f(x+1)$$

as required.

Now (1,2) gives

$$(3) \quad [i, \cdot] \Longrightarrow [i', \cdot]$$

by an induction over x, and hence so that (0) with an induction over i gives the required result.

(b) Using the iterator $\mathsf{I} = \mathsf{I}_{\mathcal{N}}$, the term

$$\ulcorner Grz \urcorner = \lambda y : \mathcal{N}', x : \mathcal{N} \,.\, \mathsf{I} x y \ulcorner 2 \urcorner$$

will do.

(c) Using the iterator $\mathsf{J} = \mathsf{I}_{\mathcal{N}'}$, the term

$$\ulcorner F \urcorner = \lambda i, x : \mathcal{N} \,.\, \mathsf{J} i \ulcorner Grz \urcorner \ulcorner f \urcorner x$$

will do.

(d) We first show

$$[n] \quad (\forall m)[\ulcorner Grz \urcorner^n \ulcorner f \urcorner \ulcorner m \urcorner \rhd\!\rhd \ulcorner Grz^n f m \urcorner]$$

by induction on n. The base case [0] is just the given representation of f by $\ulcorner f \urcorner$. The induction step $[n] \Rightarrow [n+1]$ requires an auxiliary result. Assuming $[n]$ we show

$$\langle k \rangle \quad (\ulcorner Grz \urcorner^n \ulcorner f \urcorner)^k \ulcorner 2 \urcorner \rhd\!\rhd \ulcorner (Grz^n f)^k 2 \urcorner$$

by induction on k.

The base case $\langle 0 \rangle$ is trivial. For the induction step $\langle k \rangle \Rightarrow \langle k+1 \rangle$ with

$$m = (Grz^n f)^k 2$$

we have

$$\begin{array}{lcll} (\ulcorner Grz \urcorner^n \ulcorner f \urcorner)^{k+1} \ulcorner 2 \urcorner & = & (\ulcorner Grz \urcorner^n \ulcorner f \urcorner)\big((\ulcorner Grz \urcorner^n \ulcorner f \urcorner)^k \ulcorner 2 \urcorner\big) & \\ & \rhd\!\rhd & \ulcorner Grz \urcorner^n \ulcorner f \urcorner \ulcorner m \urcorner & \\ & \rhd\!\rhd & \ulcorner Grz^n f m \urcorner & \\ & = & \ulcorner (Grz^n f)\big((Grz^n f)^k 2\big) \urcorner & = \quad \ulcorner (Grz^n f)^{k+1} 2 \urcorner \end{array}$$

as required. Here the first reduction uses local induction hypothesis $\langle k \rangle$ and the second uses the global induction $[n]$.

With this result we have

$$\begin{array}{rcll} \ulcorner Grz\urcorner^{n+1}\ulcorner f\urcorner\ulcorner m\urcorner & = & \ulcorner Grz\urcorner(\ulcorner Grz\urcorner^{n}\ulcorner f\urcorner)\ulcorner m\urcorner & \\ & \rhd\!\!\rhd & (\ulcorner Grz\urcorner^{n}\ulcorner f\urcorner)^{m+1}\ulcorner 2\urcorner & \\ & \rhd\!\!\rhd & \ulcorner (Grz^{n}f)^{m}2\urcorner & \\ & = & \ulcorner Grz(Grz^{n}f)m\urcorner & = \quad \ulcorner Grz^{n+1}fm\urcorner \end{array}$$

to verify $[n+1]$. Here the first reduction uses the construction of Grz and the second uses the auxiliary result $\langle m\rangle$.

Finally, with $[n]$ we have

$$\ulcorner F\urcorner\ulcorner n\urcorner\ulcorner m\urcorner \rhd\!\!\rhd |\ulcorner n\urcorner\ulcorner Grz\urcorner\ulcorner f\urcorner\ulcorner m\urcorner \rhd\!\!\rhd \ulcorner Grz\urcorner^{n}\ulcorner f\urcorner\ulcorner m\urcorner \rhd\!\!\rhd \ulcorner Grz^{n}fm\urcorner$$

as required to show that $\ulcorner F\urcorner$ represents F. ■