

An investigation of the use of
**APPLIED λ -CALCULI TO
ORGANISE THE RECURSIVE
FUNCTIONALS**
originally developed by Gödel,
Kleene, Grzegorzczuk, Platek and
others

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2004

Mark Andrew Price
Department of Mathematics

Contents

Table of Contents	4
Abstract	6
Declaration	7
Copyright	8
Introduction	9
1 Informal Recursion	11
1.1 Step-Wise Recursion	11
1.2 Kleene's Trick	21
1.3 Gödel and Kleene Recursors	24
2 Kleene's S1-S8	29
2.1 The Rules S1-S8	29
3 Platek's P1-P14	33
3.1 The Rules P1-P14	33
4 Setting up the Syntax	37
4.1 Types	38
4.2 Raw Terms	40
4.3 Judgements	43
4.4 Derivation System	43

4.5	Currying and Uncurrying	47
4.6	Semantics of the Category of Sets	48
5	Using the Syntax	52
5.1	Capturing the Natural Numbers	53
5.2	Substitution	53
5.3	α -equivalence	54
5.4	Computation Mechanism	55
5.5	Normalisation	63
5.6	Capturing Pairing Gadgets	67
5.7	Defining the System	68
6	Computation in the Syntax	70
6.1	Computation	70
6.2	Type Erasure	72
6.3	Subject Reduction	78
7	λ-Translation	80
7.1	Simulating Combinator Terms in λ -Calculi	81
7.2	λ -simulation algorithms	85
7.3	Getting New (and Old) Combinators from Old	101
8	Formal Recursion	105
8.1	Formal Recursion	105
8.2	Bernays' Trick	108
9	The Calculus of Primitive Recursion	113
9.1	The System λPR_0	113
9.2	Setting up λPR	114
9.3	The strength of λPR	117
9.4	Translation in λPR	121

10 Kleene Computable Functions	130
10.1 S1-S8 in λ PR	130
10.2 Naming the Kleene Computable Functionals?	134
11 Platek's Functions	138
11.1 P1-P14 in a λ -calculus	138
Conclusion	143
Bibliography	145

List of Tables

2.1	Kleene's S1-S8	31
3.1	Platek's P1-P14	34
4.1	The construction rules for derivations	44
6.1	Computation Rules	71
8.1	Reduction Properties of Rec_σ	109

Abstract

This thesis contains an analysis of various systems equivalent to Gödel's T, a discussion of various λ -translation algorithms and an analysis of the systems of Kleene and Platek at the first order level.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the Victoria University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the Department of Mathematics.

Introduction

This dissertation arose from an attempt to understand [17]. This is a survey paper and covers a lot more mathematics than could be covered in the time available.

Early on the decision was made to start by analysing Gödel's T. Here the underlying λ -calculus was developed and studied. It turned out that there were a variety of systems with exactly the same power as Gödel's and so a bigger λ -calculus which included various combinators was set up and used. This bigger system allowed a full analysis of all the required systems.

When analysing the relationships between the various subsystems the problem of λ -translation arose. This was studied in a fair amount of depth and we have a rather full discussion of the various algorithms that exist. A paper of Grzegorzczuk's allowed us to rediscover a seemingly forgotten result of his. Thus there is a full explanation of his algorithm and it's possible benefits.

After having set up the necessary syntax and analysed Gödel's T, it was necessary to move on. Coming back to [17], it seemed appropriate to move on to an analysis of Kleene's work. Initially it seemed that we would be able to study some of the higher order functionals generated from S1-S9 but time constraints meant that this looked unlikely. Initially study was restricted to the first order case, which rules out S9, and what functions this rules generated.

Now we had the choice to move on to either PCF or Platek's work knowing that time would not allow a full analysis of either. The decision was made to attempt to study Platek's work mainly because we had access to a copy of his original thesis. Thus we could return to study the original material in the hope that some fresh insight might be gained.

The dissertation itself has been laid out in a manner which does not reflect the order in which the work has been done. The material has been split into two areas, concrete and syntactic. With Chapters 1 to 3 covering the concrete material and the remaining chapters covering the syntactic.

I would like to take this opportunity to thank all the people who have made this dissertation possible. You know who you are.

Chapter 1

Informal Recursion

We begin by analysing various forms of recursion. Initially we do this informally since it will allow us to see what is actually going on before formalising the mathematics in Chapter 8.

The first section will examine body, head and tail recursion as well as some more complicated examples. The main aim of this section will be converting from recursion operators to iterators. We will also examine what role parameters play in recursion.

In Section 1.2 we will look at the general case for converting recursors to iterators. This involves using a technique which I have called Kleene's trick after it's inventor. The use of the name Kleene's trick also signifies that the technique is being used in an informal setting. When we formalise the technique it will be called Bernays' trick.

In the final section of this chapter we will look at two types of recursion operators which will be useful to us later on; Gödel and Kleene recursors. They will be used when we examine our main λ -calculus and Kleene computable functions.

1.1 Step-Wise Recursion

We wish to convert all types of recursion into an iteration. This will make analysis of the recursion easier and allow us to analyse some of the finer structure of the various systems we will study.

There are three main types of recursion and these are defined as follows.

1.1 DEFINITION.

The three main forms of **step-wise recursion** are the following.

	Base	Step	
(Head)	$f(p, 0) = gp$	$f(p, x') = h(p, x, f(p, x))$	
(Tail)	$f(p, 0) = gp$	$f(p, x') = f(q, x)$	where $q = k(p, x)$
(Body)	$f(p, 0) = gp$	$f(p, x') = h(p, x, f(q, x))$	where $q = k(p, x)$

Where g, h, k are the data functions with the following types.

$$g : \mathbb{G} = \mathbb{P} \rightarrow \mathbb{T} \qquad h : \mathbb{H} = \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{T} \rightarrow \mathbb{T} \qquad k : \mathbb{K} = \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{P}$$

With \mathbb{P} being the space where the parameter functions live and \mathbb{T} being the target space. ■

These recursions are by no means the only types of recursion and we will consider some much nastier types of recursion when we start looking at how we can convert these into iteration.

The following example shows how we go about converting head recursion into an iteration.

1.2 EXAMPLE.

We are given functions

$$\begin{aligned} f &: \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{T} \\ g &: \mathbb{P} \rightarrow \mathbb{T} \\ h &: \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{T} \rightarrow \mathbb{T} \end{aligned}$$

initially.

We now consider the head recursion with base step

$$f(p, 0) = gp$$

and recursion step as follows.

$$f(p, x') = h(p, x, f(p, x))$$

We use h to produce a function

$$H : \mathbb{P} \rightarrow (\mathbb{N} \rightarrow \mathbb{T}) \rightarrow (\mathbb{N} \rightarrow \mathbb{T}) = \mathbb{P} \rightarrow (\mathbb{N} \rightarrow \mathbb{T})'$$

with the following property.

$$Hp(x, t) = (x', h(p, x, t))$$

We now let $t = f(p, x)$ to obtain

$$Hp(x, f(p, x)) = (x', h(p, x, f(p, x))) = (x', f(p, x'))$$

where the last equality comes from the recursion step. By a simple induction we obtain

$$(Hp)^x(0, gp) = (x, f(p, x))$$

which is the required iteration. ■

The induction used here is quite straightforward but to satisfy those of you who might be concerned we will show it explicitly now. An almost identical induction will be used in all the future examples but we will not do it explicitly again since it is a matter of routine.

The base case is shown by the following observation.

$$(0, f(p, 0)) = (0, gp) = (Hp)^0(0, gp)$$

The first equality holds by the base step of the recursion. We now show the induction, $x \mapsto x'$, step.

$$(Hp)^{x'}(0, gp) = Hp((Hp)^x(0, gp)) = Hp(x, f(p, x)) = (x', f(p, x'))$$

There was really no thought required for that induction and we will never make mention of it again.

The reader might be lulled into a false sense of security in assuming that the above trick will work for tail and body recursions however it fails as soon as we apply to tail recursion as can be seen in the next example.

1.3 EXAMPLE.

Given the following functions.

$$f : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{T}$$

$$g : \mathbb{G}$$

$$k : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{P}$$

We consider the tail recursion with base step

$$f(p, 0) = gp$$

and the following recursion step.

$$f(p, x') = f(k(p, x), x)$$

Again we create a function

$$H : \mathbb{P} \rightarrow (\mathbb{N} \rightarrow \mathbb{T})'$$

where

$$Hp(x, t) = (x', t)$$

holds. We now let $t = f(p, x)$ to obtain

$$Hp(x, f(p, x)) = (x', f(p, x))$$

which is where we run into problems. We do not get $f(p, x')$ on the right since there is a function, k embedded in the recursion step. The simple trick of Example 1.2 fails in more complicated cases. ■

The same trick fails for body recursion but for a different reason. We can actually create the function H this time and it will be the same as in Example 1.2 before we replaced t . However when we try to replace t this time we notice that we cannot just let $t = f(p, x)$ since it needs to be $f(k, x)$ on the right hand side.

We now need to come up with another method for dealing with tail and body recursions since our initial method has failed at the first hurdle. We handle tail recursions in the following way.

1.4 EXAMPLE.

We are given functions

$$f : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{T}$$

$$g : \mathbb{G}$$

$$k : \mathbb{K}$$

and have tail recursion with base step

$$f(p, 0) = gp$$

and the following recursion step.

$$f(p, x') = f(k(p, x), x)$$

This time we let H be produced in the following way.

$$H : (\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T})) \rightarrow (\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T})) = (\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T}))'$$

where

$$H(x, \phi) = (x', \phi')$$

and ϕ' is given by the following.

$$\phi'p = \phi q \quad \text{where } q = k(p, x)$$

We now let $\phi = f(\cdot, x)$ and so obtain the following expression for H ,

$$H(x, f(\cdot, x)) = (x', \phi')$$

with

$$\phi'p = f(q, x) = f(p, x')$$

when we apply a parameter p . We thus have

$$H(x, f(\cdot, x)) = (x', f(\cdot, x'))$$

and by induction we obtain

$$H^x(0, g) = (x, f(\cdot, x))$$

as the required iteration. ■

This method is more versatile than the one we used for head recursions. In fact it can be applied to body recursions without any modification.

1.5 EXAMPLE.

We are given functions

$$f : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{T}$$

$$g : \mathbb{G}$$

$$h : \mathbb{H}$$

$$k : \mathbb{K}$$

and wish to consider the body recursion with base step

$$f(p, 0) = gp$$

and the following recursion step.

$$f(p, x') = h(p, x, f(q, x)) \quad \text{where } q = k(p, x)$$

We use h to produce a function

$$H : (\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T}))'$$

where

$$H(x, \phi) = (x', \phi')$$

and ϕ' is given by the following.

$$\phi'p = h(p, x, \phi q) \quad \text{where } q = k(p, x)$$

We now let $\phi = f(\cdot, x)$ to obtain the following expression for H .

$$H(x, f(\cdot, x)) = (x', \phi') \quad \text{where } \phi'p = h(p, x, f(q, x)) = f(p, x')$$

We thus find that

$$H(x, f(\cdot, x)) = (x', f(\cdot, x'))$$

holds and by induction we get

$$H^x(0, g) = (x, f(\cdot, x))$$

as the required iteration. ■

A body recursion can be generalised to get some much nastier recursions and we will look at handling two very nasty examples. Thankfully the same trick used to handle tail and body recursions can be used to handle these much more fearsome beasts. The first is an example of a type of recursion often called Devil recursion.

1.6 EXAMPLE.

Given the functions

$$f : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{T}$$

$$g : \mathbb{G}$$

$$h : \mathbb{H}$$

$$k : \mathbb{K}$$

$$l : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{T} \rightarrow \mathbb{P}$$

we consider the recursion with base step

$$f(p, 0) = gp$$

and the following recursion step.

$$f(p, x') = h(p, x, f(r, x)) \quad \text{where } r = l(q, x, t)$$

$$\text{where } t = f(q, x)$$

$$\text{where } q = k(p, x)$$

Again we use h to produce a function H which behaves in the following way.

$$H : (\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T}))'$$

$$H(x, \phi) = (x', \phi')$$

We use the following expression for ϕ' .

$$\phi'p = h(p, x, \phi r) \quad \text{where } r = l(q, x, t)$$

$$\text{where } t = \phi q$$

$$\text{where } q = k(p, x)$$

Again we let $\phi = f(\cdot, x)$.

$$H(x, f(\cdot, x)) = (x', \phi') \quad \text{where } \phi'p = h(p, x, f(r, x)) = f(p, x')$$

We now have

$$H(x, f(\cdot, x)) = (x', f(\cdot, x'))$$

which by induction gives

$$H^x(0, g) = (x, f(\cdot, x))$$

as the required iteration. ■

We now look at possibly one of the nastiest types of step wise recursion we can come up, in this example we alter the types within the recursion itself. Again our standard method for dealing with step wise recursion will conquer the dreaded beast.

1.7 EXAMPLE.

We are given the following functions

$$f : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{T}$$

$$g : \mathbb{G}$$

$$h : \mathbb{P} \rightarrow \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$$

$$k : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{P}$$

$$l : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{P}$$

$$m : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{T} \rightarrow \mathbb{P}$$

which give the following step-wise recursion. For the base step we have

$$f(p, 0) = gp$$

with the recursion step given as follows.

$$f(p, x') = h(q, r, x, f(s, x)) \quad \text{where } q = k(p, x)$$

$$\text{where } r = l(p, x)$$

$$\text{where } s = m(p, x, f(p, x))$$

We produce H in the usual way.

$$H : (\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T}))'$$

$$H(x, \phi) = (x', \phi')$$

We use the following expression for ϕ' .

$$\begin{aligned}\phi'p &= h(q, r, x, \phi s) && \text{where } q = k(p, x) \\ & && \text{where } r = l(p, x) \\ & && \text{where } s = m(p, x, \phi p)\end{aligned}$$

As usual we let $\phi' = f(\cdot, x)$.

$$H(x, f(\cdot, x)) = (x', \phi') \quad \text{where } \phi'p = h(q, r, x, f(s, x)) = f(p, x')$$

We now have

$$H(x, f(\cdot, x)) = (x', f(\cdot, x'))$$

which by induction gives

$$H^x(0, g) = (x, f(\cdot, x))$$

as the required iteration. ■

So this trick for converting recursion into iteration is very powerful, in fact it will even cope with head recursion which we dealt with separately earlier.

All the above types of recursion have involved parameters. It is possible to have recursion without these parameters. The following example shows how to go from head recursion with parameters to head recursion without parameters.

1.8 EXAMPLE.

We are given functions

$$\begin{aligned}f &: \mathbb{N} \times \mathbb{P} \rightarrow \mathbb{T} \\ g &: \mathbb{G} \\ h &: \mathbb{N} \rightarrow \mathbb{P} \rightarrow \mathbb{T} \rightarrow \mathbb{T}\end{aligned}$$

where \mathbb{P} is the space of parameters. These give rise to head recursion in the following way.

$$\begin{aligned}f(0, p) &= g \\ f(x', p) &= hxp f(x, p)\end{aligned}$$

We can now consider f as a function

$$f : \mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T})$$

where the recursion steps are

$$\begin{aligned} f0 &= g \\ fx' &= \bar{h}x(fx) \end{aligned}$$

without parameter input. We find that \bar{h} is a function

$$\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T}) \rightarrow (\mathbb{P} \rightarrow \mathbb{T}) = \mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T})'$$

defined by

$$\bar{h}x\phi p = hxp(\phi p)$$

where p is the parameter.

Thus we have f being given by a parameter free recursion. ■

We have moved to a parameter free recursion by 'hiding' the parameters rather than explicitly stating them. So strictly speaking our parameter free recursion is not really parameter free.

Also note that we have made the function more 'complicated' since instead of just using first order types, it now uses higher order types. So while we have made our function 'simpler' in the sense that it now longer needs parameters, we lose out because it now needs higher types as inputs.

We thus find that our parameter recursion examples are just examples of parameter free recursion. Since the above trick can be applied to any parameter recursion we wish to look at.

1.9 EXAMPLE.

We look at the functions used in Example 1.7. We change the order of the inputs and now consider f as a function

$$\mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T})$$

which gives rise to

$$\begin{aligned} f0 &= g \\ fx' &= \bar{h}x(fx) \end{aligned}$$

with

$$\begin{aligned}\bar{h}x\phi p &= hxp\phi q \\ q &= kpx\phi r \\ r &= ltx\phi s \\ s &= mpx\end{aligned}$$

where p and t are parameters. Thus we have a parameter free recursion. ■

Thus we have ‘tricks’ for converting recursion with parameters to parameter free recursion and iteration. One shows that we lose nothing by removing the parameters, the other that we lose nothing by removing recursors from any λ -calculus system.

1.2 Kleene’s Trick

In this section we will look at how to go from a recursor to an iterator in an informal setting. This is done using something called Kleene’s trick. It is sometimes also referred to as Bernays’ trick and for our purposes we will call the informal version Kleene’s trick and the formal version Bernays’ trick, thus avoiding any misunderstanding. It is important to note that both tricks use the same idea and that the informal version allows us to get a better grasp of what is actually going on.

We construct a parameter-free function $R : \mathbb{N} \rightarrow \mathbb{S}$ by

$$R0 = s \quad Rx' = \psi x(Rx)$$

where $s : \mathbb{S}$ and $\psi : \mathbb{N} \rightarrow \mathbb{S}'$ are given.

We wish to think of this as higher order function $R_{\mathbb{S}}$ which when supplied with the input data s and ψ will return the function R . We find that

$$R_{\mathbb{S}} : (\mathbb{N} \rightarrow \mathbb{S}') \rightarrow \mathbb{S}^+$$

gives us $R = R_{\mathbb{S}}\psi s$ as the constructed function, where $\mathbb{S}^+ = \mathbb{S}' \rightarrow \mathbb{S}$.

This new function $R_{\mathbb{S}}$ is the recursor over \mathbb{S} which is essentially the recursion gadget we have in our system. Since this is an informal discussion we are only looking at the ideas behind the conversion rather than making sure we have exactly

the right recursor. It is not too difficult to see how we would turn $R_{\mathbb{S}}$ into the recursion gadget we have in λBIG .

It is quite straightforward to see that all body recursion can be captured by Gödel recursors. This is shown in the following lemma.

1.10 LEMMA.

Every body recursion can be achieved using a Gödel recursor.

Proof.

For body recursion we are given data functions

$$\theta : \mathbb{F} \quad \psi : \mathbb{N} \rightarrow \mathbb{S} \rightarrow \mathbb{F} \quad \kappa : \mathbb{N} \rightarrow \mathbb{P}'$$

and a function

$$\phi : \mathbb{N} \rightarrow \mathbb{F}$$

which behaves in the following way.

$$\phi 0 p = \theta p \quad \phi r' p = \psi r s p \quad \text{where } s = \phi r p^+ \quad \text{where } p^+ = \kappa r p$$

We now let $\Psi : \mathbb{N} \rightarrow \mathbb{F}'$ be given by

$$\Psi r f p = \psi r (f p^+) p \quad \text{where } p^+ = \kappa r p$$

for $r : \mathbb{N}, f : \mathbb{F}, p : \mathbb{P}$.

Consider $\Phi : \mathbb{N} \rightarrow \mathbb{F}$ which is obtained by Gödel recursion from θ and Ψ . So we have

$$\Phi 0 = \theta \quad \Phi r' = \Phi r (\Psi r)$$

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

It just remains to show that $Rr = \Phi r$ and hence $R = \Phi$. This is done by an induction over r . The base case follows from noting that

$$R0p = \theta p = \Psi 0 p$$

holds. For the induction step $r \mapsto r'$ we note that

$$\Phi r' p = \Psi r (\Phi r) p = \Psi r (Rr) p \text{ by the induction hypothesis}$$

where

$$\Psi r(Rr)p = \psi r((Rr)p^+)p = Rr'p$$

as required. ■

We note that θ, ψ, κ are all first order types but that $\Phi : \mathbb{N} \rightarrow \mathbb{N}''$ is second order. So we have in some sense lost the ‘simplicity’ behind body recursion in order to capture it in our λ -calculus.

We now wish to turn our Gödel recursor into an iterator. The first thing we note is that if the input function ψ for $R_{\mathbb{S}}$ is insensitive to its first argument we can omit this argument and use a data function $\psi : \mathbb{S}'$ to produce R where

$$Rr = \psi^r s$$

for all $r \in \mathbb{N}$ and $s \in \mathbb{S}$. Thus we find that Rr is the r -fold iterate of ψ evaluated at s and so hence can be captured by an iterator.

If however the input function is not insensitive to its first argument then we need to use Kleene’s trick. Essentially we need to find a function $F : (\mathbb{N} \times \mathbb{S})'$ such that

$$F^x(\mathbf{0}, \theta) = (x, Rx)$$

holds. Then all we have to do is use our pairing gadgets to get at the right hand side of this function.

We find that the required F is the following function.

$$F(x, u) = (x', \phi x u)$$

We let $u = Rx$ and we find that

$$F(x, Rx) = (x', \phi x(Rx)) = (x', Rx')$$

holds. Now we proceed by induction to show that

$$F^x(\mathbf{0}, \theta) = (x, Rx)$$

is true. The base case follows from the simple observation that

$$F^0(\mathbf{0}, \theta) = F(\mathbf{0}, \theta) = (\mathbf{0}, R\mathbf{0})$$

holds. The induction step, $x \mapsto x'$ is given by the following.

$$F^{x'}(0, \theta) = F(F^x(0, \theta)) = F(x, Rx) = (x', Rx')$$

We now have a function F which can be captured by an iterator and so we have shown how to go from a recursor to an iterator.

1.3 Gödel and Kleene Recursors

Each recursor needs to consume various inputs, and it is useful to divide these into three kinds: the data gadgets, the recursion input, and the parameters.

For the simpler recursors there are two data gadgets.

- A function (or value) g used to obtain the base value
- A function h used to make the recursion step

There may also be other data functions which are used to manipulate the parameters.

For us the recursion input is always a natural number. (There are other quite natural free structures over which a recursion can proceed. For instance, lists over a fixed alphabet.)

The parameters are simply the other inputs. Often these can be hidden but in other cases they play a crucial role. We will see examples of both kinds.

1.11 EXAMPLE.

The recursion step from head recursion

$$f(p, x') = h(p, x, f(p, x))$$

is an example where the parameter is simply another input.

The recursion step from Example 1.7

$$\begin{aligned} f(p, x') = h(p, x, f(q, x)) \quad &\text{where } q = k(p, x, f(r, x)) \\ &\text{where } r = l(t, x, f(s, x)) \\ &\text{where } s = m(t, x) \end{aligned}$$

is an example where the parameter is hidden and plays quite a crucial role. ■

Of course, all of these gadgets must be typed, and the types must be such that certain combinations are possible.

For some recursors the exact order of the inputs is not too important, but sometimes one particular order can make for a simpler account. A useful rule of thumb is: Either consume the parameter last (and hence hide them), or consume them first.

1.12 EXAMPLE.

Here we only show the recursion steps. This recursion step

$$f(p, x') = h(p, x, f(q, x)) \text{ where } q = k(p, x)$$

is an example where the parameter is consumed first. While the following recursion step

$$fx' = hx(fx)$$

is an example where the parameter is hidden. ■

Since we do not find concurrence for the choice of types of recursors within the literature it would be impossible to match the choice of types with all the accounts. Therefore we will just use the most convenient choice of types and stick with that.

As an illustration we will look at the difference between the Gödel recursors and the Kleene recursors, as described in [1] on pages 345 and 362 respectively. They don't give the various types explicitly (although the information is contained within the account). Also the recursors consume the input as tuples which we will get around by using a curried form.

From [1] p.345, the G-recursor has rules

$$Ggh0 = g \quad Gghx' = hx(Gghx)$$

where g and h are the data functions and x is the recursion input. These have types

$$\begin{aligned} g &: \mathbb{G} \\ h &: \mathbb{H} = \mathbb{N} \rightarrow \mathbb{G} \rightarrow \mathbb{G} = \mathbb{N} \rightarrow \mathbb{G}' \\ x &: \mathbb{N} \\ G &: \mathbb{G} \rightarrow \mathbb{H} \rightarrow \mathbb{N} \rightarrow \mathbb{G} \end{aligned}$$

where \mathbb{G} is an arbitrary type.

For the particular case with $\mathbb{G} = \mathbb{N}$ we obtain the base recursor. Thus

$$Bak0 = a \quad Bakx' = kx(Bakx)$$

where

$$\begin{aligned} a &: \mathbb{N} \\ k &: \mathbb{K} = \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ B &: \mathbb{N} \rightarrow \mathbb{K} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \end{aligned}$$

are the types.

The K-recursor as described on p.362 has the following type for \mathbb{G} .

$$\mathbb{G} = \mathbb{P}_m \rightarrow \dots \rightarrow \mathbb{P}_1 \rightarrow \mathbb{N}$$

So it displays the m parameter types. For this illustration we can collapse (tuple) the parameters so that

$$\mathbb{G} = \mathbb{P} \rightarrow \mathbb{N}$$

is the appropriate type. However, the case $m = 0$ is allowed, and then $\mathbb{G} = \mathbb{N}$.

The K-recursor has rules

$$Kgl0p = gp \quad Kglx'p = lx(Kglxp)p$$

where

$$\begin{aligned} g &: \mathbb{G} = \mathbb{P} \rightarrow \mathbb{N} \\ l &: \mathbb{L} = \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{G} \\ x &: \mathbb{N} \\ p &: \mathbb{P} \\ K &: \mathbb{G} \rightarrow \mathbb{L} \rightarrow \mathbb{N} \rightarrow \mathbb{G} \end{aligned}$$

are the types involved.

Notice that when $m = 0$, that is when \mathbb{P} is empty, K is just B.

As indicated in [1] and first proved by Kleene, the K-recursors are no more powerful than B .

Consider the data functions and parameters

$$g : \mathbb{G} \quad l : \mathbb{L} \quad p : \mathbb{P}$$

for the K-recursors, as given above. Let

$$h : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

be the function given by

$$hpxy = lxy$$

for $p : \mathbb{P}, x : \mathbb{N}, y : \mathbb{N}$. Thus we move from l to h simply by shuffling the inputs.

Notice that

$$gp : \mathbb{N} \quad hp : \mathbb{K}$$

are data gadgets for the B-recursor.

1.13 LEMMA.

For arbitrary data

$$g : \mathbb{G} \quad l : \mathbb{L} \quad p : \mathbb{P}$$

with

$$h : \mathbb{P} \rightarrow \mathbb{K}$$

as defined above, we have

$$Kglxp = B(gp)(hp)x$$

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

Proof.

We proceed by induction on x . The base case, $x = 0$, is immediate.

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

$$Kglx'p = lx(Kglxp)p = lx(B(gp)(hp)x)p$$

with the second equality following from the induction hypothesis. We also have

$$B(gp)(hp)x' = (hp)x(B(gp)(hp)x) = lx(B(gp)(hp)x)p$$

as required. ■

A slight rearrangement of the data function l leads to a rather neater simulation of the K-recursor. Suppose we require l to consume the parameters first. Thus we

have

$$\begin{aligned} g & : \mathbb{G} = \mathbb{P} \rightarrow \mathbb{N} \\ l & : \mathbb{L} = \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\ x & : \mathbb{N} \\ p & : \mathbb{P} \\ K & : \mathbb{G} \rightarrow \mathbb{L} \rightarrow \mathbb{N} \rightarrow \mathbb{G} \end{aligned}$$

with

$$Kgl0p = gp \quad Kglx'p = lpx(Kglxp)$$

as the recursive rule. Then

$$Kglxp = B(gp)(lp)x = \top Bglpx$$

where \top is the Turner combinator.

This shows that K is nothing more than a parameterised version of B . This becomes even neater if we make K consume the recursion input last. Thus

$$K : \mathbb{G} \rightarrow \mathbb{L} \rightarrow \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

with

$$Kglp0 = gp \quad Kglpx' = lpx(Kglpx)$$

as the recursion rules. Then

$$K = \top B$$

and we don't need to mention the inputs.

Chapter 2

Kleene's S1-S8

Here we introduce and analyse a system of rules for generating functions introduced by Kleene. His S1-S8 allow us to generate all of the primitive recursive functions. This result will be proved much later in Chapter 9 when we examine the functions we get from his rules.

Initially we will examine his rules in their original setting without attempting to translate them into a λ -calculus. One of the disadvantages of doing this is that we will be unable to distinguish between a function and its construction. However after seeing the difficulties of this approach it will allow us to greater appreciate the advantages of using embedding it in a λ -calculus which does distinguish between a function and its construction.

2.1 The Rules S1-S8

Most of the material in this section is based on the accounts of Kleene computable functionals in [17] pages 28 and 29 and in [1] page 360 onwards.

Firstly we introduce concrete types and functions which we need to make clear are different from clean and curried types which we will introduce in Definition 9.1. The concrete types we introduce here are actual sets and functions and so exist in the 'real world'. The curried are syntactic analogues of these functions and so only exist in the mind of the mathematician.

2.1 DEFINITION.

The clean concrete types

$$(\mathbb{N}[l] \mid l < \omega)$$

are generated by

$$\mathbb{N}[0] = \mathbb{N} \quad \mathbb{N}[l + 1] = \mathbb{N}[l] \rightarrow \mathbb{N}$$

(for $l < \omega$). ■

We now define the functionals which we wish to use these types to get at. These functionals are something which Kleene himself come up with in [13], which attempt to get at higher order recursion.

2.2 DEFINITION.

Let $\mathbb{S}_1, \dots, \mathbb{S}_m$ be clean types. Each clean functional (Kleene computable functional) has the form

$$F : \mathbb{S}_m \times \dots \times \mathbb{S}_1 \rightarrow \mathbb{N}$$

(with $m \neq 0$). ■

Here we have deliberately indexed these from right to left. We will see why this is useful later on when we look at Kleene's rules for construction of clean functionals.

Strictly speaking we should distinguish between a clean function and the construction of such a function. The same function may be constructed in several different ways. This is similar to the distinction we will make between primitive recursive functions and their constructions. Similarly we are not too concerned about which construction is used just that one exists.

Each clean construction produces

$$\begin{aligned} \{label\} \quad F : \mathbb{S}_m \times \dots \times \mathbb{S}_1 \rightarrow \mathbb{N} \\ (x_m, \dots, x_1) \mapsto f \end{aligned}$$

where f is an 'expression' built up from the inputs x_1, \dots, x_m and certain other data.

The label encodes the construction.

S1	$(\underline{x} : \underline{\sigma})$	$= x_1 + 1$
S2	$(\underline{x} : \underline{\sigma})$	$= q$
S3	$(\underline{x} : \underline{\sigma})$	$= x_1$
S4	$(\underline{x} : \underline{\sigma})$	$= g(h(\underline{x}), \underline{x})$
S5	$f(\underline{x}, 0)$	$= g\underline{x}$
	$f(\underline{x}, r')$	$= h(\underline{x}, r, (f\underline{x}, r))$
S6	$(\underline{x} : \underline{\sigma})$	$= (x_{k+1}, x_1, \dots, x_k, x_{k+2}, \dots, x_r)$
S7	$(\underline{x} : \underline{\sigma})$	$= x_1(x_2)$
S8	$f(\underline{x} : \underline{\sigma}, y : \Pi^{++})$	$= y(\lambda z : \Pi, h(\underline{x}, y, z))$

Table 2.1: Kleene's S1-S8

We find that in [17] the following notation is used

$$\{label\} \quad (\underline{x} : \underline{\sigma}) = f$$

where

$$\begin{aligned} \underline{\sigma} &= \mathbb{S}_m \times \dots \times \mathbb{S}_1 \\ \underline{x} &= (x_m, \dots, x_1) \end{aligned}$$

are the abbreviations. We will use both notations interchangeably since sometimes it is clearer to use Longley's notation to hide the specific types and at other times we want to clearly show what is happening with them.

Each such construction proceeds by iterative use of the rules S1-S9. We begin by looking at S1-S7 before moving on to look at S8.

Kleene's S9 has quite a different form to the other rules. It has a self-referential nature, since one of the inputs is a coding of a construction of a function within the system. Because of this aspect we do not attempt to analyse S9 in this dissertation.

The Kleene rules are given in Table 2.1.

We note that Kleene makes no distinction between functions and the derivation of the functions. When we translate S1-S8 into a suitable λ -calculus we will be making this distinction and as a result our analysis will be clearer.

S1 is the successor function. We note that we are only allowed to apply it to the first coordinate.

S2 allows us to obtain any numerical constant in our system. This is a rather strange way of doing things. It would make more sense to have just the constant

zero since we have already introduced the successor function. We can generate every numerical function from just the zero and successor function, thus allowing our system to be simpler.

S3 is essentially the projection function and we note that it only projects from the first coordinate. This seems a bit strange since it would be easier to allow projection from any position. This limited projection means we need an additional rule, S6, which allows us to shuffle the coordinates around. Thus S1 combined with S6 allows projection from any coordinate.

S4 is a rather strange type of composition. As we shall see later it will allow us to have substitution in our system.

S5 is the standard primitive recursion. At the lower order levels this is what will give us the power of the system.

S6 is just a book keeping rule. Since we have restricted a lot of rules to only allow them to act on the first coordinate, we need a rule which will allow us to permute any coordinate to the first position. This is precisely what S6 does.

S7 is just application. We note that it is application at the lower order level and that higher type application is covered by S8.

S8 is higher type application. This is where the system gets a lot of its power at higher order levels, along with S9.

We note that two of the most desirable rules have not been included. We do not have the standard composition rule nor the substitution rule. Admittedly substitution can be obtained from S4. It is far from clear why Kleene did not include these rules. It could be possible that they cause problems at the higher order levels but this is not clear and no explanation is given in the literature.

Chapter 3

Platek's P1-P14

In Chapter 2 we looked at clean functionals which used an indexing system. This index system allows us to use the entire system as an input and so get at a system containing higher types. Kleene uses S9 to get at this system however we do not have space to cover it in this dissertation.

This index system can be rather confusing and often hides what is actually happening and so it is a logical question to ask whether or not it is possible to avoid using indexing. One way of doing this is to use hereditarily consistent functionals which were first introduced by Platek in [19].

Platek also showed that Kleene had gone wrong in his original theory, set up in [13], [14]. Platek showed that the first recursion theorem does not hold in this theory. Which Kleene himself acknowledges when he attempts to correct the error in [15].

Here we will examine the simplest of the systems set up Platek in it's original context. A more fuller analysis, including a translation into a λ -calculus will be covered in Chapter 10.

3.1 The Rules P1-P14

Having read Platek's thesis, [19], and looked at the summarised version in [18], I have realised that there is too much information contained within these to be able to cover it in this dissertation.

P1	$\phi(x, \underline{z})$	$= 0$
P2	$\phi(x, \underline{z})$	$= x$
P3	$\phi(x, y, \underline{z})$	$= \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$
P4	$\phi(\underline{z})$	$= 0$
P5	$\phi(x, \underline{z})$	$= x + 1$
P6	$\phi(x, y, \underline{z})$	$= \mathbf{Pair}(x, y)$
P7	$\phi(x, \underline{z})$	$= \mathbf{Right}(x)$
P8	$\phi(x, \underline{z})$	$= \mathbf{Left}(x)$
P9	$\phi(\underline{z})$	$= \psi_1(\psi_2(\underline{z}), \underline{z})$
P10	$\phi(0, \underline{z})$	$= \psi(\underline{z})$
	$\phi(k + 1, \underline{z})$	$= \psi_2(\phi(k, \underline{z}), k, \underline{z})$
P11	$\phi(\underline{z})$	$= \psi(\underline{z}')$
P13	$\phi(x_1, \dots, x_k, \underline{z})$	$= \psi(x_1, \dots, x_k)$
P14	$\phi(\underline{z})$	$= \Phi_i(\lambda \underline{x} \psi_1(\underline{x}, \underline{z}), \dots, \lambda \underline{x} \psi_k(\underline{x}, \underline{z}))$

Table 3.1: Platek's P1-P14

However there is something which can be taken from Platek's work which is relevant. In [18], various systems are set up and we will examine the simplest of these and only allow total functions as inputs.

This new system has 14 rules which we will call P1-P14. These are listed in Table 3.1. In the text these are referred to as S1-S14 but this labeling will cause confuse with Kleene's S1-S9 and Kleene's is the more standard use of the S label.

We are only concerned with looking at total functions, so some of the conditions of Platek's P1-P14 can be ignored. Also P12 will not be studied since it involves using the indexing system as an input. This gets at higher order functions in much the same way as Kleene's S9 but this does not concern us here.

We need to allow the following constants in our system.

$$\mathbb{N}, \mathbf{Pair}, \mathbf{Left}, \mathbf{Right}, \underline{z} = z_1, \dots, z_k \in A$$

Also ϕ_i is a function with type \mathcal{N}' , Φ_i is a total function. We can let Φ_i be a jump operator which will allow us to get an idea of the size of the system. It is however

far from clear whether or not a jump operator is an allowable choice for Φ_i but we will assume it is.

In Platek's version, he looks at a set which can contain more than just the natural numbers. So he has extra conditions on P1, P5 and P10 which we have ignored. This is why P1 appears to not actually add anything to the system. For our purposes we will ignore it since it's main use is as a conditional to determine whether or not a given term is a natural number but all our terms are natural numbers and so return the constant 0, which we can get from P4.

We now examine in turn each of Platek's P1-P14.

P1 introduces the zero constant.

P2 is a projection from the first coordinate. As with Kleene we must ask ourselves why Platek restricts the projection.

P3 introduces a conditional. We must assume that conditionals cannot be constructed in this system. We will discuss possible reasons for this when we translate P1-P14 into a λ -calculus.

P4 again introduces the zero constant. If we were working in a set which was bigger than the natural numbers, P1 and P4 would not be identical. P1 is in fact a conditional which checks to see whether or not a particular element of the set is a natural number or not.

P5 is the successor function. Combined with P4 this allows us to obtain any numerical constant we wish.

P6 introduces the pairing gadget.

P7 introduces the right projection gadget.

P8 introduces the left projection gadget.

P9 is the same type of composition used by Kleene is S4.

P10 is primitive recursion.

P11 is just a shuffling rule. Platek needs this since he restricts a lot of his rules to only act on the first coordinate.

P13 adds dummy variables.

P14 is a type of substitution. This rule is where the power is. Depending on

the restrictions on the Φ_i we can easily go beyond the class of primitive recursive functions.

Chapter 4

Setting up the Syntax

In this chapter we look at various syntactic systems designed to capture numeric gadgets at the first order and higher order levels. There are four principal systems designed to compare the use of

λ -abstraction vs combinator abstraction
recursors vs iterators

and various subsidiary gadgets. These systems have many things in common, and differ in only a few small – but crucial – aspects. It is convenient (both for comparison and presentational purposes) to exhibit these systems as subsystems of one all embracing system, λ BIG.

The system λ BIG is an applied λ -calculus (with combinators) designed to capture numeric gadgets. As usual with such a system, there are various syntactic categories, and these can take a bit of time to set up. Here is a brief summary of these categories, and what we will do with them.

The first section formally introduces the idea of types. It shows how to generate syntactic types as well as the conventions for handling them.

The next section looks at raw terms. These are introduced into our system as constants and we examine the conventions for generating more terms and how to handle them.

Section 4.3 introduces the idea of judgements. These allow us to impose rules on how terms are combined as well as creating a typing discipline.

In Section 4.4 we look at the rules that our system imposes on the terms. The rules for derivations allow us to determine whether or not a given term is a part of the system. We also look at some examples of derivations.

In the next section we look at currying and uncurrying. These techniques allow us to change the inputs of functions. They are crucial tools when we start to formalise recursion.

Finally we look at the semantics of our system. We find that the natural setting for this is in the category of sets. We show that derivations and reductions are well defined in this setting.

4.1 Types

All types are generated from an atom \mathcal{N} , intended to name the set of natural numbers.

All systems will have an arrow formation

$$\frac{\sigma \quad \rho}{\sigma \rightarrow \rho}$$

for types, and some will have a product formation

$$\frac{\sigma \quad \rho}{\sigma \times \rho}$$

(with associated gadgets).

In particular, the type structure is fairly straight forward.

Formally, we have the following.

4.1 DEFINITION.

The full syntactic category of **types** is generated as follows.

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

We let $\rho, \sigma, \tau, \dots$ range over types. ■

This is a recursive definition and allows lots of types to be obtained from just our original type \mathcal{N} .

4.2 EXAMPLE.

The following are all types built up just from \mathcal{N} .

$$\begin{array}{ll} \mathcal{N} & \\ (\mathcal{N} \rightarrow \mathcal{N}) & (\mathcal{N} \times \mathcal{N}) \\ ((\mathcal{N} \rightarrow \mathcal{N}) \rightarrow (\mathcal{N} \rightarrow \mathcal{N})) & ((\mathcal{N} \rightarrow \mathcal{N}) \times \mathcal{N}) \\ (\mathcal{N} \times (\mathcal{N} \rightarrow \mathcal{N})) & ((\mathcal{N} \rightarrow \mathcal{N}) \times (\mathcal{N} \rightarrow \mathcal{N})) \end{array}$$

etc ■

The notation for types needs to be made a bit clearer since as we start to generate them things become messy as we have shown in Example 4.2. We thus introduce the following convention

4.3 CONVENTION.

Let τ' stand for $(\tau \rightarrow \tau)$. ■

This generalises in the following way.

$$\begin{aligned} \tau'' &= (\tau' \rightarrow \tau') \\ &= (\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau) \\ \tau''' &= (\tau'' \rightarrow \tau'') \\ &= (\tau' \rightarrow \tau') \rightarrow (\tau' \rightarrow \tau') \\ &= ((\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)) \rightarrow ((\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)) \end{aligned}$$

The brackets in the type are part of the standard definition and ensure that it is uniquely parsed. However to make things easier to read and this is pretty much the standard convention, they are left out when it is possible to do so. The informal convention for admitting brackets from terms is as follows. Given a type $(\tau \rightarrow (\sigma \rightarrow \rho))$ we leave out the brackets to obtain $\tau \rightarrow \sigma \rightarrow \rho$. Thus to reintroduce brackets they are reinstated from the right. This convention will be made formal in Convention 4.5.

4.2 Raw Terms

These are generated from a stock of placeholders – which here we call **identifiers** – together with certain constants. These constants can be divided into 5 kinds.

- **Basic numeric gadgets**

$$0 : \mathcal{N} \qquad \text{Suc} : \mathcal{N}'$$

These are intended to name zero and the successor function respectively.

- **Recursion gadgets**

$$\text{It}_\sigma : \mathcal{N} \rightarrow \sigma''$$

$$\text{Rec}_\sigma : \sigma \rightarrow (\sigma \rightarrow \mathcal{N} \rightarrow \sigma) \rightarrow \mathcal{N} \rightarrow \sigma$$

These are called the iterator over σ and the recursion operator over σ respectively.

- **Pairing gadgets**

$$\text{Pair}_{\sigma,\rho} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho$$

$$\text{Left}_{\sigma,\rho} : \sigma \times \rho \rightarrow \sigma$$

$$\text{Right}_{\sigma,\rho} : \sigma \times \rho \rightarrow \rho$$

These allow us to ‘pair’ terms and retrieve the right and left elements of a product of terms.

- **Combinators**

$$\text{I}_\sigma : \sigma \rightarrow \sigma$$

$$\text{K}_{\rho,\sigma} : \sigma \rightarrow (\rho \rightarrow \sigma)$$

$$\text{S}_{\rho,\sigma,\tau} : (\rho \rightarrow \sigma \rightarrow \tau) \rightarrow (\rho \rightarrow \sigma) \rightarrow (\rho \rightarrow \tau)$$

$$\text{B}_{\rho,\sigma,\tau} : (\rho \rightarrow \sigma) \rightarrow ((\tau \rightarrow \rho) \rightarrow (\tau \rightarrow \sigma))$$

$$\text{C}_{\rho,\sigma,\tau} : (\rho \rightarrow (\sigma \rightarrow \tau)) \rightarrow (\sigma \rightarrow (\rho \rightarrow \tau))$$

$$\text{D}_{\rho,\sigma} : (\sigma \rightarrow (\sigma \rightarrow \rho)) \rightarrow (\sigma \rightarrow \rho)$$

These are more combinators than we strictly need and the relationships between them will be discussed later.

The type of the recursion operator used here is the same as the type used in [17]. However there does not appear to be a standard type for the recursion operator, for example [7] and [21] have a recursion operator of type $\sigma \rightarrow (\mathcal{N} \rightarrow \sigma') \rightarrow (\mathcal{N} \rightarrow \sigma)$. We have already discussed these possibilities at the informal level. However this is not a problem since it is possible to change the order of the types using the following tricks.

- To go from $t : \sigma \rightarrow \rho \rightarrow \tau$ to $\bar{t} : \rho \rightarrow \sigma \rightarrow \tau$ define $\bar{t} = \lambda x : \rho, y : \sigma. tyx$
- To go from $t : \rho \times \sigma \rightarrow \tau$ to $\bar{t} : \sigma \times \rho \rightarrow \tau$
define $\bar{t} = \lambda z : \sigma \times \rho. t\text{Pair}(\text{Right } z \text{ Left } z)$

Thus we can change the order of types as long as they are in a curried form.

We define the raw terms in the following way.

4.4 DEFINITION.

The raw terms of the calculus are generated from an unlimited stock of identifiers and the constants given above in the following manner.

- Each identifier is a raw term.
- Each constant is a raw term.
- If p, q are raw terms then the application (pq) 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.

These are the only raw terms. ■

Here we let x, y, z, \dots range over the identifiers and p, q, r, s, t, \dots range over the raw terms.

Here each term comes with brackets attached and it is often notationally easier to remove them. The main function of the brackets is to ensure that terms are unique. So we can only remove them when it is clear where they should be placed.

Now we can set up the following formal convention with regards the use of brackets in our systems.

4.5 CONVENTION.

- Given a type $(\tau \rightarrow (\sigma \rightarrow \rho))$ we leave out the brackets to obtain $\tau \rightarrow \sigma \rightarrow \rho$.
- Given a term $((ts) r)$ we remove the brackets to obtain tsr .

Giving a convention for both types and terms. ■

To replace the brackets in a type we reinstate them from the right and to replace the brackets for a term we reinstate the brackets from the left. This means that brackets in terms are reinstated in the opposite way to the way to types. This is not done to confuse the reader but because it matches well with currying.

There is another convention we need to introduce here which is in conflict with Convention 4.5 for the use of brackets.

4.6 CONVENTION.

For arbitrary terms t, s we let

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

for each $m \in \mathbb{N}$ and there are m occurrences of t .

Thus we have that

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

for each $m \in \mathbb{N}$. ■

The following example shows the usefulness of this new notation.

4.7 EXAMPLE.

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

We proceed by induction over n (with m fixed). Since $t^0 s = s$ and $m + 0 = m$, the base case, $m = 0$, is immediate.

For the induction step, $n \mapsto n + 1$, we have

$$t^{n+1}(t^m s) = t(t^n(t^m s)) = t(t^{m+n} s) = t^{m+n+1} s$$

using the definition of t^{n+1} at the first and third equality, and the induction hypothesis at the second. ■

4.3 Judgements

The formation of raw terms pays no attention to the intended typing discipline. To correct this we work with judgements

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

each of which has a

- **context** Γ , ie a list of declarations $x_l : \sigma_l, \dots, x_1 : \sigma_1$
- a **statement** $t : \tau$ which is a raw term paired with a type.

The intention is that the judgement should be readable as

‘within the legal context Γ ,
the raw term t is well-formed,
to inhabit the acceptable type τ ’

and to achieve this each judgement must be built up in a certain way. This is given by the derivation system.

4.4 Derivation System

A derivation system is a set of rules for generating judgements. Any judgement generated according to a given derivation system is correct in the sense that it can be read in the manner given in Section 4.3.

We have the following three provisos for our derivation system.

- At an axiom the context Γ must be legal and the statement $k : \kappa$ 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 .

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

Table 4.1: The construction rules for derivations

- 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).

These together with the rules contained in Table 4.1 give us the required derivation system.

We use the notation

$$(\nabla) \Gamma \vdash t : \tau$$

where ∇ represents the derivation of the judgement. Each derivation is either a leaf or a compound built up from smaller derivations in one of the ways mentioned in Table 4.1.

We will now show how to grow derivations in the following examples.

4.8 EXAMPLES.

We show that each of

$$\begin{aligned} (i) \quad & \vdash \mathbf{I}_\lambda : \iota & (ii) \quad & \vdash \mathbf{K}_\lambda : \kappa & (iii) \quad & \vdash \mathbf{S}_\lambda : \sigma \\ (iv) \quad & \vdash \mathbf{B}_\lambda : \beta & (v) \quad & \vdash \mathbf{C}_\lambda : \gamma & (vi) \quad & \vdash \mathbf{D}_\lambda : \delta \end{aligned}$$

is derivable.

For arbitrary types θ, ϕ, ψ , let

$$\begin{array}{ll}
 \mathbf{I}_\lambda = (\lambda x : \theta . x) : \iota & \iota = \theta \rightarrow \theta \\
 \mathbf{K}_\lambda = (\lambda x : \theta, y : \phi . x) : \kappa & \kappa = \theta \rightarrow \phi \rightarrow \theta \\
 \mathbf{S}_\lambda = (\lambda x : \theta \rightarrow \phi \rightarrow \psi, y : \theta \rightarrow \phi, z : \theta . (xz)(yz)) : \sigma & \sigma = \alpha \rightarrow \epsilon \rightarrow \zeta \\
 \mathbf{B}_\lambda = (\lambda x : \theta \rightarrow \phi, y : \psi \rightarrow \theta, z : \psi . x(yz)) : \beta & \beta = \epsilon \rightarrow \eta \\
 \mathbf{C}_\lambda = (\lambda x : \theta \rightarrow (\phi \rightarrow \psi), y : \phi, z : \theta . xzy) : \gamma & \gamma = \theta \rightarrow \iota \rightarrow \mu \\
 \mathbf{D}_\lambda = (\lambda x : \theta \rightarrow (\theta \rightarrow \phi), y : \theta . xyy) : \delta & \delta = \nu \rightarrow \epsilon
 \end{array}$$

where

$$\begin{array}{l}
 \alpha = \theta \rightarrow \phi \rightarrow \psi \\
 \epsilon = \theta \rightarrow \phi \\
 \zeta = \theta \rightarrow \psi \\
 \eta = (\psi \rightarrow \theta) \rightarrow (\psi \rightarrow \theta) \\
 \iota = \phi \rightarrow \psi \\
 \mu = \phi \rightarrow \zeta \\
 \nu = \theta \rightarrow \epsilon
 \end{array}$$

to produce six raw terms and six types.

We proceed by showing the existence of a proof tree whose final line is the appropriate judgment.

(i) Using the context

$$\Gamma = x : \theta$$

we have

$$\frac{\Gamma \vdash x : \theta}{\vdash \mathbf{I}_\lambda : \iota} \text{ (Introduction)}$$

to give the required result.

(ii) Using the context

$$\Gamma = x : \theta, y : \phi$$

we have

$$\frac{\Gamma \vdash x : \theta}{x : \theta \vdash (\lambda y : \phi . x) : \phi \rightarrow \theta} \text{ (Introduction)} \\
 \frac{x : \theta \vdash (\lambda y : \phi . x) : \phi \rightarrow \theta}{\vdash \mathbf{K}_\lambda : \kappa} \text{ (Introduction)}$$

to give the required result.

(iii) Using the context

$$\Gamma = x : \theta \rightarrow \phi \rightarrow \psi, y : \theta \rightarrow \phi, z : \theta$$

we have

$$\frac{\frac{\frac{\Gamma \vdash x : \theta \rightarrow \phi \rightarrow \psi \quad \Gamma \vdash z : \theta}{\Gamma \vdash (xz) : \phi \rightarrow \psi} \quad \frac{\Gamma \vdash y : \theta \rightarrow \phi \quad \Gamma \vdash z : \theta}{\Gamma \vdash (yz) : \phi}}{\Gamma \vdash .(xz)(yz) : \psi}}{x : \theta \rightarrow \phi \rightarrow \psi, y : \theta \rightarrow \phi \vdash \lambda z : \theta. (xz)(yz) : \theta \rightarrow \psi}}{x : \theta \rightarrow \phi \rightarrow \psi \vdash \lambda y : \theta \rightarrow \phi, \lambda z : \theta. (xz)(yz) : (\theta \rightarrow \phi) \rightarrow (\theta \rightarrow \psi)} \vdash S_\lambda : \sigma$$

to give the required result.

(iv) Using the context

$$\Gamma = x : \theta \rightarrow \phi, y : \psi \rightarrow \theta, z : \psi$$

we have

$$\frac{\frac{\frac{\Gamma \vdash x : \theta \rightarrow \phi}{\Gamma \vdash x(yz) : \phi} \quad \frac{\Gamma \vdash y : \psi \rightarrow \theta \quad \Gamma \vdash z : \psi}{\Gamma \vdash (yz) : \theta}}{x : \theta \rightarrow \phi, y : \psi \rightarrow \theta \vdash \lambda z : \psi. x(yz) : \psi \rightarrow \phi}}{x : \theta \rightarrow \phi \vdash \lambda y : \psi \rightarrow \theta, \lambda z : \psi. x(yz) : (\psi \rightarrow \theta) \rightarrow (\psi \rightarrow \phi)} \vdash B_\lambda : \beta$$

to give the required result.

(v) Using the context

$$\Gamma = x : \theta \rightarrow (\phi \rightarrow \psi), y : \phi, z : \theta$$

we have

$$\frac{\frac{\frac{\Gamma \vdash x : \theta \rightarrow (\phi \rightarrow \psi) \quad \Gamma \vdash z : \theta}{\Gamma \vdash (xz) : \phi \rightarrow \psi} \quad \Gamma \vdash y : \phi}{\Gamma \vdash xzy : \psi}}{x : \theta \rightarrow (\phi \rightarrow \psi), y : \phi \vdash \lambda z : \theta. xzy : \theta \rightarrow \psi}}{x : \theta \rightarrow (\phi \rightarrow \psi) \vdash \lambda y : \phi, \lambda z : \theta. xzy : \phi \rightarrow (\theta \rightarrow \psi)} \vdash C_\lambda : \gamma$$

to give the required result.

(vi) Using the context

$$\Gamma = x : \theta \rightarrow (\theta \rightarrow \phi), y : \theta$$

we have

$$\frac{\frac{\frac{\Gamma \vdash x : \theta \rightarrow (\theta \rightarrow \phi) \quad \Gamma \vdash y : \theta}{\Gamma \vdash xy : \theta \rightarrow \phi} \quad \Gamma \vdash y : \theta}{\Gamma \vdash xyy : \phi}}{x : \theta \rightarrow (\theta \rightarrow \phi) \vdash \lambda y : \theta . xyy : \theta \rightarrow \phi}}{\vdash D_\lambda : \delta}$$

to give them required result. ■

Thus we now have some examples of how to generate derivation trees.

4.5 Currying and Uncurrying

We need to consider how it is possible to go from types of the form

$$\sigma \rightarrow \rho \rightarrow \tau$$

to

$$\sigma \times \rho \rightarrow \tau$$

and back again.

This is done using a technical result called currying and we go in the other direction using a process called uncurrying.

This ideas allow us to ‘move’ between a calculus of just arrow types to one with product types. However it is not as straightforward as that since currying or uncurrying is not always possible. All the the systems used in this dissertation will allow currying and uncurrying without any of the more technical problems surrounding their use arising.

4.9 DEFINITION.

Given a function

$$f : \sigma \rightarrow \rho \rightarrow \tau$$

we can produce another function

$$\widehat{f} : \sigma \times \rho \rightarrow \tau$$

which we will call the *curried form* of f .

Given a function

$$g : \pi \times \beta \rightarrow \alpha$$

we can produce another function

$$\widetilde{g} : \pi \rightarrow \beta \rightarrow \alpha$$

which we will call the *uncurried form* of g . ■

These functions are distinguished in the definition since they are technically different functions however for all the functions we will look at in this dissertation we will treat them as being the same function and so will not make this distinction again.

4.6 Semantics of the Category of Sets

We need to consider the semantics of the systems we have considered in this dissertation. The natural home for these semantics is *Set* which we define below.

4.10 DEFINITION.

Set is the category whose objects are sets and whose arrows are functions. ■

This is a very simple category and will be sufficient for us to do semantics in since it is cartesian closed.

Given a derivation

$$(\nabla) \quad \Gamma \vdash t : \tau$$

we want to be able to think of this as something in *Set*. We introduce the following notation to show that we are thinking in *Set*.

The above derivation becomes

$$[[\Gamma]] \xrightarrow{[[\nabla]]} [[\tau]]$$

which shows are working in **Set**.

This notation illustrates the fact that this depends on the derivation ∇ .

We want to find something in **Set** which corresponds to a type in our system. Since types are defined recursively we need to have rules within **Set** which allow us to manipulate our types in **Set**.

The only objects which exist in **Set** are sets and so we define each type τ to a set $\llbracket \tau \rrbracket$. We find that arrow types correspond to the set of functions between sets and product types correspond to products of sets.

We have the following definition which should be compared to Definition 4.1.

4.11 DEFINITION.

Given a type τ we associate it with a set in **Set**, $\llbracket \tau \rrbracket$. If

$$\tau = \sigma \rightarrow \rho$$

then we associate it with

$$\llbracket \tau \rrbracket = \llbracket \sigma \rightarrow \rho \rrbracket = \llbracket \sigma \rrbracket \rightsquigarrow \llbracket \rho \rrbracket$$

where \rightsquigarrow is the function going from $\llbracket \sigma \rrbracket$ to $\llbracket \rho \rrbracket$. If

$$\tau = \sigma \times \rho$$

then we associate it with

$$\llbracket \tau \rrbracket = \llbracket \sigma \times \rho \rrbracket = \llbracket \sigma \rrbracket \boxtimes \llbracket \rho \rrbracket$$

where \boxtimes indicates a product being taken. ■

Thus we have a translation of types from our calculus into **Set**. We now need to work out what happens to the typing discipline.

Given a context

$$\Gamma = x_1 : X_1, \dots, x_n : X_n$$

we associate each of the types

$$X_1, \dots, X_n$$

to sets in *Set*.

$$[[X_1], \dots, [X_n]]$$

We associate the empty context with the final initial object $\mathbf{1}$ so that we have

$$[[\]] = \mathbf{1}$$

which allows us to see that the context Γ now becomes the following.

$$[[\Gamma]] = [[X_1]] \boxtimes \dots \boxtimes [[X_n]]$$

We now see how the rules for creating a derivation translate into *Set*, thus allowing us to build derivations semantically.

4.12 DEFINITION.

We have the following rules

$$\begin{array}{l}
 \text{(Weakening)} \quad \frac{[[\Gamma]] \xrightarrow{[\nabla]} [[\tau]]}{[[\Gamma]] \boxtimes [[\sigma]] \xrightarrow{\text{left}} [[\Gamma]] \xrightarrow{[\nabla]} [[\tau]]} \quad \text{Left proj } [[\Gamma]] \boxtimes [[\sigma]] \\
 \text{(Introduction)} \quad \frac{[[\Gamma]] \boxtimes [[\sigma]] \xrightarrow{\nabla} [[\rho]]}{[[\Gamma]] \xrightarrow{\tilde{\nabla}} [[\sigma]] \rightsquigarrow [[\rho]]} \quad \text{Currying} \\
 \text{(Elimination)} \quad \frac{[[\Gamma]] \xrightarrow{\nabla} [[\pi]] \rightsquigarrow [[\tau]] \quad [[\Gamma]] \xrightarrow{\square} [[\pi]]}{[[\Gamma]] \xrightarrow{\langle \nabla, \square \rangle} ([[\pi] \rightsquigarrow [\tau]]) \boxtimes [[\pi]] \xrightarrow{\text{eval}} [[\tau]]}
 \end{array}$$

where $\tilde{\nabla}$ is the curried version of ∇ . ■

We have the following theorem which shows us that derivations and reductions are well defined in this semantics.

4.13 THEOREM.

If

$$(\nabla) \quad \Gamma \vdash t : \tau$$

and

$$t \gg t'$$

then there there exists a derivation

$$(\nabla') \quad \Gamma \vdash t' : \tau$$

in the semantics.

It is not necessary to prove this result here, it is enough to know that it can be proven.

Chapter 5

Using the Syntax

In the previous chapter we set up all the syntax required for this dissertation. Here we will develop the algorithms necessary to carry out calculations in this system.

In the first section we will look at how we use our system to get at the natural numbers. We will find that all the natural numbers can be captured within our system.

In Section 5.2 we will show how substitution is carried out within our system. We will only look at the simplest algorithm for carrying this out since it is not within the aims of this dissertation to discuss the efficiency of the various algorithms. A reasonably thorough account can be found in [20] in Chapter 5.

We follow this section with a brief discussion of α -equivalence. This formalises the intuitive notion of ‘dummy’ variables within a system. We do not apply the results of α -equivalence to other sections since it is a rather tedious exercise in equivalence classes. However we do assure the reader that it is possible and hint about how to go about this.

We then move on to look at the computation mechanism in our system. This is where the real power of the system lies since this allows us to simulate functions within our system as well as carry out various computations.

Section 5.5 briefly looks at the concept of normalisation. This essentially formulates and answers the question, ‘Are our computations well-defined?’. As we shall see the answer is yes, but we will not prove the result.

Next we look at the behaviour of pairing gadgets within our system. We look at the ways of simulating pairing gadgets and discuss whether or not this is possible within a typed system.

Finally we conclude this chapter by setting up the various systems we will need throughout the dissertation. In fact we will set up one big system and let all the other systems we need be subsystems.

5.1 Capturing the Natural Numbers

Using the constants 0 and Suc we can capture the natural numbers in the following canonical way.

5.1 DEFINITION.

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

$$\ulcorner m \urcorner = \text{Suc}^m 0$$

to obtain the numeral for m . ■

In other words we have that

$$\ulcorner 0 \urcorner = 0 \quad \ulcorner m + 1 \urcorner = \text{Suc} \ulcorner m \urcorner$$

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

We introduce the following convention for the successor function.

5.2 CONVENTION.

The operator $(\cdot)'$ will be used to represent the successor function when it is clear to do so. Thus $x' = \text{Suc}x$. ■

5.2 Substitution

The idea of substitution is very important in a lambda calculus and it can be very easy to make mistakes when doing a substitution. The ‘obvious’ textual substitution does not work since we have to worry about identifier capture. For this reason we need a clearly defined algorithm for substitution which is given below.

5.3 DEFINITION.

We say that s is **substituted** for x in t and that the following term

$$t[x := s]$$

arises 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.

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

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. ■

This is the simplest form of the substitution algorithm which works. It can get quite messy computationally and if we were concerned with such issues another more slicker algorithm would be used.

5.3 α -equivalence

This section formalises the notion of when two terms of equivalent. Intuitively the two terms

$$(\lambda y . yx) \quad (\lambda z . zx)$$

seem to be the same since we appear to have just renamed y as z . This is formalised in the following definition.

5.4 DEFINITION.

Two terms are **α -equivalent** if they agree except for the choice of bound identifiers, eg the terms

$$(\lambda y . yx) \quad (\lambda z . zx)$$

are α -equivalent. ■

We will not in general distinguish between α -equivalent terms since they do the same thing semantically. The terms given above are α -equivalent terms and it is clear that as long as x, y, z are pairwise distinct identifiers they capture the same thing. It is possible to formally set up α -equivalence in terms of equivalent classes and then choose a representative from each but this is tedious and a routine exercise for most mathematicians and so is left as an exercise if the reader is really concerned about the full formality. This would also require the the substitution algorithm mentioned in Section 5.2 to be revised slightly but again it is not entirely taxing. There are ways to avoid α -equivalence for example Bourbaki, de Bruijn and dual de Bruijn indexes. However α -equivalence seems to be a much more natural way of doing things and does not require any algorithms involving numbers which seem to confuse issues and make things rather messy. Thus they will not be used here nor mentioned again.

5.4 Computation Mechanism

We need to be able to simulate functions in the lambda calculus and for that reason we introduce a notion which allows us to represent an evaluation of a function. We thus introduce a reduction.

5.5 DEFINITION.

There are 10 kinds of 1-step reductions.

- For each redex $(\lambda x : \sigma . r) s$ there is a redex removal

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

using the immediate reduct. More precisely, an abstraction redex is converted into its abstraction reduct.

- For terms l, r we have

$$\text{Left}_{\bullet}(\text{Pair}_{\bullet}lr) \triangleright l \qquad \text{Right}_{\bullet}(\text{Pair}_{\bullet}lr) \triangleright r$$

so a pairing redex is converted into its pairing reduct.

- For terms r, s, t we have

$$\mathbf{It}_\bullet \ulcorner 0 \urcorner ts \triangleright s \qquad \mathbf{It}_\bullet (\text{Suc}r) ts \triangleright t (\mathbf{It}_\bullet rts)$$

so an iterator redex is converted into its iterator reduct.

- For terms r, s, t we have

$$\mathbf{Rec}_\bullet ts \ulcorner 0 \urcorner \triangleright t \qquad \mathbf{Rec}_\bullet ts (\text{Suc}r) \triangleright s (\mathbf{Rec}_\bullet tsr) r$$

so a recursion redex is converted into its recursion reduct.

- For terms x, y, z we have

$$\begin{aligned} \mathbf{I}x &\triangleright x \\ \mathbf{K}yx &\triangleright y \\ \mathbf{S}zyx &\triangleright (zx)(yx) \\ \mathbf{B}zyx &\triangleright z(yx) \\ \mathbf{C}zyx &\triangleright zxy \\ \mathbf{D}yx &\triangleright yxx \end{aligned}$$

so each combinator redex reduces to a combinator reduct.

There are no other 1-step reductions. ■

For the pairing reduction I have not included the 1-step reduction

$$\mathbf{Pair} (\text{Left}p) (\text{Right}p) \triangleright p$$

which is known as ‘surjective pairing’ since it causes a lot of problems with the reduction properties. In fact it stops them being well-defined which is not something we want. This is used by Thibault in [21] as the only reduction property of the pairing gadgets.

The following example given in [4] shows why surjective pairing causes problems in the untyped λ -calculus. It is not possible to type this example in our system since it involves a Turing fixed point. It is possible to type fixed points, so this example can be used in a typed system however the typing of fixed points is beyond this dissertation.

5.6 EXAMPLE.

Assume that we have surjective pairing in addition to our other 1-step reductions for the pairing gadgets. So that

$$\text{Pair}(\text{Left}p)(\text{Right}p) \triangleright p$$

holds.

Now consider the term

$$L = YN$$

where

$$N = YV$$

and

$$V = \lambda x, y. \text{Pair}(\text{Left}(Ey))(\text{Right}(E(xy)))$$

where Y is the Turing fixed point and so

$$Y = \Omega\Omega \quad \Omega = \lambda x, y. y(xxy)$$

holds.

The first result we will prove is the following result for Turing fixed points.

5.7 LEMMA.

For each term f we have that

$$Yf \triangleright f(Yf) \tag{5.1}$$

holds.

Proof.

We notice that

$$Y = \Omega\Omega \triangleright \lambda y. y(\Omega\Omega y) \triangleright \lambda y. y(Yy)$$

holds. ■

As a special case of this we observe that

$$L \triangleright NL$$

holds.

We also notice that for any M

$$NM \triangleright\triangleright VNM$$

which is just another case of (5.1).

VNM reduces yet further to give the following.

$$NM \triangleright\triangleright \text{Pair}(\text{Left}(EM))(\text{Right}(E(NM))) \quad (5.2)$$

Now we have that

$$L \triangleright\triangleright NL$$

which is a special case of (5.1).

We also have that

$$NL \triangleright\triangleright \text{Pair}(\text{Left}(EL))(\text{Right}(E(NL)))$$

which follows from (5.2).

This reduces yet further to give the following reduction.

$$\text{Pair}(\text{Left}(EL))(\text{Right}(E(NL))) \triangleright\triangleright \text{Pair}(\text{Left}(E(NL)))(\text{Right}(E(NL)))$$

We can now apply surjective pairing and so we have shown that

$$L \triangleright\triangleright E(NL)$$

and we let $A = E(NL)$.

We now take a different reduction path for L and show

$$L \triangleright\triangleright NL \triangleright\triangleright NA$$

is an alternate reduction path.

We already know that

$$L \triangleright\triangleright A$$

and

$$L \triangleright\triangleright NL$$

hold so the result follows.

To show that surjective pairing is not well-defined we need to show that there is not a common reduct of both A and NA .

We do this by noting that any term J can be written uniquely as

$$J = E(E(\dots(EJ')\dots))$$

where J' is not of the form EJ'' . We call the E 's J 's head E 's.

Assume that A and NA have a common reduct. We now analyse how A and NA get there.

Firstly we reduce N and A independently.

We find that

$$\begin{aligned} N &\triangleright (\lambda z . z(Yz))V \\ &\triangleright V(VYV) \\ &\triangleright V(\lambda y . \text{Pair}(\text{Left}(Ey))(\text{Right}(E((YV)y)))) = VN' \\ &\triangleright \lambda y . \text{Pair}(\text{Left}(Ey))(\text{Right}(E(N'y))) \\ &\triangleright \lambda y . \text{Pair}(\text{Left}(Ey))(\text{Right}(E(N''))) \end{aligned}$$

where $N'' = \text{Pair}(\text{Left}(Ey))(\text{Right}(E(Ny)))$.

We can apply surjective pairing here only if $N'y \triangleright y$, which implies that $Ny \triangleright y$. By (5.8) we have

$$Ny \triangleright \text{Pair}(\text{Left}(Ey))(\text{Right}(E(Ny)))$$

which if $Ny \triangleright y$ will reduce even further to give

$$Ny \triangleright Ey$$

which is a counter-example to confluence. Thus we may assume that the reduction of N stops here.

Applying A to N we get the following reduction.

$$\begin{aligned} NA &\triangleright \text{Pair}(\text{Left}(EA))(\text{Right}(E(N''[A/y]))) \\ &\triangleright \text{Pair}(\text{Left}(EA))(\text{Right}(E(E(NA)))) \text{ since } L \triangleright E(NL) \text{ and surjective pairing} \\ &\triangleright \text{Pair}(\text{Left}(E(E(NA))))(\text{Right}(E(E(NA)))) \text{ since } L \triangleright E(NL) \\ &\triangleright E(E(NA)) \text{ by surjective pairing} \end{aligned}$$

Note that $E(\text{NA})$ has less head E 's than $E(E(\text{NA}))$ since all of the head E 's of $E(E(\text{NA}))$ are left untouched by any reduction from $E(E(\text{NA}))$. Also we have that $A \triangleright E(\text{NA})$ and $\text{NA} \triangleright E(\text{NA})$ but $E(\text{NA})$ has less head E 's than $E(E(\text{NA}))$ which is a contradiction. ■

As can be seen by the above example surjective pairing causes reduction to not be well-defined in the untyped λ -calculus. So we will not allow it in our system.

These 1-step reductions can be combined in the following way to give rise to the following relation.

5.8 DEFINITION.

For terms t^- and t^+ we write

$$t^- \triangleright t^+$$

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 selected redex within t^- by the corresponding 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 $\triangleright\triangleright$ be the reflexive closure of \triangleright . Thus for terms t^-, t^+ we have

$$t^- \triangleright\triangleright t^+$$

when either t^- and t^+ are the same term (up to α -equivalence) or $t^- \triangleright t^+$. ■

One way of showing that an instance of $t^- \triangleright\triangleright t^+$ occurs is to write it out explicitly in the following chain

$$t^- = t_0 \triangleright t_1 \triangleright \dots \triangleright t_{l-1} \triangleright t_l = t^+$$

of redex removals. So this is an instance of 1-step reductions embedded in a term.

The notation of $\triangleright\triangleright$ needs to be formalised in the following way.

5.9 DEFINITION.

The reduction relation $\triangleright\triangleright$ is the least relation on terms generated by the following rules.

$$\begin{array}{c}
\text{Leaf} \quad \frac{\text{Redex} \triangleright \text{Reduct}}{\text{Redex} \triangleright\triangleright \text{Reduct}} \\
\\
\text{Application} \quad \frac{q^- \triangleright\triangleright q^+}{q^- p \triangleright\triangleright q^+ p} \quad \frac{p^- \triangleright\triangleright p^+}{qp^- \triangleright\triangleright qp^+} \\
\\
\text{Abstraction} \quad \frac{r^- \triangleright\triangleright r^+}{(\lambda x . r^-) \triangleright\triangleright (\lambda x . r^+)} \\
\\
\text{Composition} \quad \frac{t^- \triangleright\triangleright t^0 \quad t^0 \triangleright\triangleright t^+}{t^- \triangleright\triangleright t^+}
\end{array}$$

That is, each instance

$$t^- \triangleright\triangleright t^+$$

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

We often want to be able to count the number of reduction steps involved in a particular reduction and the symbol $\triangleright\triangleright$ hides the number of steps since it stands for any finite reduction. Thus we introduce the following convention.

5.10 CONVENTION.

Given a 1-step reduction $t_1 \triangleright\triangleright t_2$ we use the notation $t_1 \triangleright t_2$ to emphasise that we have only done a single reduction. ■

This convention will mainly be used when we wish to make explicit the use of a 1-step reduction, formally we should always use $\triangleright\triangleright$ for any finite reduction and so to formalise where we have used \triangleright just replace it by $\triangleright\triangleright$.

We wish to be able to ‘capture’ functions within the calculus. Informally we wish that if we input numeric data the returned value should be the same as we would have gotten from evaluating the function outside the calculus. Essentially we calculate the function within the calculus and outside the calculus and check if the results match. The following definition makes this formal.

5.11 DEFINITION.

Let f be an l -placed function on \mathbb{N} and let $\ulcorner f \urcorner$ be a λ -term with

$$\vdash \ulcorner f \urcorner : \mathcal{N} \rightarrow \dots \rightarrow \mathcal{N} \rightarrow \mathcal{N}$$

where there are $l + 1$ occurrences of \mathcal{N} . We say that f **represents** f (in the calculus) if the reduction

$$\ulcorner f \urcorner \ulcorner m_l \urcorner \dots \ulcorner m_1 \urcorner \Downarrow \Downarrow \ulcorner m_0 \urcorner$$

holds for all $m_0, m_1, \dots, m_l \in \mathbb{N}$ with $f(m_l, \dots, m_1) = m_0$. ■

The following example shows how we ‘capture’ successor, addition and multiplication in our calculus.

5.12 EXAMPLE. We have three terms

$$S_\zeta = \lambda u : \zeta'', y : \zeta', x : \zeta . y(uyx)$$

$$A_\zeta = \lambda v, u : \zeta'', y : \zeta', x : \zeta . (vy)(uyx)$$

$$M_\zeta = \lambda v, u : \zeta'', y : \zeta', x : \zeta . v(uy)x$$

which are well formed in our λ -calculus.

We will show that these terms represent successor, addition and multiplication respectively. To do this we prove the following.

$$S_\zeta \overline{m_\zeta} \Downarrow \overline{(m+1)_\zeta}$$

$$A_\zeta \overline{n_\zeta} \overline{m_\zeta} \Downarrow \overline{(m+n)_\zeta}$$

$$M_\zeta \overline{n_\zeta} \overline{n_\zeta} \Downarrow \overline{(m \times n)_\zeta}$$

For the first term we have

$$\begin{aligned} \lambda u : \zeta'', y : \zeta', x : \zeta . y(uyx) \overline{m_\zeta} &\Downarrow \lambda y : \zeta', x : \zeta . y(\overline{m_\zeta}yx) \\ &\Downarrow \lambda y : \zeta', x : \zeta . y(y^m x) \\ &= \lambda y : \zeta', x : \zeta . y^{m'} x \\ &= \overline{(m+1)_\zeta} \end{aligned}$$

as it's reduction path. Note that we have not shown that the equality holds between the second and third line. It is not as obvious as it appears and it requires an induction

argument. However rest assured it does hold. We will find something similar in each of the following reductions. The remaining two terms reduce as follows.

$$\begin{aligned}
\lambda v, u : \zeta'', y : \zeta', x : \zeta . (vy)(uyx)\overline{n_\zeta m_\zeta} &\triangleright \lambda u : \zeta'', y : \zeta', x : \zeta . (\overline{n_\zeta y})(uyx) \\
&\triangleright \lambda y : \zeta', x : \zeta . (\overline{n_\zeta y})(\overline{m_\zeta yx}) \\
&\triangleright \lambda y : \zeta', x : \zeta . y^n(y^m x) \\
&= \lambda y : \zeta', x : \zeta . y^{n+m}x \\
\\
\lambda v, u : \zeta'', y : \zeta', x : \zeta . v(uy)x\overline{n_\zeta m_\zeta} &\triangleright \lambda u : \zeta'', y : \zeta', x : \zeta . \overline{n_\zeta}(uy)x\overline{m_\zeta} \\
&\triangleright \lambda u : \zeta'', y : \zeta', x : \zeta . (uy)^n x\overline{m_\zeta} \\
&\triangleright \lambda y : \zeta', x : \zeta . (\overline{m_\zeta y})^n x \\
&\triangleright \lambda y : \zeta', x : \zeta . (y^m)^n x \\
&\triangleright \lambda y : \zeta', x : \zeta . y^{m \times n} x
\end{aligned}$$

This we find that the above terms simulate successor, addition and multiplication in our λ -calculus. ■

5.5 Normalisation

When we consider the reduction mechanism discussed in Section 5.4 there are a few questions that come to mind.

The first and perhaps most obvious of these is does it stop? If so do we distinguish terms when there are no more reductions possible?

The answer to the first question is that no, it is possible for a reduction to carry on for ever. However most of the ones we will consider will terminate. The answer to the second question is yes and motivates the following definition.

5.13 DEFINITION.

A term is **normal** if no further reductions are possible. ■

Below we examine a few examples of normal terms and terms which are not normal. We find that in general if we were to write down a random term it is most likely to not be normal.

5.14 EXAMPLE.

The term

$$(x(y\lambda u : P.(xu)u))(y\lambda u : P.(xu)u)$$

is normal. It is clear that it cannot be reduced any further.

The term

$$\lambda u : P.(xu)u(y\lambda u.(xu)u)$$

is not normal. It does however reduce to a normal term. We reduce it to its λ reduct

$$(x(y\lambda u : P.(xu)u))(y\lambda u : P.(xu)u)$$

which we have just shown to be normal. ■

The next question that comes to mind when considering reduction is whether or not two reduction paths lead to the same normal term. So if we are given a reduction

$$x \triangleright y$$

and another reduction

$$x \triangleright z$$

whether or not there are reductions

$$y \triangleright\triangleright m \quad z \triangleright\triangleright m$$

so that we get reduction to a common term. This leads to the following definition.

5.15 DEFINITION.

A relation, \sim , is **confluent** if for each divergent wedge

$$t_0 \sim t_1 \quad t_0 \sim t_2$$

from a common source t_0 , there is a convergent wedge

$$t_1 \sim t_3 \quad t_2 \sim t_3$$

to a common target t_3 . ■

We find that the relation $\triangleright\triangleright$ is confluent but that we need the reflexive property to guarantee this.

Since we are working within a typed λ -calculus, we are interested in what happens to the context when we go to this common term. We are also interested in whether or not this common term can be derived in the calculus.

Thus we have the following theorem, which we shall not prove since it is rather tedious and similar proofs can be found in [5] and [22].

5.16 THEOREM.

For each derivation

$$(\nabla) \quad \Gamma \vdash t : \tau$$

there is an essentially unique normal term t^ such that*

$$t \triangleright\triangleright t^*$$

holds. Furthermore there is a derivation

$$(\nabla^*) \quad \Gamma \vdash t^* : \tau$$

in the same context.

Note that here by essentially unique we mean up to α -equivalence.

Here is an example of some derivations and how they lead to normal terms but go against the idea of normal terms having the ‘best’ derivation.

5.17 EXAMPLE.

Let P, Q be types and let

$$X = P \rightarrow (P \rightarrow Q) \quad Y = (P \rightarrow Q) \rightarrow P$$

and

$$\Gamma = X, Y$$

to obtain two more types X, Y and a context Γ . From here we move on by deriving the necessary derivations.

Firstly we derive the derivation

$$(\nabla) \quad \Gamma \vdash t : P \rightarrow Q$$

from terms $x : X, y : Y$.

$$\frac{\frac{\Gamma, u : P \vdash x : X \quad \Gamma, u : P \vdash u : P}{\Gamma, u : P \vdash xu : P \rightarrow Q} \quad \Gamma, u : P \vdash u : P}{\Gamma, u : P \vdash (xu)u : Q} \\ \Gamma \vdash (\lambda u : P. (xu)u) : P \rightarrow Q$$

We now use the derivation (∇) to get a derivation for a term with type Q . There are two derivations which will get us this term, the first being.

$$\frac{\frac{\nabla}{\Gamma \vdash t : P \rightarrow Q} \quad \frac{\Gamma \vdash y : Y \quad \Gamma \vdash t : P \rightarrow Q}{\Gamma \vdash yt : P}}{\Gamma \vdash t(yt) : Q}$$

So written out fully we have a derivation for the term

$$\lambda u : P. (xu)u(y\lambda u : P. (xu)u) : Q$$

which is not normal.

We can derive a slightly different term with type Q in the following way.

$$\frac{\frac{\Gamma \vdash x : X \quad \frac{\Gamma \vdash y : Y \quad \Gamma \vdash t : P \rightarrow Q}{\Gamma \vdash yt : P}}{\Gamma \vdash x(yt) : P \rightarrow Q} \quad \Gamma \vdash yt : P}{\Gamma \vdash (x(yt))(yt) : Q}$$

So written out fully we have a derivation for the term

$$(x(y\lambda u : P. (xu)u))(y\lambda u : P. (xu)u)$$

which is normal.

Looking at the derivation trees we see that the first derivation tree is ‘better’ since the second derives a term of type $P \rightarrow Q$ in the derivation which we already have a much shorter derivation for.

However the ‘worse’ derivation tree produces a normal term which is considered a ‘better’ term.

Also note that as we showed in Example 5.14 the normal term is the λ reduct of the other term so they are linked and this corresponds to Theorem 5.16 ■

The moral of the above example is that a normal term does not always have the ‘best’ derivation tree.

5.6 Capturing Pairing Gadgets

If we do not have the pairing gadgets in our calculus is it then possible to recreate them?

The answer to this is sort of. As the following example shows it is possible to create terms which behave like pairing gadgets although they do not have product types involved in their typing.

5.18 EXAMPLE.

We find that

$$P = \lambda l, r, z. zlr \quad L = \lambda w. wt \quad R = \lambda w. wf$$

where

$$t = \lambda y, x. y \quad f = \lambda y, x. x$$

are the auxiliary terms form pairing gadgets.

$$L(Plr) \triangleright Plrt \triangleright tlr \triangleright l$$

$$R(Plr) \triangleright Plrf \triangleright flr \triangleright r$$

Giving us constructed pairing gadgets in the untyped λ -calculus. ■

If we try to type the above example we run into problems.

We begin by deriving t and f . Let $\Gamma = y : \sigma, x : \rho$ where σ and ρ are arbitrary types.

We thus obtain

$$\frac{\Gamma \vdash y : \sigma}{y : \sigma \vdash \lambda x : \rho. y : \rho \rightarrow \sigma} \quad \frac{}{\vdash \lambda y : \sigma, x : \rho. y : \sigma \rightarrow \rho \rightarrow \sigma}$$

as the derivation tree for t and

$$\frac{\frac{\Gamma \vdash x : \rho}{y : \sigma \vdash \lambda x : \rho. x : \rho \rightarrow \rho}}{\vdash \lambda y : \sigma, x : \rho. x : \rho \rightarrow \rho \rightarrow \sigma}$$

as the derivation tree for f .

We now derive L. We must have $w : \sigma \rightarrow \rho \rightarrow \sigma \rightarrow \epsilon$ where ϵ is an arbitrary type. Thus we obtain the following derivation tree.

$$\frac{w : \sigma \rightarrow \rho \rightarrow \sigma \rightarrow \epsilon \vdash w : \sigma \rightarrow \rho \rightarrow \sigma \rightarrow \epsilon \quad w : \sigma \rightarrow \rho \rightarrow \sigma \rightarrow \epsilon \vdash f : \sigma \rightarrow \rho \rightarrow \sigma}{\frac{w : \sigma \rightarrow \rho \rightarrow \sigma \rightarrow \epsilon \vdash wf : \epsilon}{\vdash \lambda w : \sigma \rightarrow \rho \rightarrow \sigma \rightarrow \epsilon. wf : \sigma \rightarrow \rho \rightarrow \sigma \rightarrow \epsilon \rightarrow \epsilon}}$$

We now do the same for R and find that it has type $\rho \rightarrow \rho \rightarrow \sigma \rightarrow \kappa$. However we now have problems since we want to be able to apply a typed form of *Plr* to both R and L and the only way this is possible is if their types agree. We can achieve this since the types were arbitrary, so we let $\kappa = \epsilon$ and $\sigma = \rho$. However we are now pairing objects with the same type and so our pairing gadget is not very good.

Thus we have shown that if we try to create a pairing gadget in a typed system we can only create one which pairs objects of the same type which is not a very useful pairing gadget.

5.7 Defining the System

After all our hard work we now have set up a system to work in and we will call our system λBIG . We are interested in the following subsystems of λBIG ; $\lambda G[\rightarrow; \text{Rec}]$, $\lambda G[\rightarrow, \times; \text{lt}]$, $CL[\rightarrow; \text{Rec}]$ and $CL[\rightarrow, \times; \text{lt}]$.

- $\lambda G[\rightarrow; \text{Rec}]$ is the subsystem of λBIG which does not have product types, the iterator, pairing gadgets or any of the combinators.
- $\lambda G[\rightarrow, \times; \text{lt}]$ is the subsystem of λBIG which does not have the recursion operator or the combinators.

- $CL[\rightarrow; \text{Rec}]$ is the subsystem of λBIG which does not have product types, the iterator or pairing gadgets.
- $CL[\rightarrow, \times; \text{lt}]$ is the subsystem of λBIG which does not have the recursion operator.

These systems all represent Gödel's T. Gödel himself used combinators and recursion operators and his original system would be similar to the system $CL[\rightarrow; \text{Rec}]$ and his original account is in [6]. However he did not use judgements and there appears to be no accounts in the literature of setting up a system equivalent to Gödel's T using judgements. This system is also used by Longley in [17].

The system $CL[\rightarrow; \text{Rec}]$ is used by Grzegorzcyk in [7].

Chapter 6

Computation in the Syntax

Here we look at computation in general and show how we can go to and from a typed system. This is quite a useful trick since sometimes we find that it is easier to work in an untyped system and then convert back. The main use for this transfer between systems is within programming languages. It is easier to carry out computations within an untyped system but there is more scope for error in the programming in an untyped system. So the programming is carried out in a typed system before it is converted to an untyped system where the calculation is carried out. The result is then returned to a typed system.

6.1 Computation

We want a way to formalise our reduction calculations. This is done by introducing the notion of a computation. This allows us to grow a tree-like structure which allows us to analyse our reduction calculations in a more formal setting. We begin by defining a computation.

6.1 DEFINITION.

A Computation

$$(\square) \quad t^- \triangleright t^+$$

is a finite rooted tree of reductions. These reductions are grown according to the

Rule	Shape	Remarks
(Redex Reduction)	$\frac{t^- \triangleright t^+}{t^- \triangleright\triangleright t^+}$	$t^- = (\lambda y : \sigma . r)s$ $t^+ = r[y := s]$
(Axiom Reduction)	$\frac{t^- \triangleright t^+}{t^- \triangleright\triangleright t^+}$	1-step leaf
(Left Application)	$\frac{\begin{array}{c} \mathbf{q} \\ q^- \triangleright q^+ \end{array}}{q^- p \triangleright\triangleright q^+ p}$	
(Right Application)	$\frac{\begin{array}{c} \mathbf{p} \\ p^- \triangleright p^+ \end{array}}{qp^- \triangleright\triangleright qp^+}$	
(Abstraction)	$\frac{\begin{array}{c} \mathbf{r} \\ r^- \triangleright r^+ \\ t^- \triangleright\triangleright t^+ \end{array}}{t^- \triangleright\triangleright t^+}$	$t^- = (\lambda y : \sigma . r^-)$ $t^+ = (\lambda y : \sigma . r^+)$
(Transitive Composition)	$\frac{\begin{array}{c} \mathbf{l} \qquad \mathbf{r} \\ t^- \triangleright\triangleright t^0 \quad t^0 \triangleright\triangleright t^+ \\ t^- \triangleright\triangleright t^+ \end{array}}{t^- \triangleright\triangleright t^+}$	

Table 6.1: Computation Rules

rules of Table 6.1. Such a computation is said to **organise** the root reduction $t^- \triangleright\triangleright t^+$ where t^- is the **subject** and t^+ is the **object** of the computation. ■

Here is an example of a computation.

6.2 EXAMPLE.

For arbitrary θ, ψ, ϕ consider the following type.

$$\beta = (\psi \rightarrow \phi) \rightarrow (\theta \rightarrow \psi) \rightarrow (\theta \rightarrow \phi)$$

Also let

$$\rho = \theta \rightarrow \phi \quad \sigma = \theta \rightarrow \psi \quad \tau = \psi \rightarrow \phi \quad \nu = \sigma \rightarrow \rho$$

so that we obtain $\beta = \tau \rightarrow \theta \rightarrow \rho$. Also let

$$\begin{aligned} \kappa_2 &= \tau \rightarrow \theta \rightarrow \tau & \alpha &= \theta \rightarrow \tau & \sigma_2 &= \alpha \rightarrow \sigma \rightarrow \rho \\ \kappa_1 &= \sigma_2 \rightarrow \tau \rightarrow \sigma_2 & \mu &= \tau \rightarrow \sigma_2 & \sigma_1 &= \mu \rightarrow \kappa_2 \rightarrow \beta \end{aligned}$$

system when carrying out calculations and then return to a typed system once we have our result. To do this we would need to have an algorithm which allows us to remove types from a typed term and one which allows us to add types to an untyped term. We also need to prove that our calculations in the untyped system give the same result as when we carry out our calculations in the typed system.

To go from a typed system to an untyped system we just remove all the types. This is formally expressed as follows.

6.3 DEFINITION.

The type erasing operation from typed terms to untyped terms

$$(\cdot)^\epsilon : \lambda_{typed} \rightarrow \lambda_{untyped}$$

is generated by recursion using

- $(x : \sigma)^\epsilon = x$
- $Z_\bullet^\epsilon = Z$
- $(qp)^\epsilon = q^\epsilon p^\epsilon$
- $(\lambda x : \sigma . y)^\epsilon = \lambda x . r^\epsilon$

for each identifier x, y , type σ , constant Z_\bullet , and terms q, p . ■

We now prove the following theorem which shows that given a typed term, t^- and a reduction $t^- \triangleright t^+$ if we apply the above algorithm to obtain untyped terms and then carry out the reduction, the term we end up with can be retyped to obtain t^+ . Thus the algorithm to type an untyped term is embedded within the proof.

6.4 THEOREM.

Type erasure

$$(\square) \quad t^- : \sigma \triangleright t^+ : \rho \quad \longmapsto \quad (\square^\epsilon) \quad t^{-\epsilon} \triangleright t^{+\epsilon}$$

converts each typed computation \square into an untyped computation \square^ϵ . Furthermore, given an untyped computation

$$(\square^*) \quad t^{-\epsilon} \triangleright t^*$$

where the subject $t^{-\epsilon}$ is the erasure of a typed term t^- , there is a unique typed computation \square (as above) with $\square^\epsilon = \square^*$ and hence $t^* = t^{+\epsilon}$.

Proof.

We first show that type erasure passes through substitution.

$$(t[:= s])^\epsilon = t^\epsilon[x := s^\epsilon]$$

We proceed by induction over t . The only step which is not immediate is the abstraction step.

$$t = (\lambda y : \sigma . r)$$

Firstly we have

$$t[x := s] = \lambda v : \sigma . r'[x := s]$$

where $r' = r[y := v]$ for some suitable v . Hence

$$(t[x := s])^\epsilon = \lambda v : \sigma . (r'[x := s])^\epsilon = \lambda v : \sigma . r'^\epsilon[x := s^\epsilon]$$

where

$$(r')^\epsilon = (r[y := v])^\epsilon = r^\epsilon[y := v^\epsilon]r^\epsilon[y := v]$$

by two uses of the induction hypothesis. But notice that

$$t^\epsilon[x := s^\epsilon] = (\lambda y . r^\epsilon) = [x := s^\epsilon] = \lambda v . r^{\epsilon'}[x := s^\epsilon]$$

where

$$r^{\epsilon'} = r^\epsilon[y := v] = r'^\epsilon$$

provided the naming of y is done in the same way on both occasions. Thus

$$(t[x := s])^\epsilon = \lambda v . r'^\epsilon[x := s^\epsilon] = \lambda v . r^{\epsilon'}[x := s^\epsilon] = t^\epsilon[x := s^\epsilon]$$

as required.

Now given typed terms t^+, t^- and a typed 1-step reduction $t^- \triangleright t^+$ we can remove all the types. It is clear that we still have a 1-step reduction $t^{-\epsilon} \triangleright t^{+\epsilon}$, of the untyped terms $t^{-\epsilon}, t^{+\epsilon}$. Thus the full algorithm $\square \mapsto \square^\epsilon$ is immediate.

Conversely, given an untyped computation

$$(\square^*) \quad t^{-\epsilon} \triangleright\triangleright t^*$$

we need to produce a typed computation \square with $\square^\epsilon = \square^*$. Firstly just consider the constants and here we proceed by recursion over \square^* . We look at the base case.

$$t^{-\epsilon} = \mathsf{I}r^! \quad t^* = r^! \quad t^{-\epsilon} = \mathsf{K}s^!r^! \quad t^* = s^! \quad t^{-\epsilon} = \mathsf{S}t^!s^!r^! \quad t^* = (t^!r^!)(s^!r^!)$$

$$t^{-\epsilon} = \mathsf{B}t^!s^!r^! \quad t^* = t^!(s^!r^!) \quad t^{-\epsilon} = \mathsf{C}t^!s^!r^! \quad t^* = t^!r^!s^! \quad t^{-\epsilon} = \mathsf{D}s^!r^! \quad t^* = s^!r^!r^!$$

$$t^{-\epsilon} = \mathsf{Left}(\mathsf{Pair}s^!r^!) \quad t^* = s^! \quad t^{-\epsilon} = \mathsf{Right}(\mathsf{Pair}s^!r^!) \quad t^* = r^!$$

$$t^{-\epsilon} = \mathsf{It}\overline{0}^!s^!r^! \quad t^* = t^* = r^! \quad t^{-\epsilon} = \mathsf{It}(\mathsf{Suct}^!)s^!r^! \quad t^* = s^!(\mathsf{Its}^!t^!r^!)$$

$$t^{-\epsilon} = \mathsf{Rec}s^!r^!\overline{0}^! \quad t^* = t^* = s^! \quad t^{-\epsilon} = \mathsf{Rec}s^!r^!(\mathsf{Suct}^!) \quad t^* = r^!(\mathsf{Rec}s^!r^!t^!)t^!$$

All the cases are very similar and so we will only prove the S case.

From the shape of $t^{-\epsilon}$ and the way erasure works we have that

$$t^- = \mathsf{S}_\bullet tsr$$

for a unique typed combinator S_\bullet and unique typed terms t, s, r . Since

$$t^{-\epsilon} = (\mathsf{S}_\bullet tsr)^\epsilon = \mathsf{S}_\bullet^\epsilon t^\epsilon s^\epsilon r^\epsilon = \mathsf{S}t^\epsilon s^\epsilon r^\epsilon$$

we see that

$$t^\epsilon = t^! \quad s^\epsilon = t^! \quad r^\epsilon = r^!$$

hold. Let $t^+ = (tr)(sr)$ so that $t^{+\epsilon} = t^*$. There is a unique 1-step computation with t^- as subject, namely

$$(\square) \quad t^- \triangleright\triangleright t^+$$

and this satisfies $\square^\epsilon = \square^*$.

Now we look at the λ -terms and consider the base case. Firstly we look at an untyped redex removal. Thus

$$t^{-\epsilon} = (\lambda y. r^!)s^! \quad t^* = r^![y := s^!]$$

for some untyped terms $r^!, s^!$. From the way erasure works we must have

$$t^- = (\lambda y : \sigma . r) s$$

for some untyped terms r, s . Then $r^! = r^\epsilon$ and $s^! = s^\epsilon$. Let

$$t^+ = r[y := s]$$

so that

$$t^- \triangleright t^+$$

is the only possible redex removal with t^- as the subject. Also

$$t^{+\epsilon} = (r[y := s])^\epsilon = r^\epsilon[y := s^\epsilon] = t^*$$

so we have a typed computation \square with $\square^\epsilon = \square^*$.

These are all the base cases, we now need to prove the induction steps. The first of these is across left application.

Suppose \square^* is a left application where

$$q^! \triangleright q^* \quad t^{-\epsilon} = q^! p^! \quad t^* = q^* p^!$$

for some untyped terms $q^!, p^!, q^*$. From the shape of $t^{-\epsilon}$ and the way erasure works we have that $t^- = q^- p$ for some typed terms q^-, p . Then

$$t^{-\epsilon} = (q^- p)^\epsilon = q^{-\epsilon} p^\epsilon$$

so that $q^! = q^{-\epsilon}$ and $p^! = p^\epsilon$. This gives

$$q^{-\epsilon} \triangleright q^*$$

and hence, by the induction hypothesis, there is a unique typed computation

$$q^- \triangleright q^+$$

with $q^\epsilon = q^*$. With this q^+ let $t^+ = q^+ p$, so that

$$t^{+\epsilon} = q^{+\epsilon} p^\epsilon = q^* p^! = t^*$$

and hence \square is an example of the required kind of typed computation with $\square^\epsilon = \square^\epsilon$. From the shape of t^- this is the only possible example.

The step across right application is similar.

Consider the step across right application. Here \square^* is a right application where

$$r^! \triangleright r^* \quad t^{-\epsilon} = \lambda y . r^! \quad t^* = \lambda y . r^*$$

for some untyped terms $r^!, r^*$. From the way that type erasure works we must have $t^- = \lambda y : \sigma . r^-$ for some typed term r^- . Then $r^! = r^{-\epsilon}$ and hence the untyped computation

$$r^{-\epsilon} \triangleright r^*$$

has a typed erased subject. By the induction hypothesis there is a unique typed computation

$$r^- \triangleright r^+$$

with $r^\epsilon = r^*$, and then $r^* = r^{+\epsilon}$. Let $t^+ = \lambda y : \sigma . r^+$ to obtain the required typed computation.

The final induction step is across transitive composition. Suppose \square^* is a composition where

$$t^{-\epsilon} t^{-\epsilon} \triangleright t^! \quad t^! \triangleright t^*$$

for some intermediate untyped term $t^!$. By the induction hypothesis there is a unique typed computation

$$t^- \triangleright t^0$$

and then $t^! = t^{0\epsilon}$. But now we have

$$t^{0\epsilon} \triangleright t^*$$

so, by a second use of the induction hypothesis, there is a unique typed computation

$$t^0 \triangleright t^+$$

with $r^\epsilon = r^*$, and hence $t^* = t^{+\epsilon}$. Then $\square^\epsilon = \square^*$ and, from the shape of \square^* , this is the only possible computation. ■

6.3 Subject Reduction

We have two important facilities in a λ -calculus

the derivation system the computation mechanism

and a mediating facility, the substitution algorithm. We want to look at the main interaction between these two facilities with the goal of showing that if we are given a derivation and a reduction, more explicitly a computation, we will get a derivation for the reduced term.

We need to derive an algorithm which when given a ‘compatible’ pair

$$(\nabla) \quad \Gamma \vdash t^- : \tau \qquad (\square) \quad t^- \triangleright t^+$$

will return a derivation

$$(\nabla.\square) \quad \Gamma \vdash t^+ : \tau$$

and hence show that the object t^+ is well-formed.

Before looking at the general case it is useful to calculate the algorithm for the 1-step reduction cases.

6.5 LEMMA.

There exists an algorithm which when supplied with a derivation where the root subject is the subject of a reduction axiom will return a derivation of the corresponding object in the same context and when supplied with a derivation with a redex root subject will return a derivation with the corresponding reduct as subject in the same context.

For the proof of the above result we refer the reader to two proofs which can be combined to give the general result above. They can be found on P.81 and P.63 of [20].

Now it is possible to existence of the general algorithm.

6.6 THEOREM.

There exists an algorithm which, when supplied with

$$(\nabla) \quad \Gamma \vdash t^- : \tau \qquad (\square) \quad t^- \triangleright t^+$$

a compatible derivation and computation, will return a derivation

$$(\nabla \cdot \square) \quad \Gamma \vdash t^+ : \tau$$

the result of the action of \square on ∇ .

For our purposes the fact that such a result is known is enough. If the reader wishes to find complete details of a proof one can be found by combining the proofs found on P.63 and P.81 of [20].

As we observed in Theorem 6.4, the types in a computation have no importance. In the same way they have no role to play in a subject reduction.

Suppose we have a derivation

$$(\nabla) \quad t^{-\epsilon} : \tau$$

with a typed subject t^- . Suppose we now erase all the types of t^- to produce an untyped term $t^{-\epsilon}$. Suppose we also produce an untyped computation

$$(\square^*) \quad t^{-\epsilon} \triangleright\triangleright t^*$$

to an untyped term t^* . By Theorem 6.4 there is a typed computation

$$(\nabla) \quad t^- \triangleright\triangleright t^+$$

with $\square^\epsilon = \square^*$ and $t^{+\epsilon} = t^*$, and then the subject reduction algorithm produces

$$(\nabla \cdot \square) \quad \Gamma \vdash t^+ : \tau$$

which means we can insert types into t^* in a coherent fashion. In fact, the two computations \square and \square^ϵ have exactly the same derivation, so we may determine $\nabla \cdot \square$ directly from the given codes of ∇ and \square^* . This is summed up in the following theorem.

6.7 THEOREM.

Given a typed term $\Gamma \vdash t^- : \tau$ and a computation $(\square) \quad t^{-\epsilon} \triangleright\triangleright t^$ then there exists a unique typed term $\Gamma \vdash t^+ : \tau$ such that $t^* = t^{+\epsilon}$ and $t^- \triangleright\triangleright t^+$ by the ‘same’ reduction.*

Chapter 7

λ -Translation

In this chapter we will look at the relationship between the λ -calculus and combinator logic systems set up in the previous three chapters. Our main goal is to be able to simulate λ -abstraction in the combinator systems and simulate the combinator terms in the λ -calculus.

Simulating the combinator terms in the lambda calculus is fairly straightforward and will be covered in just one section. The rest of the chapter will be used to discuss the various algorithms which can be used to simulate λ -abstraction in terms of combinators.

The first section is short and just shows how we can simulate combinator terms in a λ -calculus. This just requires us to name the necessary λ -terms.

The majority of the work in this chapter is contained within this section. There are many ways to simulate λ -abstraction, each of which produces an algorithm. We shall analysis a variety of algorithms for their efficiency. One of those algorithms first stated by Grzegorzcyk seems to have gone unnoticed since he first published it and as we shall see, it is more efficient than some of the algorithms out there.

We conclude this chapter by looking at the relationships between the various combinator terms we have introduced as well as introducing some new ones. Essentially all combinator terms can be obtained from I, K, S and this is the main result of the section.

7.1 Simulating Combinator Terms in λ -Calculi

In this section we will be working in λ BIG. The aim is to find terms $I_\lambda, K_\lambda, S_\lambda, B_\lambda, C_\lambda$ and D_λ which have the same reduction properties as the combinators I, K, S, B, C and D .

Informally we find that the terms

$$\begin{array}{ll}
 I_\lambda = (\lambda x : \theta . x) : \iota & \iota = \theta \\
 K_\lambda = (\lambda x : \theta, y : \phi . x) : \kappa & \kappa = \theta \rightarrow \phi \rightarrow \theta \\
 S_\lambda = (\lambda x : \theta \rightarrow \phi \rightarrow \psi, y : \theta \rightarrow \phi, z : \theta . (xz)(yz)) : \sigma & \sigma = \alpha \rightarrow \epsilon \rightarrow \zeta \\
 B_\lambda = (\lambda x : \theta \rightarrow \phi, y : \psi \rightarrow \theta, z : \psi . x(yz)) : \beta & \beta = \epsilon \rightarrow \eta \\
 C_\lambda = (\lambda x : \theta \rightarrow (\phi \rightarrow \psi), y : \phi, z : \theta . xzy) : \gamma & \gamma = \theta \rightarrow \iota \rightarrow \mu \\
 D_\lambda = (\lambda x : \theta \rightarrow (\theta \rightarrow \phi), y : \theta . xyy) : \delta & \delta = \nu \rightarrow \epsilon
 \end{array}$$

where

$$\begin{array}{l}
 \alpha = \theta \rightarrow \phi \rightarrow \psi \\
 \epsilon = \theta \rightarrow \phi \\
 \zeta = \theta \rightarrow \psi \\
 \eta = (\psi \rightarrow \theta) \rightarrow (\psi \rightarrow \theta) \\
 \iota = \phi \rightarrow \psi \\
 \mu = \phi \rightarrow \zeta \\
 \nu = \theta \rightarrow \epsilon
 \end{array}$$

give the desired properties. We show this by going through each derivation in turn. This is done in the following examples.

7.1 EXAMPLES.

We find that

$$\begin{array}{ll}
 I_\lambda p & = (\lambda x . x)p \\
 & \triangleright p \\
 K_\lambda pq & = (\lambda x, y . x)pq \\
 & \triangleright (\lambda y . p)q \\
 & \triangleright p
 \end{array}$$

$$\begin{aligned}
S_{\lambda pqr} &= (\lambda x, y, z. (xz)(yz))pqr \\
&\triangleright (\lambda y, z. (pz)(yz))qr \\
&\triangleright (\lambda z. (pz)(qz))r \\
&\triangleright (pr)(qr) \\
\\
B_{\lambda pqr} &= (\lambda x, y, z. x(yz))pqr \\
&\triangleright (\lambda y, z. p(yz))qr \\
&\triangleright (\lambda z. p(qz))r \\
&\triangleright p(qr) \\
\\
C_{\lambda pqr} &= (\lambda x, y, z. xzy)pqr \\
&\triangleright (\lambda y, z. pzy)qr \\
&\triangleright (\lambda z. pz)qr \\
&\triangleright prq \\
\\
D_{\lambda pq} &= (\lambda x, y. xy)y)pq \\
&\triangleright (\lambda y. pyy)q \\
&\triangleright pqq
\end{aligned}$$

give the required reduction properties. ■

These examples were done untyped since it is neater and allows a clearer view of what is happening. No information is lost in doing this since by Theorem 6.7 we can regain the information easily.

We also note that these terms are well-formed since we showed how to derive these terms in Example 4.8.

More formally we need to define a function $(\cdot)_{\lambda}$ taking combinator terms to the corresponding λ -term, x_{λ} , which has the same reduction properties. To be even more explicit we have the following definition.

7.2 DEFINITION.

We define a function

$$(\cdot)_{\lambda} \lambda\text{BIG} \rightarrow \lambda\text{BIG}$$

in the following way. Given a combinator-term X we send it to a λ -term X_{λ} using the following rules:

- $x_\lambda = x$ if x is an identifier
- $Z_\lambda = \widehat{Z}$ if Z is a selected combinator
- $Z_\lambda = Z$ if Z is not a selected combinator
- $(XY)_\lambda = X_\lambda Y_\lambda$
- $(\lambda x : \sigma . r)_\lambda = \lambda x : \sigma . r_\lambda$

\widehat{Z} is one of the following combinators $I_\lambda, K_\lambda, S_\lambda, B_\lambda, C_\lambda$ and D_λ which are defined above and have the following properties

$$\begin{aligned} I_\lambda p &\triangleright p \\ K_\lambda pq &\triangleright p \\ S_\lambda pqr &\triangleright (pr)(qr) \\ B_\lambda pqr &\triangleright p(qr) \\ C_\lambda pqr &\triangleright prq \\ D_\lambda pq &\triangleright pqq \end{aligned}$$

as proved in Example 7.1. ■

To make this definition well-defined we need the following theorem which shows that translations under $(\cdot)_\lambda$ do not change the type of a derived term.

Consider the translator

$$r \mapsto r_\lambda$$

as given by the algorithm of Definition 7.2. This converts a term r into a term r_λ without certain combinators.

7.3 THEOREM.

There exists an algorithm which, when supplied with a derivation

$$(\nabla) \quad \Gamma \vdash r : \rho$$

will return a derivation

$$(\nabla_\lambda) \quad \Gamma \vdash r_\lambda : \rho$$

in the same context but where the root subject has been translated.

Proof.

The algorithm proceeds by recursion over the structure of ∇ .

If the supplied derivation is a leaf

$$(\nabla) \quad \Gamma \vdash Z : \rho$$

with one of the nominated combinators as the subject, then the algorithm returns the corresponding derivation

$$(\nabla) \quad \Gamma \vdash Z_\lambda : \rho$$

as given in Example 7.1.

For the other terms the algorithm returns that leaf.

At each recursion step the algorithm simply passes across that step. For example, the elimination rule.

$$\frac{\Gamma \vdash q : \pi \rightarrow \tau \quad \Gamma \vdash p : \pi}{\Gamma \vdash qp : \tau}$$

We note that the top line gets mapped to $\Gamma \vdash q_\lambda : \pi \rightarrow \tau \quad \Gamma \vdash p_\lambda : \pi$ under $(\cdot)_\lambda$. Now we apply the elimination rule to obtain $\Gamma \vdash q_\lambda p_\lambda : \tau$ which is the required derived term. ■

In short the algorithm simply locates the nominated leaves of

$$(\nabla) \quad \Gamma \vdash r : \rho$$

and grows extra branches above to convert the subject Z into a compound term.

Thus we have a well-defined way of translating combinators into λ -terms, which preserves the typing discipline. This correspondance is the only one we will look at since it is the standard one and the simplest. There is no need to consider more complicated translations since there is no substantial gain in efficiency. In later sections we will see that it is often more efficient to only translate some of the combinators rather than all of them.

7.2 λ -simulation algorithms

In Section 7.1 we showed how to translate CL-terms into terms in a λ -calculus using the function $(\cdot)_\lambda$ and now we want to find a function $(\cdot)_{\text{CL}}$ which takes terms from a λ -calculus and associates them with a term not involving any λ s. This will allow us to eliminate some or all uses of λ in a term.

Unfortunately this is not as straightforward as the other translation. The main problem as we shall see involves finding a way to simulate λ -abstraction and there are many different algorithms for doing this. We shall analyse a few of these algorithms and compare their computational merits.

We start with the following definition.

7.4 DEFINITION.

Let **CL** be a combinator system. This is a derivation system built up without the use of λ -abstraction. Also we only assert that the combinators are chosen from the list of available constants. A λ -simulator for **CL** is an algorithm which, when supplied with a **CL**-derivation

$$(\nabla) \quad \Gamma, x : \sigma \vdash t : \tau$$

will return a **CL**-derivation

$$(\tilde{\nabla}) \quad \Gamma \vdash [x : \sigma]t : \sigma \rightarrow \tau$$

where

$$([x : \sigma]t) \triangleright\triangleright t$$

holds. ■

The idea is that the constructed **CL**-term $[x : \sigma]t$ behaves like the λ -term $\lambda x : \sigma t$, and hence the corresponding applied λ -system can be ‘embedded’ in **CL**.

Of course a system **CL** need not have a translator or it may have many. The efficiency of such translators may differ quite a lot.

The following example shows a system without a translator.

7.5 EXAMPLE.

Consider the combinator system which only contains the combinators \mathbf{K} and \mathbf{l} . Let $\binom{n}{k}$ be the name for a combinator which satisfies

$$\binom{n}{k} x_1 \cdots x_n \triangleright x_k$$

for all x_1, \dots, x_n . We want to simulate this combinator in our system. Call a combinator \mathbf{A} special if it behaves like $\binom{n}{k}$ for some $1 \leq k \leq n$ where $k = n$ or $n = k + 1$. Hence \mathbf{A} selects the ultimate or penultimate component of a list of arguments (of a specified length). The pair (k, n) is the index of \mathbf{A} . We now show that $\mathbf{K}^m \mathbf{l}$ and $\mathbf{K}^m \mathbf{K}$ are special with indices $(m + 1, m + 1)$ and $(m + 1, m + 2)$ respectively. We proceed by induction over m . We need to show the following.

$$\mathbf{K}^m \mathbf{l} x_1 \cdots x_{m+1} \triangleright x_{m+1} \qquad \mathbf{K}^m \mathbf{K} x_1 \cdots x_{m+2} \triangleright x_{m+1}$$

Since

$$\mathbf{l} x_1 \triangleright x_1 \qquad \mathbf{K} x_1 x_2 \triangleright x_1$$

the base case, $m = 0$, is immediate.

For the induction step, $m \mapsto m + 1$, we have

$$\begin{aligned} \mathbf{K}^{l+1} \mathbf{l} x_1 \cdots x_{l+2} &\triangleright\triangleright\triangleright \mathbf{K} (\mathbf{K}^l \mathbf{l}) x_1 \cdots x_{l+2} \\ &\triangleright \mathbf{K}^l \mathbf{l} x_2 \cdots x_{l+2} \\ &\triangleright x_{l+2} \\ \mathbf{K}^{l+1} \mathbf{K} x_1 \cdots x_{l+3} &\triangleright\triangleright\triangleright \mathbf{K} (\mathbf{K}^l \mathbf{K}) x_1 \cdots x_{l+3} \\ &\triangleright \mathbf{K}^l \mathbf{K} x_2 \cdots x_{l+3} \\ &\triangleright x_{l+2} \end{aligned}$$

using the induction hypothesis at the second step.

Now we show that every combinator built up from \mathbf{K} and \mathbf{l} is special. Again this is done by induction. It then follows that there are combinators, for example $\binom{5}{7}$ which can not be built up from just \mathbf{K} and \mathbf{l} .

Both \mathbf{K} and \mathbf{l} are special with indexes $(1, 2)$ and $(1, 1)$ respectively. Any other combinator built up from \mathbf{K} and \mathbf{l} will have the form $\mathbf{A} = \mathbf{LR}$ where \mathbf{L} and \mathbf{R} are

special by the induction hypothesis. Note that the behaviour of L is fixed by a pair (k, n) where $1 \leq k \leq n$ with $k = n$ or $n = k + 1$. We now look at the possible values for k and n .

When $k = n = 1$ we have $A = LR \triangleright R$ so A behaves like R .

When $k = 1, n = 2$ we have $Ax = LRx \triangleright R$ so that, using the index (j, m) of R , we have that $Axx_1 \cdots x_m Rx_1 \cdots x_m \triangleright x_j$ and hence A is special with index $(j+1, m+1)$.

For the other cases we have that $2 \leq k \leq n$ so that $Ax_2 \cdots x_n = LAx_2 \cdots x_n \triangleright x_k$ and hence A is special with index $(k-1, n-1)$. ■

However for our purposes we are concerned with whether or not the combinator system with just I, K and S has a translator. The following result shows us that this is the case.

7.6 THEOREM.

Let CL be any combinator system which has combinators

$$I_\theta : \theta \rightarrow \theta$$

$$K_{\psi, \theta} : \psi \rightarrow \theta \rightarrow \psi$$

$$S_{\phi, \theta, \psi} : (\phi \rightarrow \psi \rightarrow \theta) \rightarrow (\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \theta)$$

for all types θ, ϕ, ψ . Then CL has a translator.

Proof.

Given a derivation

$$(\nabla) \quad \Gamma, x : \sigma \vdash t : \tau$$

the algorithm proceeds by recursion on the structure of ∇ . There are two cases

$$\nabla \text{ a leaf} \qquad \nabla \text{ an application}$$

where the leaf case has two subcases.

(L-x) Suppose the leaf

$$(\nabla) \quad \Gamma, x : \sigma \vdash x : \sigma$$

is the projection from x . Then

$$(\tilde{\nabla}) \quad \Gamma \vdash I_\sigma : \sigma \rightarrow \sigma$$

is the retarded derivation.

(L-non x) Suppose the leaf

$$(\nabla) \quad \Gamma, x : \sigma \vdash Z : \rho$$

is not the projection from x . Here Z can be any variable declared in Γ (so that ∇ is an axiom). In this subcase

$$(\tilde{\nabla}) \quad \Gamma \vdash \mathbf{K}_{\rho, \sigma} Z : \sigma \rightarrow \tau$$

is the returned derivation.

(Application) Suppose ∇ concludes

$$\frac{\Gamma, x : \sigma \vdash q : \pi \rightarrow \tau \quad \Gamma, x : \sigma \vdash p : \pi}{\Gamma, x : \sigma \vdash (qp) : \tau}$$

for sub-derivations Q,P. By recursion the algorithm returns derivations

$$(\tilde{Q}) \quad \Gamma \vdash \tilde{q} : \sigma \rightarrow \pi \rightarrow \tau \quad (\tilde{P}) \quad \Gamma \vdash \tilde{p} : \sigma \rightarrow \pi$$

where

$$\tilde{q} = [x : \sigma]q \quad \tilde{p} = [x : \sigma]p$$

are the root statements. Using the combinator

$$\mathbf{S}_\bullet : (\sigma \rightarrow \pi \rightarrow \tau) \rightarrow (\sigma \rightarrow \pi) \rightarrow (\sigma \rightarrow \tau)$$

the algorithm returns

$$\frac{\frac{\Gamma \vdash \mathbf{S}_\bullet : - \quad \Gamma \vdash \tilde{q} : -}{\Gamma \vdash \mathbf{S}_\bullet \tilde{q} : -} \quad \Gamma \vdash \tilde{p} : -}{\Gamma \vdash \mathbf{S}_\bullet \tilde{q} \tilde{p} : \sigma \rightarrow \tau}$$

as the translated derivation. ■

Note that in Theorem 7.5 the terms

I, K, S

need not be primitive. They can be compound terms but not containing free identifiers. They can even contain ‘hidden’ λ -constructions.

7.7 EXAMPLE.

Let CL be a combinator system which has

$$I \quad K \quad B \quad C \quad D$$

as its base terms. Then CL has a translator.

To prove this it suffices to produce a combinator S_\bullet with the appropriate reduction properties. There are two versions of this which are worth comparing. Let's look at the untyped version.

Let

$$\begin{array}{ll} S = \text{BM}(\text{BB}) & S = \text{BMC} \\ \text{where } M = \text{B}(\text{BD})\text{C} & \text{where } M = \text{B}(\text{BD})\text{B} \end{array}$$

to produce two very similar terms. The left one occurs in [2] and [9]. The right hand one seems to be new, but we will see where it comes from a little later. Both of these have the required reduction properties.

$$\begin{array}{ll} \text{B}(\text{B}(\text{BD})\text{C})(\text{BB})xyz \triangleright (\text{B}(\text{BD})\text{C})(\text{BB}x)yz & \text{B}(\text{B}(\text{BD})\text{B})\text{C}xyz \triangleright \text{B}(\text{BD})\text{B}(\text{C}x)yz \\ \triangleright (\text{B}(\text{BD})\text{C})\text{B}(xy)z & \triangleright \text{BD}(\text{B}(\text{C}x))yz \\ \triangleright \text{BD}(\text{CB})(xy)z & \triangleright \text{D}(\text{B}(\text{C}x)y)z \\ \triangleright \text{D}(\text{CB}xy)z & \triangleright \text{B}(\text{C}x)yz \\ \triangleright \text{CB}xyz & \triangleright \text{C}x(yz)z \\ \triangleright \text{C}x(yz)z & \triangleright xz(yz) \\ \triangleright xz(yz) & \end{array}$$

Thus we have a translator for this system. To type S we recall that

$$S_\bullet : (\sigma \rightarrow \pi \rightarrow \tau) \rightarrow (\sigma \rightarrow \pi) \rightarrow (\sigma \rightarrow \tau)$$

is the type of our usual S combinator and that our candidates for S will have the same type. ■

Theorem 7.5 result is essentially a typed version of the translation of untyped combinator terms into untyped λ -terms given in Lemma 2.14 in [20]. In turn this is a

version of the Curry-Feys translator which can be found in [8]. According to [3] this is essentially the Curry-Schönfinkel algorithm.

In that algorithm at an application the machine inspects whether or not x actually appears in q or p . If it does not then

$$B_{\bullet} \tilde{q}\tilde{p} \qquad C_{\bullet} \tilde{q}\tilde{p}$$

can be returned as the respective translated terms. Here B_{\bullet} and C_{\bullet} are the appropriate standard combinators. Of course if x occurs in neither q nor p then

$$K(qp)$$

can be returned as the new root.

In one sense this Curry-Feys algorithm is more efficient. The returned terms are not so big, since they do not contain unnecessary compounds.

7.8 EXAMPLE. More explicitly the Curry-Schönfinkel algorithm is the following.

$$\begin{aligned} (a) \quad & [x]x = I \\ (b) \quad & [x]Z = KZ \\ (e) \quad & [x](PQ) = S([x]P)([x]Q) \end{aligned}$$

Note that the labeling comes from the ‘fuller’ Curry-Feys algorithm.

We apply this algorithm to the λ -term for S ; $\lambda x, y, z. (xz)(yz)$. Firstly we find a term Q such that

$$Qz \triangleright (xz)(yz)$$

holds.

$$\begin{aligned} Q &= S([z](xz))([z](yz)) \\ &= S(S([z]z))(S([z]y)([z]z)) \\ &= S(S(Kx)I)(S(Ky)I) \end{aligned}$$

Now we find a term R such that

$$Ry \triangleright Q$$

holds.

$$\begin{aligned} R &= [y](S(S(Kx)I))(S(Ky)I) \\ &= S([y](S(S(Kx)I)))([y](S(Ky)I)) \\ &= S(S([y]S)([y](S(Kx)I)))(S([y](S(Ky)I)))([y]I) \end{aligned}$$

This term is clearly starting to blow up. In fact the final term contains 79 combinators. This is not a very efficient algorithm.

The Curry-Feys algorithm has the following improvements.

- (c) $x \notin FV(P), x \in FV(Q), [x](PQ) = BP([x]Q)$
- (d) $x \in FV(P), x \notin FV(Q), [x](PQ) = C([x]P)Q$

Again we find a Q such that

$$Qz \triangleright (xz)(yz)$$

holds.

$$\begin{aligned} Q &= S([z](xz))([z](yz)) \\ &= S(Bx([z]z))(By([z]z)) \\ &= S(BxI)(ByI) \end{aligned}$$

We now try to find a R such that

$$Ry \triangleright Q$$

holds.

$$\begin{aligned} R &= [y](S(BxI))(ByI) \\ &= B(S(BxI))([y](By)I) \\ &= B(S(BxI))(C([y]By)I) \\ &= B(S(BxI))(C(BBI)I) \end{aligned}$$

We now go on to find T such that

$$Tx \triangleright R$$

holds.

To do this we apply the Curry-Feys algorithm with x as the variable to the term R.

$$\begin{aligned}
T &= [x](B(S(BxI)))(C(BBI)I) \\
&= C([x]B(S(BxI)))(C(BBI)I) \\
&= C(BB([x]S(BxI)))(C(BBI)I) \\
&= C(BB(BS([x](BxI))))(C(BBI)I) \\
&= C(BB(BS(C([x]BxI))))(C(BBI)I) \\
&= C(BB(BS(C(BBI)I)))(C(BBI)I)
\end{aligned}$$

This term is still very big but it has not blown up in the way that the Curry-Schönfinkel algorithm has. Thus the Curry-Feys algorithm is more efficient. ■

In another sense the Curry-Feys algorithm is less efficient, since it is required to inspect the whole of a derivation before it passes across an application. However since both algorithms are based on recursions, they are hardly designed for efficiency in the first place.

One translator which seems to have gone unnoticed in the literature is due to Grzegorzczuk and this can be found in Theorem 1.3 of [7]. However in Grzegorzczuk's paper, his theorem is false. This is because he only builds up terms from I, B, C, D and it is not possible to create a term from these combinators which reduces in the same way as K . To show this we introduce the following terminology.

7.9 DEFINITION.

A compound combinator, A , is *daft* if whenever

$$At_m \cdots t_1 \Downarrow t$$

then

$$\partial t = \partial t_m \cup \cdots \cup \partial t_1$$

where ∂t is the set of variables of t . ■

We note that I, S, B, C, D are all daft. The following theorem brings us close to showing the required result.

7.10 THEOREM.

If A is built up from daft combinators then A is daft.

Proof.

This is done by induction. The base cases are when A is just one of the daft combinators. So A is daft in this case by definition.

Now consider when A is built up from many daft combinators, it has two components.

$$A = LR$$

We have the following reduction for A .

$$At_m \cdots t_1 = LRt_m \cdots t_1 \Downarrow t$$

By the induction hypothesis both L and R are daft. So we have that

$$\partial t = \partial R \cup \partial t_m \cup \cdots \cup \partial t_1$$

where ∂R is empty since R is daft. Hence we have that A is daft. ■

It is clear from the definition of K that it is not daft. So by Theorem 7.10 it is not possible to find a compound term A which is not daft from I, B, C, D .

However if we just add the term K to the combinator system, we find that the theorem does hold and so we get the following result.

7.11 THEOREM. (Grzegorzczuk 's Algorithm)

Let CL be any combinator system which has the combinators

$$I_\theta : \theta \rightarrow \theta$$

$$K_{\theta,\phi} : \theta \rightarrow (\phi \rightarrow \theta)$$

$$B_{\theta,\phi,\psi} : (\theta \rightarrow \phi) \rightarrow ((\psi \rightarrow \theta) \rightarrow (\psi \rightarrow \phi))$$

$$C_{\theta,\phi,\psi} : (\theta \rightarrow (\phi \rightarrow \psi)) \rightarrow (\phi \rightarrow (\theta \rightarrow \psi))$$

$$D_{\theta,\phi} : (\phi \rightarrow (\phi \rightarrow \theta)) \rightarrow (\phi \rightarrow \theta)$$

for all types θ, ϕ, ψ . Then CL has a translator.

It is worth noting that we have already proved this theorem in Example 7.7. However here we are concerned with the particular algorithm used not the existence of a translator.

Proof.

Given a derivation

$$(\nabla) \quad \Gamma, x : \sigma \vdash t : \tau$$

the algorithm proceeds by recursion on the structure of ∇ . There are two cases

$$\nabla \text{ a leaf} \qquad \nabla \text{ an application}$$

where the leaf case has two subcases.

(L-x) Suppose the leaf

$$(\nabla) \quad \Gamma, x : \sigma \vdash x : \sigma$$

is the projection from x . Then

$$(\tilde{\nabla}) \quad \Gamma \vdash \mathbf{l}_\sigma : \sigma \rightarrow \sigma$$

is the retarded derivation.

(L-non x) Suppose the leaf

$$(\nabla) \quad \Gamma, x : \sigma \vdash Z : \rho$$

is not the projection from x . Here Z can be any variable declared in Γ (so that ∇ is an axiom). In this subcase

$$(\tilde{\nabla}) \quad \Gamma \vdash \mathbf{K}_{\rho, \sigma} Z : \sigma \rightarrow \rho$$

is the returned derivation.

(Application) Suppose (∇) concludes

$$\frac{\Gamma, x : \sigma \vdash q : \pi \rightarrow \tau \quad \Gamma, x : \sigma \vdash p : \pi}{\Gamma, x : \sigma \vdash (qp) : \tau}$$

for sub-derivations Q,P. By recursion the algorithm returns derivations

$$(\tilde{Q}) \quad \Gamma \vdash \tilde{q} : \sigma \rightarrow \pi \rightarrow \tau \qquad (\tilde{P}) \quad \Gamma \vdash \tilde{p} : \sigma \rightarrow \pi$$

where

$$\tilde{q} = [x : \sigma]q \qquad \tilde{p} = [x : \sigma]p$$

are the root statements.

If q does not contain x then the algorithm uses the combinator

$$\mathbf{B}_\bullet : (\pi \rightarrow \tau) \rightarrow ((\sigma \rightarrow \pi) \rightarrow (\sigma \rightarrow \tau))$$

to return the following derivation.

$$\frac{\frac{\Gamma \vdash \mathbf{B}_\bullet : - \quad \Gamma \vdash q : -}{\Gamma \vdash \mathbf{B}_\bullet q : -} \quad \frac{Q}{\Gamma \vdash \tilde{p} : -}}{\Gamma \vdash \mathbf{B}_\bullet q \tilde{p} : \sigma \rightarrow \tau}$$

If p does not contain x then the algorithm uses the combinator

$$\mathbf{C}_\bullet : (\sigma \rightarrow (\pi \rightarrow \tau)) \rightarrow (\pi \rightarrow (\sigma \rightarrow \tau))$$

to return the following derivation.

$$\frac{\frac{\Gamma \vdash \mathbf{C}_\bullet : - \quad \Gamma \vdash \tilde{q} : -}{\Gamma \vdash \mathbf{C}_\bullet \tilde{q} : -} \quad \frac{\tilde{Q}}{\Gamma \vdash p : -}}{\Gamma \vdash \mathbf{C}_\bullet \tilde{q} p : \sigma \rightarrow \tau}$$

If p and q both contain x then the algorithm uses the combinators

$$\mathbf{B}_\bullet : (\pi \rightarrow (\sigma \rightarrow \tau)) \rightarrow ((\sigma \rightarrow \pi) \rightarrow (\sigma \rightarrow (\sigma \rightarrow \tau)))$$

$$\mathbf{C}_\bullet : (\sigma \rightarrow (\pi \rightarrow \tau)) \rightarrow (\pi \rightarrow (\sigma \rightarrow \tau))$$

$$\mathbf{D}_\bullet : (\sigma \rightarrow (\sigma \rightarrow \tau)) \rightarrow (\sigma \rightarrow \tau)$$

to return

$$\frac{\frac{\frac{\Gamma \vdash \mathbf{B}_\bullet : - \quad \frac{\frac{\Gamma \vdash \mathbf{C}_\bullet : - \quad \Gamma \vdash \tilde{q} : \sigma \rightarrow (\pi \rightarrow \tau)}{\Gamma \vdash \mathbf{C}_\bullet \tilde{q} : -}}{\Gamma \vdash \mathbf{B}_\bullet (\mathbf{C}_\bullet \tilde{q}) : -}}{\Gamma \vdash \mathbf{D}_\bullet : -} \quad \frac{\tilde{P}}{\Gamma \vdash \tilde{p} : -}}{\Gamma \vdash \mathbf{D}_\bullet ((\mathbf{B}_\bullet (\mathbf{C}_\bullet \tilde{q})) \tilde{p}) : \sigma \rightarrow \tau}}$$

as the translated derivation. ■

Both the Curry-Schönfinkel algorithm (Theorem 7.5) and the Grzegorzcyk algorithm produce long translations. These can be shortened using the Curry-Feys improvements.

(a) $\tilde{q}\tilde{p} = Bq\tilde{p}$ if x not in q

(b) $\tilde{q}\tilde{p} = C\tilde{q}\tilde{p}$ if x not in p

(c) $\tilde{q}\tilde{x} = q$ if x not in q

Note that the Grzegorzcyk algorithm only needs the third of this improvements to be added. We compare the following example with Example 7.8.

7.12 EXAMPLE.

$\lambda x, y, z.(xz)(yz)$ in the official and the quicker Grzegorzcyk algorithm.

First we need to find a z -free term Q st

$$Qz \triangleright (xz)(yz)$$

holds. Officially this is

$$D(B(CX)Y)$$

where

$$Xz \triangleright xz \quad Yz \triangleright yz$$

and Y, X are obtained be the algorithm.

Officially

$$X = D(B(CU)V)$$

where

$$Uz \triangleright x \quad Vz \triangleright z$$

and U, V are atomic cases. Thus

$$U = Kx \quad V = I$$

are the required terms. Note that we can take

$$X = x \quad Y = y$$

and so

$$Q = D(B(Cx)y)$$

will do.

We now find a y -free term R with the property.

$$Ry \triangleright Q$$

Officially

$$R = D(B(Cd)e)$$

where

$$dy \triangleright D \quad ey \triangleright y$$

holds. This starts to blow up.

We observe that

$$BD(B(Cx))y \triangleright Q$$

so we can take

$$R = BD(B(Cx))$$

to avoid the blow up.

Finally we require S where

$$Sx \triangleright R$$

holds. The official algorithm blows up but we observe

$$S = BMC \quad \text{where } M = B(BD)B$$

gives

$$\begin{aligned} S &\triangleright B(BD)B(Cx) \\ &\triangleright BD(B(Cx)) = R \end{aligned}$$

as required. ■

Grzegorzczuk's algorithm is very similar to the one used by Hindley in [9]. Here Hindley works in a combinator system using B, C, K and D as the combinators.

So he has to find a term which will reduce in the same way as S . Here is the necessary result, Theorem 9F3 Combinatory Completeness Theorem, taken from [9].

7.13 THEOREM.

Let CL be any combinator system which has the combinators

$$\mathbf{K}_{\theta,\phi} : \theta \rightarrow (\phi \rightarrow \theta)$$

$$\mathbf{B}_{\theta,\phi,\psi} : (\theta \rightarrow \phi) \rightarrow ((\phi \rightarrow \theta) \rightarrow (\psi \rightarrow \phi))$$

$$\mathbf{C}_{\theta,\phi,\psi} : (\theta \rightarrow (\phi \rightarrow \psi)) \rightarrow (\phi \rightarrow (\theta \rightarrow \psi))$$

$$\mathbf{D}_{\theta,\phi} : (\phi \rightarrow (\phi \rightarrow \theta)) \rightarrow (\phi \rightarrow \theta)$$

for all types θ, ϕ, ψ . Then CL has a translator.

Proof.

The algorithm is the following.

- (a) $x \notin FV(N)$ $[x]N = \mathbf{K}N$
- (b) $[x]x = \mathbf{C}\mathbf{K}\mathbf{K}$
- (c) $x \notin FV(P), x \in FV(Q)$ $[x]PQ = \mathbf{B}P([x]Q)$
- (d) $x \in FV(P), x \notin FV(Q)$ $[x]PQ = \mathbf{C}([x]P)Q$
- (e) $x \in FV(P), x \in FV(Q)$ $[x]PQ = \mathbf{D}((\mathbf{B}(\mathbf{C}([x]Q)))([x]P))$

This algorithm works since it is essentially the Curry-Feys algorithm with new terms for \mathbf{I} and \mathbf{S} . ■

We now have the following corollaries.

7.14 COROLLARY.

The combinator \mathbf{I} has the same reduction properties as the term $\mathbf{C}\mathbf{K}\mathbf{K}$.

Proof.

By definition we have the following reduction for \mathbf{I} .

$$\mathbf{I}x \triangleright x$$

Now we need to look at the reduction properties of $\mathbf{C}\mathbf{K}\mathbf{K}$.

$$\mathbf{C}\mathbf{K}\mathbf{K}x \triangleright \mathbf{K}x\mathbf{K}$$

$$\triangleright x$$

Thus \mathbf{I} and $\mathbf{C}\mathbf{K}\mathbf{K}$ have the same reduction properties. ■

7.15 COROLLARY.

The combinator S has the same reduction properties as the term $B(B(BD)C)(BB)$.

Proof.

By definition we have the following reduction for S .

$$Sxyz \triangleright (xz)(yz)$$

Now we need to look at the reduction properties of $B(B(BD)C)(BB)$.

$$\begin{aligned} B(B(BD)C)(BB)xyz &\triangleright (B(BD)C)(BBx)yz \\ &\triangleright (B(BD)C)B(xy)z \\ &\triangleright BD(CB)(xy)z \\ &\triangleright D(CBxy)z \\ &\triangleright CBxyz \\ &\triangleright Cx(yz)z \\ &\triangleright xz(yz) \end{aligned}$$

Thus S and $B(B(BD)C)(BB)$ have the same reduction properties. ■

The first thing to note is the style in which the theorem has been laid out. It has not been done in the derivation style we have been using up to now. It is done in the usual style for λ -simulation algorithms. As we look at more λ -simulation algorithms and start to compare them we will use this style. This is simply because as the algorithms become more complicated they become messy to write out in the derivation style as can be seen in the proof of Theorem 7.11.

In the proof of Theorem 7.13 we defined S . This is different to the way that it was defined in Example 7.12. In fact we have two different expressions for it used in two different algorithms. It is now worthwhile comparing the two expressions to see which is the most efficient algorithm. We have the following result.

7.16 THEOREM.

Grzegorzczuk's expression for S reduces quicker than Hindley's result for S .

Proof.

If we look at Example 7.7 we see that Hindley's expression for S has taken 7 reductions while Grzegorzcyk's expression has only taken 6. Thus Grzegorzcyk's expression is slightly faster at reduction. ■

As we have seen Grzegorzcyk's algorithm is slightly faster than Hindley's. There are ways to improve the algorithm even more and we will now go on to discuss some of the other algorithms available as well as making some improvements to Grzegorzcyk's algorithm in light of these new algorithms.

We find another algorithm in [23], which attempts to avoid the blow up present in the Curry algorithms.

Turner's trick is to modify the S combinator so that it becomes

$$S'kxyz = k(xz)(yz)$$

which replaces S in the algorithm.

The algorithm can be modified further by replacing B and C by B' and C' respectively. Where

$$B'kxyz = kx(yz) \quad C'kxyz = k(xz)y$$

are the new terms.

We find this improves the Curry algorithms quite significantly.

More work has been done in improving the Curry algorithms yet further. A good summary of this work can be found in [3].

Here Bunder looks at improvements to the Turner algorithm which he concludes is the best of the currently available algorithms (in 1990). He also concludes that there are always further optimisations which can be applied to improve the algorithm however these often also complicate it. The most significant result of this paper is the fact that there will always be a blow up in certain cases.

For an alternative approach to this algorithm problem see [12] and [11]. Here Jones attempts to tackle this problem from a programming perspective. The uniqueness of Jones' account is that he does not convert the derivation tree into a natural number.

He attempts to deal with the tree as an ‘object’, this allows him to analyse the complexity of the algorithm. The complexity is usually lost by encoding the derivation tree. It is very hard to deal with derivation trees in this manner and we see that Jones runs into problems very quickly.

7.3 Getting New (and Old) Combinators from Old

We have set up our system λ BIG with six combinators however this is in fact a bit excessive. Historically people put in as many combinators as they could make up. As time went on people began to realise that it was possible to use terms built up from existing combinators instead of some of the ones already in use. The standard nowadays is just to use the combinators

$$S \quad K$$

as the primitive ones and any others you want can be built up from these. However we will define the following combinators as primitive.

7.17 DEFINITION.

The following combinators are primitive in the sense that any other combinators can be built up from them.

$$S \quad K \quad I$$

Thus we can think of λ BIG as only having these combinators as raw combinators. ■

We have called I a primitive combinator since it is easier to include it than exclude it. However it is possible to find a term made up from S and K which will reduce in the same way as I as the following proposition shows.

7.18 PROPOSITION.

The term

$$SKK$$

has the same reduction properties as

$$I$$

in the sense that

$$\text{SKK}x \triangleright x$$

holds.

Proof.

We have the following reduction.

$$\begin{aligned} \text{SKK}x &\triangleright (\text{K}x)(\text{K}x) \\ &\triangleright x \end{aligned}$$

Thus $I = \text{SKK}$. ■

We now need to show that we can get the remaining combinators from just our primitive ones. The following theorem shows how to achieve this.

7.19 THEOREM.

The terms

$$\text{SS(KI)} \quad \text{S(KS)K} \quad \text{S(BBBS)(KK)}$$

have the same reduction properties as

$$\text{D} \quad \text{B} \quad \text{C}$$

respectively.

Proof.

We have the following reductions.

$$\begin{aligned} \text{SS(KI)}xy &\triangleright (\text{S}x)(\text{K}I)x)y \\ &\triangleright (\text{S}x)Iy \\ &\triangleright (xy)(Iy) \\ &\triangleright xyy \\ \text{S(KS)K}xyz &\triangleright (\text{K}Sx)(\text{K}x)yz \\ &\triangleright \text{S(K}x)yz \\ &\triangleright (\text{K}xz)(yz) \\ &\triangleright x(yz) \end{aligned}$$

$$\begin{aligned}
S(BBS)(KK) &\triangleright (BBSx)(KKx)yz \\
&\triangleright B(Sx)(KKx)yz \\
&\triangleright Sx(KKxy)z \\
&\triangleright xz(KKxyz) \\
&\triangleright xz(Kyz) \\
&\triangleright xzy
\end{aligned}$$

Thus we have that

$$D = SS(KI) \quad B = S(KS)K \quad C = S(BBS)(KK)$$

giving us the required raw combinators built up from our primitive combinators. ■

We have now shown that it is possible to obtain all our raw combinators from just the primitive ones. Often when proving results about combinators in our system λ BIG we will just prove the result for the primitive combinators. From now on this will be done without reference to the fact that we can build up the remaining combinators from just the primitives and so do not need to prove the result for all the combinators.

We have seen how it is possible to build up some of our existing combinators from primitive combinators. It is now useful to provide examples of new combinators which can be built up from our raw combinators. These new combinators will be required later on and will be treated as though they are raw after this point.

7.20 EXAMPLE.

The Turner combinator, T

We have a combinator T which reduces in the following way

$$Txyz \triangleright w(xz)(yz)$$

and is built up from B, S as follows.

$$T = B(BSB)$$

It's reduction path is given by.

$$\begin{aligned} Txyz &= B(BSB)xyz \triangleright BSB(wx)yz \\ &\triangleright S(Bwx)yz \\ &\triangleright (Bwxz)(yz) \\ &\triangleright w(xz)(yz) \end{aligned}$$

Also note that

$$T = (S(KS)K)((S(KS)K)S(S(KS)K))$$

if we just express it in terms of primitive combinators. ■

Chapter 8

Formal Recursion

Here we formalise the informal handling of recursion in Chapter 1. We will not gain much understanding from the formalisation but it will at least assure the reader that we have not been fudging the mathematics by doing it informally.

The first section will formalise both recursion and iteration which we looked at in Section 1.1. They will be formalised by being put into our λ -calculus which we introduced in Chapters 4 and 5.

In Section 8.2 we will formalise Kleene's trick which we introduced in Section 1.2. The formalised version of Kleene's trick will be called Bernays' trick to allow us to distinguish when we are discussing it formally and when informally.

8.1 Formal Recursion

When we defined λ BIG, we introduced

$$\text{Rec}_\sigma : \sigma \rightarrow (\sigma \rightarrow \mathcal{N} \rightarrow \sigma) \rightarrow \mathcal{N} \rightarrow \sigma$$

and

$$\text{It}_\sigma : \mathcal{N} \rightarrow \sigma''$$

as recursion gadgets. The first of these will allow us to formalise recursion. The second allows us to formalise iteration.

As we mentioned at the time, it is possible to change the order of the types in the Rec_σ operator. This means we can change the order in which it receives functions and parameters. However we are going to stick with the same order which we introduced in the informal section.

As our first example we will look at head recursion and see how it can be expressed using Rec_σ .

8.1 EXAMPLE.

We are given functions

$$f : \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{T}$$

$$g : \mathbb{P} \rightarrow \mathbb{T}$$

$$h : \mathbb{P} \rightarrow \mathbb{N} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$$

with head recursion defined by the following equations.

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

We want to introduce Rec_σ so that it's reduces to gp when we evaluate it at 0 and h when we evaluate it at x' .

Firstly we 'hide' the parameters and so work with the parameter free versions of the above functions. In Example 1.8, we showed how to do this. We also change the order of the inputs of h . Again this does not cause any problems since we are working with a curried form of h .

We work with the following functions.

$$\tilde{f} : \mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T})$$

$$\tilde{g} : \mathbb{P} \rightarrow \mathbb{T}$$

$$\tilde{h} : (\mathbb{P} \rightarrow \mathbb{T}) \rightarrow \mathbb{N} \rightarrow (\mathbb{P} \rightarrow \mathbb{T})$$

Here we have the modified base and recursion steps

$$\tilde{f}0 = \tilde{g} \quad \tilde{f}x' = \tilde{h}(\tilde{f}x)x$$

We have to choose σ so that we have the following reduction properties for Rec_σ .

$$\text{Rec}_\sigma \ulcorner \tilde{g} \urcorner \ulcorner \tilde{h} \urcorner 0 \triangleright \ulcorner \tilde{g} \urcorner \quad \text{Rec}_\sigma \ulcorner \tilde{h} \urcorner \ulcorner \tilde{g} \urcorner x' \triangleright \ulcorner \tilde{h} \urcorner (\text{Rec}_\sigma x)x$$

where $\ulcorner \tilde{g} \urcorner, \ulcorner \tilde{h} \urcorner$ represent \tilde{g} and \tilde{h} in λBIG . Here we are assuming that \tilde{g} and \tilde{h} can be represented in λBIG otherwise we would not be able to formalise the recursion. This is a reasonable assumption, since for our purposes any function we want to use can be named in λBIG .

We let $\sigma = \rho \rightarrow \tau$ where ρ and τ correspond to the types for \mathbb{P} and \mathbb{T} in λBIG . Thus we have

$$\text{Rec}_{\rho \rightarrow \tau} : (\rho \rightarrow \tau) \rightarrow ((\rho \rightarrow \tau) \rightarrow \mathcal{N} \rightarrow (\rho \rightarrow \tau)) \rightarrow \mathcal{N} \rightarrow (\rho \rightarrow \tau)$$

as the required Rec_σ . ■

From this example we learn that with our version of Rec_σ , functions must be parameter free. However all the examples we looked at can be converted to parameter free recursion by ‘hiding’ their parameters. When this is not possible, the use of a recursion gadget which accepts parameters has to be used. A suitable one can be found in [17] on P.27 although the main recursion step is incorrect and should reduce to $V(\widehat{\text{rec}}_\sigma UV \mathbf{W}n)n$.

However so far we have only shown how to capture the recursion step within the calculus to completely formalise the recursion we need to derive the recursion within the calculus. We need to have two separate derivation trees, one for the base step and one for the recursion step. However these are straightforward enough since we have chosen our types correctly. Thus we have the following derivation tree, where $\Gamma = \text{Rec}_{\rho \rightarrow \tau} : -, \ulcorner \tilde{g} \urcorner : -, \ulcorner \tilde{h} \urcorner : -, 0 : -$.

$$\frac{\frac{\Gamma \vdash \text{Rec}_{\rho \rightarrow \tau} : - \quad \Gamma \vdash \ulcorner \tilde{g} \urcorner : -}{\Gamma \vdash \text{Rec}_{\rho \rightarrow \tau} \ulcorner \tilde{g} \urcorner : -} \quad \Gamma \vdash \ulcorner \tilde{h} \urcorner : -}{\Gamma \vdash \text{Rec}_{\rho \rightarrow \tau} \ulcorner \tilde{g} \urcorner \ulcorner \tilde{h} \urcorner : -} \quad \Gamma \vdash 0 : -}{\Gamma \vdash \text{Rec}_{\rho \rightarrow \tau} \ulcorner \tilde{g} \urcorner \ulcorner \tilde{h} \urcorner 0 : -}$$

For the main recursion step, the tree is the same except that we have x' instead of 0 at the final elimination.

This formalisation can be repeated for any of the other recursions we have done informally.

The iterator is used to formalise iterations which we have not looked at in the same way as we have recursion. In fact we have shown how to get from recursion to iteration using Kleene's trick, which we will formalise later.

An iterator attempts to capture the idea of applying a function many times to itself. We thus get at higher order functions by repeated application of the same lower order function to itself. This is very similar to the way that recursion operators work and hints at why it is possible to translate between the two of them.

More specifically we are given a function $f : \mathbb{S} \rightarrow \mathbb{S}$ which can be applied to itself many times.

$$f^0 = id_{\mathbb{S}} \quad f^1 = f \quad f^2 = f \circ f \quad f^3 = f \circ f^2 \dots f^{r+1} = f \circ f^r$$

The iterator captures this process by firstly obtaining the base case by the following reduction.

$$\text{It}_\sigma 0 f s \triangleright s$$

It also gives the main iteration step via the following reduction.

$$\text{It}_\sigma x' f s \triangleright f(\text{It}_\sigma x f s)$$

Again we can easily build up derivation trees for the iterators as long as we make sure the types are correctly chosen.

8.2 Bernays' Trick

In this section we wish to look at a way of transferring from a recursion to an iteration within the λ -calculus. This is done by means of an algorithm referred to as Bernays' trick, see [10]. It is a formalisation of Kleene's trick that we looked at earlier. In the literature there is not a clear distinction between Kleene's trick and Bernays' trick as I have done here. In fact different sources refer to the formalised version as both Kleene's trick and Bernays' trick.

We find that throughout this section we will need to refer to the reduction properties of the recursion operator, Rec_σ . For convenience and ease of reference they have

$$\text{Rec}_\sigma st0 \triangleright s \quad \text{Rec}_\sigma stn' \triangleright t(\text{Rec}_\sigma stn)n$$

are the reduction steps of

$$\text{Rec}_\sigma : \sigma \rightarrow (\sigma \rightarrow \mathcal{N} \rightarrow \sigma) \rightarrow \mathcal{N} \rightarrow \sigma$$

Table 8.1: Reduction Properties of Rec_σ

been listed in a table. From now on we will refer to the reduction properties of the recursion operator, Rec_σ without making reference to Table 8.1.

Our main result is finding a function which has the same reduction properties as Rec_σ but involves an iterator. We proceed by a series of lemmas.

8.2 LEMMA.

Given a primitive recursion operator, Rec_σ , which is defined in Table 8.1. We can find a function

$$F : \mathcal{N} \times \sigma \rightarrow \mathcal{N} \times \sigma = (\mathcal{N} \times \sigma)'$$

with

$$F^x(0, s) = (x', \text{Rec}_\sigma stx)$$

for all $x \in \mathbb{N}$.

Proof.

Firstly we define

$$F : (\mathcal{N} \times \sigma)'$$

by

$$F(x, u) = (x, tux)$$

and we now substitute $\text{Rec}_\sigma stx$ for u to get this equality.

$$F(x, \text{Rec}_\sigma stx) = (x', t(\text{Rec}_\sigma stx)x) = (x', \text{Rec}_\sigma stx')$$

To show that $F^x(0, s) = (x, \text{Rec}_\sigma stx)$ we proceed by induction over x .

The base case comes from the following equality.

$$F^0(0, s) = (0, \text{Rec}_\sigma st0)$$

We now show the induction step, $x \mapsto x'$, holds.

$$F^{x'}(0, s) = F(F^x(0, s)) = F(x, \text{Rec}_\sigma stx) = (x', \text{Rec}_\sigma stx')$$

So F is the required function. ■

We now show that this function can be simulated in λBIG . This means it will be possible to turn the above argument from one in terms of functions to one about elements of λBIG .

8.3 LEMMA.

The function F given in Lemma 8.2 can be simulated in λBIG by

$$\ulcorner F^\urcorner = \lambda t : \sigma \rightarrow \mathcal{N} \rightarrow \sigma, w : \mathcal{N} \times \sigma. \text{Pair}((\text{Left}w)')(t(\text{Right}w)(\text{Left}w))$$

and has the following judgment.

$$\vdash \ulcorner F^\urcorner : (\sigma \rightarrow \mathcal{N} \rightarrow \sigma) \rightarrow (\mathcal{N} \times \sigma)'$$

Proof.

We apply $\ulcorner F^\urcorner$ to the pair

$$(x', \text{Rec}_\sigma stx)$$

and

$$t : \sigma \rightarrow \mathcal{N} \rightarrow \sigma$$

which are of the required type and so we have.

$$\ulcorner F^\urcorner t(x, \text{Rec}_\sigma stx) = (x', t(\text{Rec}_\sigma stx)x)$$

Thus we have simulated F in λBIG . To show that $\ulcorner F^\urcorner$ has the required type consider the following derivation tree, where $\Gamma = t : \sigma \rightarrow \mathcal{N} \rightarrow \sigma, w : \mathcal{N} \times \sigma$.

$$\frac{\frac{\frac{\Gamma \vdash \text{Left} : - \quad \Gamma \vdash w : -}{\Gamma \vdash \text{Left}w : -}}{\Gamma \vdash \text{Suc} : - \quad \Gamma \vdash \text{Left}w : -}}{\Gamma \vdash \text{Pair} : - \quad \Gamma \vdash (\text{Left}w)' : -}}{\Gamma \vdash \text{Pair}((\text{Left}w)') : -}$$

We now need to show that

$$\ulcorner H \urcorner = \lambda s : \sigma, t : \sigma \rightarrow \mathcal{N} \rightarrow \sigma, x : \mathcal{N} \times \sigma. (\text{It}_{\mathcal{N} \times \sigma} \text{Left} x) \ulcorner F \urcorner (\text{Pair} 0 \text{Right} x)$$

simulates H in λBIG .

We apply $st(x, s)$ to $\ulcorner H \urcorner$ to get

$$\ulcorner H \urcorner st(x, s) = \text{Right} (\text{It}_{\mathcal{N} \times \sigma} x) \ulcorner F \urcorner (\text{Pair} 0 s)$$

which reduces to

$$\text{Right} \ulcorner F \urcorner^x (\text{Pair} 0 s)$$

as required.

We need just need to show that $\ulcorner H \urcorner$ has a derivation. To see this look at the following tree with $\Gamma = s : \sigma, t : \sigma \rightarrow \mathcal{N} \rightarrow \sigma, x : \mathcal{N} \times \sigma$.

$$\frac{\frac{\frac{\Gamma \vdash \text{Right} : - \quad \Gamma \vdash x : -}{\Gamma \vdash \text{Right} x : -}}{\Gamma \vdash \text{It}_{\mathcal{N} \times \sigma} : -} \quad \Gamma \vdash \ulcorner F \urcorner : -}{\Gamma \vdash \text{It}_{\mathcal{N} \times \sigma} (\text{Right} x) : -} \quad \frac{\Gamma \vdash \text{Pair} : - \quad \Gamma \vdash 0 : -}{\Gamma \vdash \text{Pair} 0 : -} \quad \frac{\Gamma \vdash \text{Right} : - \quad \Gamma \vdash x : -}{\Gamma \vdash \text{Right} x : -}}{\Gamma \vdash \text{It}_{\mathcal{N} \times \sigma} (\text{Right} x) \ulcorner F \urcorner \text{Pair} 0 (\text{Right} x) : -} \quad \frac{\Gamma \vdash \text{It}_{\mathcal{N} \times \sigma} (\text{Right} x) \ulcorner F \urcorner \text{Pair} 0 (\text{Right} x) : -}{s : \sigma, t : \sigma \rightarrow \mathcal{N} \rightarrow \sigma \vdash \lambda x : \mathcal{N} \times \sigma. \text{It}_{\mathcal{N} \times \sigma} (\text{Right} x) \ulcorner F \urcorner \text{Pair} 0 (\text{Right} x) : (\mathcal{N} \times \sigma)'} \quad \frac{s : \sigma \vdash \lambda t : \sigma \rightarrow \mathcal{N} \rightarrow \sigma, x : \mathcal{N} \times \sigma. \text{It}_{\mathcal{N} \times \sigma} (\text{Right} x) \ulcorner F \urcorner \text{Pair} 0 (\text{Right} x) : -}{\vdash \ulcorner H \urcorner : \sigma \rightarrow (\sigma \rightarrow \mathcal{N} \rightarrow \sigma) \rightarrow (\mathcal{N} \times \sigma)'}$$

Thus we find that $\ulcorner H \urcorner$ has a derivation in λBIG . ■

Chapter 9

The Calculus of Primitive Recursion

Here we set up a λ -calculus whose functions are precisely the primitive recursive functions. We will need this calculus later on when we show that Kleene's S1-S8 generate the primitive recursive functions.

In the first section we set up the calculus and show what it looks like at the lower level.

In the next section we use the exhaustive search method to show that if we only allow functions with first order types as our inputs we just get out the primitive recursive functions.

9.1 The System λPR_0

Firstly we will set up the λ -calculus of first order primitive recursive functions, λPR_0 . Most of the work was done when we set up λBIG since a lot of similar functions and rules are contained within λPR_0 .

We find that λPR_0 contains the following base functions.

$$\text{Suc} : \mathbb{N} \rightarrow \mathbb{N}$$

$$0 : \mathbb{N} \rightarrow \mathbb{N}$$

$$(\cdot) : \mathbb{N}^l \rightarrow \mathbb{N}$$

We note that (\cdot) is the projection function. It can project from any coordinate.

We have the following rules which allow us to build up functions.

$$\begin{array}{l} \text{(Recursion)} \quad \frac{g : \mathbb{N} \rightarrow \mathbb{N} \quad h : \mathbb{N} \rightarrow \mathbb{N}}{\text{Rec.} \bullet gh : \mathbb{N} \rightarrow \mathbb{N}} \\ \text{(Application)} \quad \frac{h : \mathbb{N}^l \rightarrow \mathbb{N} \quad g_1 : \mathbb{N}^l \rightarrow \mathbb{N}, \dots, g_k : \mathbb{N}^l \rightarrow \mathbb{N}}{h \circ (g_1, \dots, g_k)} \end{array}$$

The recursion rule is what gives this system its power and allows us to simulate the first order primitive recursive functions. Notice that this rule does not specify what the recursion operator actually does, it just allows us to build up terms which will allow the operator to act. The recursion is also parameter free.

The application rule is similar to the one in λBIG except that it takes multiple inputs. We also name the term it produces

$$f \underline{x} = h(g_1 \underline{x}, \dots, g_k \underline{x})$$

so that the application rule is really just simulating composition.

We do not need to add any weakening rule to this system since we can achieve the same effect by just using the above rules.

Any function we grow from these rules will be primitive recursive and so our λ -calculus will capture all first order primitive recursive functions.

9.2 Setting up λPR

We now wish to set up the much bigger λ -calculus λPR , of which λPR_0 is a subsystem. This bigger λ -calculus will capture all higher order primitive recursive functions and so will be the underlying λ -calculus of the system set up by Kleene.

In this system we will only use \rightarrow types, mainly because it will make the main result easier to prove. Note that we are not losing anything by just having \rightarrow types since by we can recover \times types from just \rightarrow types as we showed earlier.

We first need to look at the curried version of first order types which are given along with the types used by Kleene.

9.1 DEFINITION.

The arrow types of level one or curried first order types

$$(\mathcal{N}(l) \mid l < \omega)$$

are generated by

$$\mathcal{N}(0) = \mathcal{N} \quad \mathcal{N}(1+l) = \mathcal{N} \rightarrow \mathcal{N}(l)$$

for $l < \omega$.

\mathcal{N} is a Kleene-pure (or clean) type with code

$$\#\mathcal{N} = 0$$

If π is a pure type with code $\#\pi = p$ then

$$\pi \rightarrow \mathcal{N}$$

is a pure type with

$$\#(\pi \rightarrow \mathcal{N}) = p + 1$$

giving us it's code. ■

We need to distinguish between clean types and curried first order types. They both attempt to get at the same idea, that of capturing numeric gadgets. However clean types have a coding built into them, this makes them a bit messier and thus harder to use. We will in general use the curried first order types since they are neater and allow us to see what is actually happening.

Both definitions are recursive and this allows lots of types to be obtained from just our original atomic type \mathcal{N} .

9.2 EXAMPLE.

Here we see how curried first order types are built up.

$$\begin{aligned} \mathcal{N}(0) &= \mathcal{N} \\ \mathcal{N}(1) &= \mathcal{N} \rightarrow \mathcal{N} = \mathcal{N}' \\ \mathcal{N}(2) &= \mathcal{N} \rightarrow \mathcal{N}' \\ \mathcal{N}(3) &= \mathcal{N} \rightarrow (\mathcal{N} \rightarrow \mathcal{N}') \end{aligned}$$

The following are all clean types together with their code number just built up from \mathcal{N} .

$$\begin{array}{ll} \mathcal{N} & \#\mathcal{N} = 0 \\ \mathcal{N} \rightarrow \mathcal{N} & \#\mathcal{N} \rightarrow \mathcal{N} = 1 \\ (\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N} & \#(\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N} = 2 \\ ((\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N}) \rightarrow \mathcal{N} & \#((\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N}) \rightarrow \mathcal{N} = 3 \end{array}$$

We see that the clean types are a lot messier than the curried first order types. ■

We use Convention 4.5 for the brackets contained in the clean and curried first order types.

The system λPR is built up from the usual constants

$$0 : \mathcal{N} \quad \text{Suc} : \mathcal{N}'$$

and a family of recursors.

9.3 DEFINITION.

For each $l < \omega$ the constant

$$\text{Prec}_l : \mathcal{N}(l) \rightarrow \mathcal{N}(l+2) \rightarrow \mathcal{N}(l+1)$$

has 1-step reductions

$$\text{Prec}_l \text{gh} \underline{p} 0 \triangleright \underline{gp} \quad \text{Prec}_l \text{gh} \underline{p} r' \triangleright \underline{hpr}(\text{Prec}_l \text{gh} \underline{p} r)$$

where

$$\underline{p} = p_l \cdots p_1$$

is the list of parameters. ■

Collapsing the parameters $p_l \cdots p_1$ is a very useful trick since it allows us to treat them as one parameter but we have to be very careful with it since we may need to change the order of the parameters, for example, which could easily be missed if we are not paying attention.

Notice that Prec_0 does not consume any parameters.

$$\text{Prec}_0gh0 \triangleright g \qquad \text{Prec}_0ghr' \triangleright hr(\text{Prec}_0ghr)$$

This is a Gödel recursor. As we noted earlier this is also an example of a Kleene recursor.

Now we can finally define λPR .

9.4 DEFINITION.

Let λPR be the applied λ -calculus with only arrow types and constants

$$0 : \mathcal{N} \qquad \text{Suc} : \mathcal{N}' \qquad \text{Prec}_l : \mathcal{N}(l)$$

for $l < \omega$. ■

9.3 The strength of λPR

We need to make clear the distinction between the construction of a primitive recursive function and its name. It is possible for the same function to have two different constructions, as the following example illustrates.

9.5 EXAMPLE.

Our aim is to obtain two constructions for multiplication. We begin by constructing addition. This is done by primitive recursion. Our base step is

$$A0y = y$$

while the recursion step is one of the following.

$$Ax'y = \text{Suc}(Axy) \qquad Ax(\text{Suc}y)$$

The first of these is an example of head recursion while the latter is tail recursion.

We now define multiplication by the following primitive recursion.

$$M0y = 0$$

$$Mx'y = A(Mxy)y \quad Ay(Mxy)$$

We find that both of these constructions give a function which simulates the role of multiplication in the calculus. However as we have just shown they have two very different constructions.

When we constructed a function which simulates multiplication we also used two different constructions. We are not using addition as our main example since the difference between the two constructions was that one was a head recursion and one was tail. We have not shown but it is not too difficult to do so that we can translate between a head and a tale recursion very easily. This means they are essentially the same construction and we would not get a lot out of analysing the differences between the two constructions.

We wish to analyse the differences between the two constructions. We find that one is more efficient than the other We illustrate this difference in efficiency by doing a simple calculation.

Firstly we look at the effect of the addition algorithm on the size of the algorithm. We show by induction that every time the addition algorithm is used to add x and y it takes $x + 1$ steps.

The base case is immediate. The induction step follows from

$$Ax^{n+1}y = \text{Suc}(Axy)$$

and the induction argument gives us that the last part takes $x+1$ steps to get the result.

We can already see where the problem is going to arise with reagrds the efficiency of the algorithms. The first algorithm requires the addition algorithm to be called within the recursion step rather than outside it. This means we are going to be doing addition for large x and so get a very inefficient algorithm. The second algorithm is more efficient since it does not call the addition algorithm within the recursion step and so each time addition is called it only adds 6 steps to the algorithm. Thus it should turn out to be more efficient than the first. We show this by carrying out a multiplication and counting the steps.

We show that 7 multiplied by 5 is 35 using both algorithms.

$$\begin{aligned}
M75 &= A(M(6)5)5 \\
&= A(A(M(5)5)5)5 \\
&= A(A(A(M(4)5)5)5)5 \\
&= A(A(A(A(M(3)5)5)5)5)5 \\
&= A(A(A(A(A(M(2)5)5)5)5)5)5 \\
&= A(A(A(A(A(A(M(1)5)5)5)5)5)5)5 \\
&= A(A(A(A(A(A(A(M(0)5)5)5)5)5)5)5)5 \\
&= A(A(A(A(A(A(A(0)5)5)5)5)5)5)5 \\
&\vdots \text{ 1 step} \\
&= A(A(A(A(A(A(5)5)5)5)5)5)5 \\
&\vdots \text{ 6 steps} \\
&= A(A(A(A(A(10)5)5)5)5)5 \\
&\vdots \text{ 11 steps} \\
&= A(A(A(A(15)5)5)5)5 \\
&\vdots \text{ 16 steps} \\
&= A(A(A(20)5)5)5 \\
&\vdots \text{ 21 steps} \\
&= A(A(25)5)5 \\
&\vdots \text{ 26 steps} \\
&= A(30)5 \\
&\vdots \text{ 31 steps} \\
&= 35
\end{aligned}$$

We have just shown the above algorithm to be very inefficient. It has taken the algorithm 111 steps to calculate 7 times 5. This is a very large number of steps for such a simple calculation. We know the other algorithm will be more efficient but we ask ourselves how much better is it? In theory it should be a lot more efficient since we will now longer be calling the addition algorithm within the recursion step which is a very stupid thing to do anyway.

Using the other construction we obtain.

$$\begin{aligned}
M75 &= A_5(M_6(5)) \\
&= A_5(A_5(M_5(5))) \\
&= A_5(A_5(A_5(M_4(5)))) \\
&= A_5(A_5(A_5(A_5(M_3(5))))) \\
&= A_5(A_5(A_5(A_5(A_5(M_2(5)))))) \\
&= A_5(A_5(A_5(A_5(A_5(A_5(M_1(5))))))) \\
&= A_5(A_5(A_5(A_5(A_5(A_5(A_5(M_0(5)))))))) \\
&= A_5(A_5(A_5(A_5(A_5(A_5(0))))) \\
&\vdots \text{ 6 steps} \\
&= A_5(A_5(A_5(A_5(A_5(5)))) \\
&\vdots \text{ 6 steps} \\
&= A_5(A_5(A_5(A_5(10)))) \\
&\vdots \text{ 6 steps} \\
&= A_5(A_5(A_5(15))) \\
&\vdots \text{ 6 steps} \\
&= A_5(A_5(20)) \\
&\vdots \text{ 6 steps} \\
&= A_5(A_5(25)) \\
&\vdots \text{ 6 steps} \\
&= A_5(30) \\
&\vdots \text{ 6 steps} \\
&= 35
\end{aligned}$$

So the second construction is faster. ■

We are more concerned with which functions can be constructed using the rules of λ PR rather than which construction is used.

Each primitive recursive function can be named in λ PR. To name such a function we take a construction of f as a primitive recursive function. We track through this construction to produce terms in λ PR which name the component functions.

We are now concerned with how ‘big’ λ PR is. The obvious question is what other first order functions can be named in λ PR? To put this more formally we ask, given a derivation

$$\vdash f : \mathcal{N}(l)$$

in the empty context, what kind of function is named by f .

It seems ‘obvious’ that f should be primitive recursive since λ PR only has primitive recursion available to use. However, λ PR can use higher order facilities, and perhaps these have a hidden effect.

The proof is not entirely straight forward and we use the exhaustive search method to prove the required result. To prove the result we want we consider derivations

$$(F) \quad \Gamma \vdash f : \mathcal{N}(l)$$

where

$$\Gamma = x_m : \mathcal{N}, \dots, x_1 : \mathcal{N}$$

for some l, m . Such a derivation will give us

$$\vdash \lambda x_m : \mathcal{N}, \dots, x_1 : \mathcal{N}. f : \mathcal{N}(m + l)$$

and this term names a first order function of $m + l$ arguments.

Our goal is to prove the following theorem.

9.6 THEOREM.

For each derivation (F) in λ PR, where the term f is normal, the function named is primitive recursive.

We will show this in the next section using the exhaustive search method.

9.4 Translation in λ PR

We wish to find an upper bound on the functions generated by λ PR. We are only interested in the first order case and this is shown by the restriction placed on terms in the theorems.

We use the exhaustive search method to show that in λPR we only get the primitive recursive functions. The exhaustive search method goes through all the possible cases and shows which ones are actually possible. We find that any functions which are not primitive recursive cannot satisfy the restriction placed on the theorems.

The exhaustive search method is a rather tedious style of proof but when done correctly it is very thorough and shows clearly what is going on.

Note that we are only allowing first order functions as inputs. This means that even though higher order functionals might appear within the proof, eventually we will just end up with first order functions. This is by no means an obvious or straightforward result to prove.

9.7 THEOREM.

For each derivation

$$(\nabla) \quad \Gamma \vdash t : \tau$$

which satisfies the following global conditions:

- *The context Γ is a list of declarations $x : \mathcal{N}$*
- *The subject term t is normal*
- *The predicate type τ is a first order type in $\mathcal{N}(l)$*

∇ *provides a primitive recursive function or in the extreme case (where Γ is empty and $\tau = \mathcal{N}$) t is a numeral.*

Proof.

We proceed by induction over the height of ∇ . To do this we look at all the possible ways that ∇ can arise.

Firstly, ∇ may be a leaf.

(∇ axiom) Since we are working in λPR there are three possible shapes for $t : \tau$.

(i) $0 : \mathcal{N}$

(ii) $\text{Suc} : \mathcal{N}'$

(iii) $\text{Prec}_l : \mathcal{N}(l) \rightarrow \mathcal{N}(l+2) \rightarrow \mathcal{N}(l+1)$

(i) and (ii) meet the global conditions but (iii) does not since $\mathcal{N}(l) \rightarrow \mathcal{N}(l+2) \rightarrow \mathcal{N}(l+1)$ has the wrong shape to be a predicate type for the global conditions.

(i) This provides a primitive recursive function which is identically 0.

(ii) This provides a primitive recursive function f where

$$f(\mathbf{x}, y) = \text{Suc}y = y + 1$$

for the appropriate \mathbf{x} and y .

In the extreme case, we find that $\mathbf{0}$ is the only possible option and so we have a numeral.

(∇ projection) This provides a primitive recursive function f given by

$$f(\mathbf{x}) = x_j$$

where j indicates the position of the projected declaration.

In the extreme case, we find projection impossible since Γ is empty and so we have nothing to project from.

Secondly, ∇ may arise by a use of one of the construction rules. These will involve a shorter derivation to which, if the global conditions apply, we may use the induction hypothesis.

(∇ weakening) Using the induction hypothesis we see that the numerator of this particular use provides some translation of g , and then the denominator provides a primitive recursive function f obtained from g by inserting a dummy argument.

This case cannot arise in the extreme case since Γ is empty.

(∇ introduction) As explained above, the numerator and denominator of such a use provide the same function.

Again this case is not possible in the extreme case since $\tau = \mathcal{N}$ and so if this was the result of an introduction, the previous term would not have a type.

(∇ elimination) Here we have two shorter derivations

$$\Gamma \vdash s : \rho \rightarrow \tau \quad \Gamma \vdash r : \rho$$

with $t = sr$. We look at the way the left hand one can arise. We track through its construction to unravel as many uses of elimination and weakening as possible. Thus we obtain a family of shorter derivations

$$(Q) \quad \Xi \vdash q : \pi_k \rightarrow \dots \rightarrow \pi_1 \rightarrow \tau \quad (P_i) \quad \Pi_i \vdash p_i : \pi_i \quad 1 \leq i \leq k$$

where

$$t = qp_k \dots p_1$$

with q and each p_i normal, and where each context Ξ and Π_i is a left hand part of Γ . Note that $k \geq 1$.

In this process we may have removed some uses of weakening. These can be reinstated to obtain derivations

$$(Q^+) \quad \Gamma \vdash q : \pi_k \rightarrow \dots \rightarrow \pi_1 \rightarrow \tau \quad (P_i^+) \quad \Gamma \vdash p_i : \pi_i \quad 1 \leq i \leq k$$

all in the original context. Since $k \geq 1$, each of these is strictly shorter than ∇ . We do not need to reinstate the missing parts of Γ , however it makes the proof easier to follow if we do.

We consider how Q can arise. Let

$$\chi = \pi_k \rightarrow \dots \rightarrow \pi_1 \rightarrow \tau$$

be Q 's predicate type.

When we analyse the extreme case we find that each p_i has type \mathcal{N} and that q has type $\mathcal{N} \rightarrow \dots \rightarrow \mathcal{N} \rightarrow \mathcal{N}$.

(Q axiom) There are three possible choices here as listed above.

(i) This can not arise since χ must be compound.

(ii) Here we must have

$$(Q^+) \quad \Gamma \vdash \text{Suc} : \mathcal{N} \rightarrow \mathcal{N} \quad (P^+) \quad \Gamma \vdash p : \mathcal{N}$$

with $k = 1, \pi_1 = \mathcal{N}, \tau = \mathcal{N}$ and $t = \text{Suc}p$ for some p . The shorter derivation P^+ meets the global conditions and hence, by the induction hypothesis, this

derivation provides a primitive recursive function. The function provided by ∇ is the composite of this function followed by the successor function, and so is primitive recursive.

In the extreme case, we find that this case can arise. We note that by the induction hypothesis, p must be a numeral and so the result is the numeral $p + 1$.

(iii) We have

$$(Q^+) \quad \Gamma \vdash \text{Prec}_l : \mathcal{N}(l) \rightarrow \mathcal{N}(l+2) \rightarrow \mathcal{N}(l+1)$$

and several

$$(P_i^+) \quad \Gamma \vdash p_i : \pi_i$$

which gives rise to

$$\Gamma \vdash \text{Prec}_l p_\bullet \dots p_\bullet : \mathcal{N}(m)$$

which satisfies the global conditions.

There are at least two p_i since if we just have one, which we call

$$(G) \quad \Gamma \vdash g : \mathcal{N}(l)$$

we obtain

$$\Gamma \vdash \text{Prec}_l g : \mathcal{N}(l+2) \rightarrow \mathcal{N}(l+1)$$

which does not satisfy the global conditions. We now add another p_i which we call

$$(H) \quad \Gamma \vdash h : \mathcal{N}(l+2)$$

to obtain

$$\Gamma \vdash \text{Prec}_l gh : \mathcal{N}(l+1)$$

which does satisfy the global conditions. If we wish we can add more p_i 's to further unravel $\mathcal{N}(l+1)$ up to a maximum of l lots. These will still produce a first order type and so satisfy the global conditions.

We now can apply the induction hypothesis. We apply it to (G) to obtain a primitive recursive function and similarly to (H) since they are both derivations which are shorter than the original one we are looking at. From these two primitive recursive functions we can use the application rule to obtain gh which is also primitive recursive. We then apply the application rule again with Prec_l which we have shown to be primitive recursive to obtain $\text{Prec}_l gh$ which is primitive recursive.

In the extreme case, we find that we cannot have derivations which will produce functions of higher type which we can use as input functions into the recursor. Thus this case cannot arise.

This completes the case where Q is an axiom.

(Q projection) This case can not occur since the only possible projected statement from the context Γ has form $x : \mathcal{N}$, and this type does not match χ .

(Q weakening) This case is built into the unraveling ∇ .

(Q introduction) In this case we have $q = (\lambda x : \pi . r)$ for some term r , where $\pi = \pi_k$. But then with $p = p_k$ we see that the redex $(\lambda x : \pi . r)p$ is an initial part of t . Since t is normal, this case cannot arise.

(Q elimination) This case is built into the unraveling ∇ .

This concludes the proof. ■

This theorem can be extended to a system which allows product types \times and iterators. The global conditions change slightly since our predicate type τ can now be a product type $\sigma \times \rho$ for some σ, ρ . We also find that our conclusion also has the added case that t can be $\text{Pair}lr$ for some terms l, r if τ is a product type. This is laid out in the following theorem.

9.8 THEOREM.

For each derivation

$$(\nabla) \quad \Gamma \vdash t : \tau$$

which satisfies the following global conditions:

- The context Γ is a list of declarations $x : \mathcal{N}$
- The subject term t is normal
- The predicate type τ is either a first order type $\mathcal{N}(l)$ or a product type $\sigma \times \rho$ for some $\sigma \times \rho$

then if τ is first order ∇ provides a primitive recursive function. otherwise τ is a product type and t is $\text{Pair}lr$ for some terms l, r . In the extreme case (where Γ is empty and $\tau = \mathcal{N}$) we find that we have a numeral.

Proof.

The proof is almost identical to the proof of Theorem 9.7 except that we have to add extra steps at the (∇ axiom) and (Q axiom) stages.

(∇ axiom)' There are seven possible shapes for $t : \tau$.

- | | |
|--|--|
| (i) $0 : \mathcal{N}$ | (v) $\text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho$ |
| (ii) $\text{Suc} : \mathcal{N}'$ | (vi) $\text{Left} : \sigma \times \rho \rightarrow \sigma$ |
| (iii) $\text{Prec}_l : \mathcal{N}(l) \rightarrow \mathcal{N}(l+2) \rightarrow \mathcal{N}(l+1)$ | (vii) $\text{Right} : \sigma \times \rho \rightarrow \rho$ |
| (iv) $\text{It} : \mathcal{N} \rightarrow \mathcal{N}''$ | |

Here σ, ρ are arbitrary. The new cases are the iterator and the pairing gadgets. We find that these do not arise since each of

$$\mathcal{N} \rightarrow \mathcal{N}'' \quad \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad \sigma \times \rho \rightarrow \sigma \quad \sigma \times \rho \rightarrow \rho$$

has the wrong shape to be a predicate type for the global conditions.

(Q axiom)' Here there are seven cases and we will only concern ourselves with the four new cases.

(iv) We have

$$(Q^+) \quad \Gamma \vdash \text{It} : \mathcal{N} \rightarrow \mathcal{N}' \rightarrow \mathcal{N} \rightarrow \mathcal{N}$$

and at least two and perhaps all three of

$$(E^+) \quad \Gamma \vdash e : \mathcal{N} \quad (H^+) \quad \Gamma \vdash h : \mathcal{N}' \quad (G^+) \quad \Gamma \vdash g : \mathcal{N}$$

where e, h, g are terms. We have either of

$$t = \text{lte}hg \quad t = \text{lte}h$$

depending on whether or not G^+ occurs. However, each of E^+, H^+, G^+ is a shorter derivation that meets the global conditions so, by the induction hypothesis, they provide functions

$$e(\mathbf{x}) \quad h(\mathbf{x}, \cdot) \quad g(\mathbf{x})$$

which are primitive recursive. We then find that the function produced is

$$f(\mathbf{x}, y, z) = h(\mathbf{x}, \cdot)^z y$$

with $e(\mathbf{x})$ being substituted for z and, when required $g(\mathbf{x})$ being substituted for y . Since

$$f(\mathbf{x}, y, 0) = y \quad f(\mathbf{x}, y, z') = h(\mathbf{x}, f(\mathbf{x}, y, z))$$

we see that f is primitive recursive in h .

(v) Here we have at least

$$(Q^+) \quad \Gamma \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad (L^+) \quad \Gamma \vdash l : \sigma$$

and possibly

$$(R^+) \Gamma \vdash r : \sigma$$

as well. In other words one of

$$q = \text{Pair}l \text{ with } \tau = \rho \rightarrow \sigma \times \rho \quad q = \text{Pair}lr \text{ with } \tau = \sigma \times \rho$$

holds. The first of these is discounted by the global conditions on τ , and the second meets the required condition so we can apply the induction hypothesis and hence get a primitive recursive function.

(vi) Here we have at least

$$(Q^+) \quad \Gamma \vdash \text{Left} : \sigma \times \rho \quad (P^+) \quad \Gamma \vdash p : \sigma \times \rho$$

where there may be other auxiliary derivations P_2^+, \dots, P_k^+ . The shorter derivation P^+ meets the global conditions and hence the induction hypothesis gives $p = \text{Pair}lr$ for some terms l, r . We now have

$$t = \text{Left}(\text{Pair}lr) \triangleright l$$

which contradicts the normality of t so this case can not arise.

(vii) As with (vi) this case can not arise.

This completes the (Q axiom) step.

For the extreme case here, we note that nothing is added which will occur when we look at the extreme. Thus the result holds. ■

Chapter 10

Kleene Computable Functions

We return to Kleene's S1-S8 which we briefly discussed in Chapter 2. We are now going to translate them into λ PR which we have just set up. Our main result will be that we only get the primitive recursive functions if we restrict ourselves to first order functions.

In the first section we show how to translate S1-S8 into λ PR. We also analyse some of the peculiarities of some of the rules.

We conclude by analysing what functions we get from S1-S8. Together with the main result of Section 9.4 we prove that we only get primitive recursive functions if we restrict our input to first order functions.

10.1 S1-S8 in λ PR

We convert each clean construction into a derivation in λ PR. This calculus is described in Chapter 9.

The syntactic clean types are generated in the obvious way. Thus each source

$$\begin{array}{c} \mathbb{S}_m \times \cdots \times \mathbb{S}_1 \\ x_m, \cdots, x_1 \end{array}$$

give a clean context

$$\Gamma = x_m : \sigma_m, \dots, x_1 : \sigma_1$$

in the obvious way.

We describe an algorithm which converts each clean S1-S7 construction

$$\{label\} \quad (\underline{x} : \underline{\sigma}) = f$$

into a derivation

$$(Label) \quad \Gamma \vdash f : \mathcal{N}$$

which names the same function.

The algorithm proceeds by recursion over the input construction.

The translation of the first three rules is immediate.

$$(S1) \quad \Gamma \vdash \text{Suc}x_1 : \mathcal{N} \text{ the successor function}$$

$$(S2) \quad \Gamma \vdash \text{Suc}^m \ulcorner 0 \urcorner : \mathcal{N} \text{ the constant function}$$

$$(S3) \quad \Gamma \vdash \quad x_1 : \mathcal{N} \text{ the projection function}$$

Notice that in S1 and S3 it is the gate declaration that is used. This is why we indexed from right to left otherwise we would have to keep track of which is our highest number and this can change whereas one will not.

For some reason Kleene has decided to introduce every constant rather than just introduce the zero constant. This amounts to the same thing since every numerical constant can be obtained from zero and the successor function. Only introducing the zero function would have kept the system simpler and neater but Kleene does not seem to like doing things this way.

(S6) This is essentially a book keeping rule which allows us to project from any position. It brings the k^{th} input to the 1^{st} position by a permutation. There is no reason why we can not just project from the k^{th} position and apply the successor function to any input. This means that (S1) and (S3) become

$$(S1)' \quad \Gamma \vdash \text{Suc}x_k : \mathcal{N}$$

$$(S3)' \quad \Gamma \vdash \quad x_k : \mathcal{N}$$

thus removing the need for (S6).

However we must be careful in just dropping (S6) and using (S1)' and (S3)' instead since both (S4) and (S5) assume that (S6) is included. Again it does not require much to alter them so that they are more general however this will be left to the reader.

It is also up to the reader whether they want to use the more generalised rules or the original rules used by Kleene and I guess that will depend on whether or not the reader is of minimalist persuasion. The inclusive of (S6) in the first place shows that Kleene was not thinking of keeping things simple. We will include (S6) for historical reasons only.

(S7) This is essentially the rule

$$\frac{\Gamma \vdash q : \pi \rightarrow \mathcal{N} \quad \Gamma \vdash p : \pi}{\Gamma \vdash qp : \mathcal{N}}$$

where the two numerators are projections.

(S4) Informally this rule starts from two functions

$$g : \mathbb{N} \times \mathbb{S} \rightarrow \mathbb{N} \quad h : \mathbb{S} \rightarrow \mathbb{N}$$

and produce

$$f : \mathbb{S} \rightarrow \mathbb{N}$$

where

$$f\underline{x} = g(h\underline{x}, \underline{x})$$

(for $\underline{x} \in \mathbb{S}$).

It is more convenient to swap the inputs of g . We use

$$g : \mathbb{S} \times \mathbb{N} \rightarrow \mathbb{N}$$

with h as above and produce f by

$$f\underline{x} = g\underline{x}(h\underline{x})$$

which is just a use of the \mathbf{S} -combinator.

Within λPR we convert

$$\Gamma, y : \mathcal{N} \vdash g : \mathcal{N} \quad \Gamma \vdash h : \mathcal{N}$$

into

$$\Gamma \vdash (\lambda y : \mathcal{N} . g)h : \mathcal{N}$$

by the use of an introduction followed by an elimination.

(S5) As in (S4) it is more convenient if we re-order the inputs. Informally the rule converts

$$g : \mathbb{S} \rightarrow \mathbb{N} \quad h : \mathbb{S} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

into

$$f : \mathbb{S} \times \mathbb{N} \rightarrow \mathbb{N}$$

by

$$f(\underline{x}, 0) = g\underline{x} \quad f(\underline{x}, r') = h(\underline{x}, r, (f\underline{x}, r))$$

for $\underline{x} \in \mathbb{S}$ and $r \in \mathbb{N}$. Within λPR this is a use of the basic recursor

$$\text{Rec}_{\mathcal{N}} : \mathcal{N} \rightarrow (\mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N} \rightarrow \mathcal{N}$$

with no parameter inputs. The rule converts

$$\Gamma \vdash g : \mathcal{N} \quad \Gamma, y : \mathcal{N}, z : \mathcal{N} \vdash h : \mathcal{N}$$

into

$$\Gamma \vdash \text{Rec}_{\mathcal{N}}g(\lambda y, z : \mathcal{N}. h) : \mathcal{N}'$$

by a couple of introductions and eliminations.

(S8) is called an application but is a rather strange use of the term as we shall see.

Informally, let Π be a clean type and set

$$\Pi^+ = \Pi \rightarrow \mathbb{N} \quad \Pi^{++} = \Pi^+ \rightarrow \mathbb{N}$$

to produce the next two clean types.

After re-ordering the inputs the rule produces a function

$$f(\underline{x} : \mathbb{S}, y : \Pi^{++}) = y(\lambda z : \Pi. h(\underline{x}, y, z))$$

where h has been produced already.

Thus the rule uses the following derivation.

$$\frac{\Gamma, y : \tau^{++}, z : \tau \vdash h : \mathcal{N}}{\Gamma, y : \tau^{++} \vdash y : \tau^{++} \quad \Gamma, y : \tau^{++} \vdash (\lambda z : \tau. h) : \tau^+} \frac{}{\Gamma, y : \tau^{++} \vdash y(\lambda z : \tau. h) : \mathcal{N}}$$

We note that it uses a projection, introduction and application within the derivation.

The version of S8 mentioned above is not the general version that Kleene himself used. Since we are only concerned with total functions, the infinitary side condition has not been mentioned. This covers the case where the input for S8 is not a first order function. If the function is not defined for y then we cannot use S8.

10.2 Naming the Kleene Computable Functionals?

Using S1-S8, we can generate a class of functionals. This we will call the class of clean functionals. We want to know what this class contains.

We shall only worry about what happens when we allow our input to be first order functions, thus we actually generate a class of clean functions. Hence forth we will refer to the class of clean functions to refer to those functions generated by S1-S8 when we only allow first order input.

In Section 9.4 we used the exhaustive search method to show that every first order function in λPR is primitive recursive. We also showed that λPR_0 is a subsystem of λPR .

We showed in Section 10.1 that every function generated by S1-S8 can be simulated in λPR_0 . We actually just showed that each rule S1-S8 can be translated into a λ -calculus. However it is clear from the formulation of S1-S8 in the λ -calculus and our definition of λPR_0 that every clean function generated by S1-S8 can be formed in λPR_0 .

We have nearly shown that the class of functions generated by S1-S8 is just the class of primitive recursive functions. We have shown that the upper bound on the class of clean functions is the class of primitive recursive functions and have gone most of the way through the proof of showing that the class of primitive recursive functions is contained within the class of clean functions.

To complete the proof that the class of clean functions is the class of primitive

recursive functions we just need to show that the primitive recursive functions can be generated by S1-S8.

We do this by showing that each of the rules for generating primitive recursive functions can be obtained by S1-S8. The primitive recursive functions are generated by the following rules.

- **Suc**, **0**, *Proj* are all primitive recursive functions.
- If g, h are primitive recursive then **Rec**. gh is also primitive recursive.
- If h, g_1, \dots, g_k are all primitive recursive functions then $h \circ (g_1, \dots, g_k)$ is primitive recursive.

We need to show that each of these rules are contained within S1-S8.

The successor function is obtained from S1. The zero function comes from S2 with $m = 0$. The projection function comes from S3 and S6, since the projection function needs to be able to project from any coordinate.

S5 allows us to capture the recursive operator. We note that the recursive operator used in the definition of the class of primitive recursive functions will have the same base and recursive step as the ones explicitly mentioned in S5.

The final rule is substitution, since we are taking a function h and changing each of its coordinates x_i to $g(x_i)$. We firstly need to work out what this substitution looks like in the judgement style. We find that given a derivation of h we need to change each coordinate to something of the form $\lambda u : \mathcal{N}. g_i(u, \dots)$. More formally, we have the following lemma.

10.1 LEMMA.

Given a derivation of h , $\Gamma \vdash h : \mathcal{N}$ we can substitute for the first coordinate x_i the following term

$$\lambda u : \mathcal{N}. g_1(u, \dots)$$

and end up with a derivation

$$\Gamma \vdash h(\lambda u : \mathcal{N}. g_1(u, \dots), \dots)$$

in the λ -calculus.

Proof.

We note that when we talked about S4 in Section 10.1, we changed the order of the inputs and said that S4 derived the term

$$(\lambda y : \mathcal{N} . g)h : \mathcal{N}$$

which is very similar to the term we want. If we had not changed the inputs, we would have derived the term

$$h(\lambda y : \mathcal{N} . g) : \mathcal{N}$$

which is exactly what we require here. ■

We have not worried about the other coordinates nor shown the more general version of substitution which primitive recursive requires since these follow as a natural corollary to the above lemma. We just need to apply S6 and S4 repeatedly until we get the required result.

We now have the final result which shows us exactly which functions are contained within the class of clean functionals.

10.2 THEOREM.

The class of functions generated by S1-S8, when we allow only first order input is precisely the class of primitive recursive functions.

Proof.

We have shown the following

$$\text{PrimRec} \subseteq \text{Clean} \subseteq \lambda\text{PR}_0 \subseteq \lambda\text{PR} \subseteq \text{PrimRec}$$

which is sufficient to prove the result. ■

Note that if we had allowed input of higher order functions the above proof would fail since the inclusions

$$\text{Clean} \subseteq \lambda\text{PR}_0$$

and

$$\lambda\text{PR} \subseteq \text{PrimRec}$$

would not hold while the other inclusions would.

This result is quite surprising since S8 allows us to use higher order functionals. Infinitively adding higher order functionals to our system should increase the size of the class of functions generated. However the proof of Theorem 9.1 shows the cases where we should be able to use higher order functionals cannot be obtained. Thus we have gained nothing at the first order level by the inclusion of S8 in our rules.

At higher order levels however, S8 does make a difference to the size of the class of clean functionals. We also find that S9, an additional rule of Kleene's which we are not going to examine here, also plays a vital role in determining what functionals belong to the class of clean functionals.

Here, the moral is that at the first order level, there is no need to have such complex (and often confusing) rules such as S1-S8 since we can generate exactly the same functions using a suitable λ -calculus with a careful choice of constants.

Note that Kleene has no explicit substitution or composition rules. These are both basic properties you want within a system. It makes no sense whatsoever to leave these out or have them as derived rules. One has to wonder why Kleene set up his system in the way he has.

Chapter 11

Platek's Functions

Finally we return to Platek's P1-P14 in the hope of making sense of them. However time restrictions and the depth of Platek's work means not as much analysis could be done on these as originally hoped.

Here we manage to translate P1-P14 into a λ -calculus but we do not explicitly state what it is. It cannot be λ PR since we show that the jump functions can be simulated in P1-P14 although it is not clear whether or not we have broken the rules of the system by allowing them. We also speculate on how big the class of Platek functions is but we do not manage to show what it is.

11.1 P1-P14 in a λ -calculus

We convert P1-P14 into a judgement style in the same way that we converted Kleene's rules.

The translation of the following rules are immediate.

$P2 \quad \Gamma \vdash \quad x_1 : \mathcal{N} \quad \text{the projection function}$

$P4 \quad \Gamma \vdash \quad 0 : \mathcal{N} \quad \text{the constant zero}$

$P5 \quad \Gamma \vdash \text{Suc}x_1 : \mathcal{N} \quad \text{the successor function}$

P6-P8 require their derivations to be translated into the appropriate λ -calculus. This is straightforward and we will illustrate how this is achieved by translating P6.

We need a derivation which ends with

$$\Gamma, x : \sigma, y : \rho \vdash \text{Pair}(x, y) : \sigma \times \rho$$

as the final step. The tree looks like this.

$$\frac{\frac{\Gamma \vdash \text{Pair} : \sigma \rightarrow \rho \rightarrow \sigma \times \rho \quad \Gamma \vdash x : \sigma}{\Gamma \vdash \text{Pair}x : \rho \rightarrow \sigma \times \rho} \quad \Gamma \vdash y : \rho}{\Gamma \vdash \text{Pair}xy : \sigma \times \rho}$$

Note that we have not restricted the type of x and y . It would be possible to restrict their types to \mathcal{N} . Unfortunately there is not time to analyse what effect this will have on the system but one would assume that it weakens it.

It would have been possible to specify a pairing gadget in the setup. The two most common are

$$\text{Pair}(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x$$

which is a recursive bijection and

$$\text{Pair}(x, y) = 2^x \cdot 3^y$$

which relies on prime factorisation. Each specific pairing gadget has its pros and cons and so there is no overall 'best' choice. This is why it is sensible to allow the person using the system to choose their own pairing gadget.

P3 requires the use of a conditional. We have the choice of either introducing the conditional as a constant into our system or creating one within it. Platek has chosen to introduce a conditional and we shall assume there is a sensible reason why he has done this.

Here is how we would go about creating a conditional in a primitive recursive system. Firstly we define the almost minus function. We define $x \dot{-} y$ by induction on x . The base case is

$$0 \dot{-} y = 0$$

and the induction step, $x' \mapsto x$, is the following.

$$x' \dot{-} y = \begin{cases} (x \dot{-} y) + 1 & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$$

Thus we have a function

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

which can be converted into the necessary conditional by the following trick. Consider the function

$$x \# y = (x \dot{-} y) + (y \dot{-} x) = \begin{cases} > 0 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

which together with the sign function

$$sg(z) = \begin{cases} 1 & \text{if } z \neq 0 \\ 0 & \text{if } z = 0 \end{cases}$$

gives us the required conditional.

$$C(x, y) = sg(x \# y) = \begin{cases} 1 & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

Since Platek introduced rather than created a conditional, we must assume that something goes wrong with the above construction. On examination of Platek's rules we see that there does not appear to be a composition rule although P9 gives us something close. This lack of a composition rule is why the above construction fails although it may be possible to get composition from his rules but again this is far from clear.

We note that P9 is identical to Kleene's S4 and so given two functions

$$g : \overbrace{\mathbb{S}_l, \dots, \mathbb{S}_1}^x, \mathbb{N} \rightarrow \mathbb{N}$$

$$h : \mathbb{S}_l, \dots, \mathbb{S}_1 \rightarrow \mathbb{N}$$

we combine them by an introduction and elimination to obtain

$$\Gamma \vdash (\lambda y : \mathcal{N}. g)h : \mathcal{N}$$

as our new function.

P10 is just primitive recursion and so can be obtained in our λ -calculus by simply using the recursion operator. Thus our rule is the following.

$$\Gamma \vdash \text{Rec}_{\mathcal{N}}g(\lambda y, z : \mathcal{N}. h) : \mathcal{N}'$$

P11 is just a book keeping rule and is identical to S6. So again if we wish we can eliminate it and simplify our rules. In fact there is no reason why it needs to be used at all.

P13 is nothing more than a removal of extra terms on the left hand side of the gate and applying a function to the remaining terms. This rule is given by the following.

$$x_l \dots, x_{k+1}, x_k, \dots, x_1 \vdash \phi_i(x_k, \dots, x_1)$$

The final rule we need to worry about is P14 which is the only 'new' rule we have encountered in Platek's system. Essentially this rule attempts to capture a form of substitution. We are given a term

$$f : \mathcal{N}$$

in a context, $\Gamma, u : \mathcal{N}$. We then carry out an introduction and then an application with another function we are also given to come up with the new term. This means we are substituting a new term in each of the coordinates in a given function. The simplest case looks like this. We are given

$$f : \mathcal{N} \quad \Phi : \mathcal{N}' \rightarrow \mathcal{N}$$

and we apply an introduction to f to obtain

$$\Gamma \vdash (\lambda u : \mathcal{N}. f) : \mathcal{N}'$$

which can be applied to Φ to get the new function.

$$\Gamma \vdash \Phi(\lambda u : \mathcal{N}. f) : \mathcal{N}$$

We now want ϕ to name a jump operator. We let Γ give the inputs \underline{z} .

We produce the terms

$$\Gamma, x : \mathcal{N} \vdash \ulcorner \psi \urcorner : \mathcal{N}$$

$$\Gamma, x : \mathcal{N} \vdash \ulcorner \theta \urcorner : \mathcal{N}$$

which name ϕ, θ .

We now want a term ϕ which will name the jump operator. We already have $\ulcorner \Phi \urcorner$ in our system.

We apply an introduction to $\ulcorner\theta\urcorner$ and $\ulcorner\phi\urcorner$ to obtain

$$\Gamma \vdash (\lambda x : \mathcal{N} . \ulcorner\psi\urcorner(x, z)) : \mathcal{N}' \quad \Gamma \vdash (\lambda x : \mathcal{N} . \ulcorner\theta\urcorner(x, z)) : \mathcal{N}'$$

which can be applied to $\ulcorner\Phi\urcorner$ to get

$$\Gamma \vdash \ulcorner\Phi\urcorner(\lambda x : \mathcal{N} . \ulcorner\psi\urcorner(x, z))(\lambda x : \mathcal{N} . \ulcorner\theta\urcorner(x, z)) : \mathcal{N}$$

which gives a typical value of ϕ .

Since we are aiming to get at a jump operator, we let

$$\Phi fg = f^{g(0)}2$$

which is named by

$$\ulcorner\Phi\urcorner = \lambda f, g : \mathcal{N}' . \text{lt}(g0)f\ulcorner 2\urcorner$$

in our calculus.

Then

$$\ulcorner\Phi\urcorner(\lambda x : \mathcal{N} . \ulcorner\psi\urcorner(x, z))(\lambda x : \mathcal{N} . \ulcorner\theta\urcorner(x, z)) : \mathcal{N}$$

reduces to

$$\text{lt}\ulcorner\theta\urcorner[x = 0](\lambda x . \ulcorner\psi\urcorner)\ulcorner 2\urcorner$$

which reduces yet further to give

$$(\lambda x . \ulcorner\psi\urcorner)\ulcorner\theta\urcorner[x=0]\ulcorner 2\urcorner$$

which is a jump operator.

This construction can be modified to show that any jump operator can be constructed in Platek's system.

One of the strange things about Platek's system is that he does not have a standard composition rule. Composition is a well understood mathematical idea, so unless I am missing something deep, it would be sensible to include it. There is a similar argument for the inclusion of a 'proper' substitution rule.

Conclusion

Unfortunately time has had a bigger effect on the material covered in this dissertation than the author would have liked. There is a lot of material which given more time (and more money) would have added to a much greater understanding of the material. Very little analysis of higher order functionals was actually carried out and this is where the real work needs to be done. The Chapters on both Platek and Kleene would have benefited with more time given to their work and a much fuller analysis would have been beneficial. However this is only the result of approximately three months work so time was always a big factor.

Rather than further lamenting on what could have been done we shall now look at what was actually achieved. We were lucky to discover a result first proved by Grzegorzcyk which seems to have gone unnoticed in the literature. Hopefully we have presented it in such a way as to allow people access to an important result.

Study of both Kleene and Platek's systems have shown many problems that arise from their approach. One of the major failings of both of them is their failure to include a substitution and composition rule. These are well understood ideas in mathematics and integral to any useful system. We find that a lot of work has to been done to get anywhere near a substitution rule and it is not clear whether or not composition can be achieved.

Another major failing of both Kleene and Platek is their approach to the development of their systems. Both seem more concerned with indexing rules rather than setting up an appropriate λ -calculus. Admittedly towards the end of his thesis, Platek manages to set up an appropriate λ -calculus but he fails to take advantage of it. From our analysis it is possible to see that using a suitable λ -calculus, it is

possible to make a very clear distinction between the construction of a function and the function itself. This is something which Kleene and Platek fail to do and so we find that it is very easy for them to get confused between the two.

One of the important lessons learnt in this dissertation is how best to approach the systems currently available to study functionals. The first thing that needs to be done is set up a suitable λ -calculus since this will make all the later analysis much clearer. Then translate the system into this λ -calculus which will then allow a clear distinction to be made between the construction and the function itself. It is possible to analyse both using this λ -calculus and this analysis is clear. This is the method currently being used to approach these systems and it is the opinion of the author that this is the best way available at the moment.

A decision was made early on to look at all the original papers with hindsight this was not the best way to approach this dissertation.

Bibliography

- [1] Jeremy Avigad and Solomon Feferman *The Handbook of Proof Theory: Chapter V Gödel's Functional ("Dialectica") Interpretation* Elsevier, (1988)
- [2] H. P. Barendregt: *The lambda calculus its syntax and semantics*, North Holland Publishing Company, 1981
- [3] M. W. Bunder: *Some improvements to Turner's algorithm for bracket abstraction*, The Journal of Symbolic Logic, Vol 55, Number 2 (June 1990) 656-669
- [4] Pierre-Louis Curien and Thérèse Hardin: *A simple proof of non confluence of lamda-calculus with surjective pairing*, Available at <http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00014.html>
- [5] Jean-Yves Girard: *Proofs and Types*, Cambridge University Press, 1989
- [6] Kurt Gödel: *Über eine bisher noch nicht benützte Erweiterung des finiten Standpunkten*, Dialectica 47-48, vol. 12 (1958) 280-287
- [7] A. Grzegorzcyk: *Recursive objects in all finite types*, Fundamenta Mathematica, LIV (1964) 73-93
- [8] J. Roger Hindley: *BCK-combinators and linear λ -terms have types*, Theoretical Computer Science, 64 (1989) 97-105
- [9] J. Roger Hindley: *Basic Simple Type Theory*, Cambridge University Press, 1997
- [10] J. Roger Hindley and Jonathan P. Seldin: *Introduction to Combinators and λ -Calculus*, Cambridge University Press, 1986

- [11] Neil D. Jones: *Computability and complexity from a programming perspective*, The Massachusetts Institute of Technology Press, 1997
- [12] Neil D. Jones: *Computability and complexity from a programming perspective*, Marktoberdorf 2001 Lecture Notes, 2001
- [13] S.C.Kleene: *Recursive functionals and quantifiers of finite types I*, Transactions of American Mathematical Society, 1959
- [14] S.C.Kleene: *Recursive Functionals and quantifiers of finite types II*, Transactions of American Mathematical Society, 1963
- [15] S.C.Kleene: *Recursive functionals and quantifiers of finite types revisited I*, Generalised Recursion Theory II, North-Holland Publishing Company, 1978
- [16] *Recursive functionals and quantifiers of finite types revisited II*, The Kleene Symposium, North-Holland Publishing Company, 1980
- [17] John R. Longley: *Notions of computability at higher types I*, 2001 Available at <http://www.dcs.ed.ac.uk/home/jrl/notions1.ps>
- [18] Johan Moldestad: *Computations in Higher Types*, Springer-Verlag, 1976
- [19] Richard A. Platek: *Foundations of Recursion Theory*, Doctoral Thesis, 1966
- [20] Harold Simmons: *Derivation and Computation* Cambridge University Press, 2000
- [21] Marie-France Thibault: *Pre-Recursive Categories*, Journal of Pure and Applied Algebra, (1982) 79-93
- [22] S. Troelstra and Schwichtenberg: *Basic Proof Theory*, Cambridge University Press, 2000
- [23] D.A.Turner: *Another algorithm for bracket abstraction*, The Journal of Symbolic Logic, Volume 44, Number 2, 1979