

Notes on the λ -calculus

J.R.B. Cockett
Department of Computer Science, University of Calgary,
Calgary, T2N 1N4, Alberta, Canada

October 19, 2010

1 Introduction

The λ -calculus was one of the first descriptions produced of computable functions. It grew out of an attempt by Alonso Church to provide a foundation for mathematics using functions as a basic building block. The attempt failed due to a form of Russell's paradox (which gave fixed point combinators). Church decided to extract a part of the calculus which worked perfectly well even if it did not achieve his original objective. This part is what we now know as the λ -calculus. In the course of developing this fragment he (and his students) realized that it was amazingly expressive and, in fact, could express all computable functions.

In this course we follow the realizations of Church and his students Rosser, Kleene, and Turing.

1.1 The λ -calculus

The λ -calculus is an algebra with a binary “application” operation and an “abstraction” operator. The application is sometimes written $f \bullet x$ (pronounced “ f applied to x ”) when we wish to emphasize the presence of a binary operation. However, it is more usually written and juxtaposition with an implicit association to the left (so that $xyz := (x \bullet y) \bullet z$). The abstraction operator binds a variable and is written using a lambda and this gives the calculus its name.

Thus terms in the calculus are formed using the following rules:

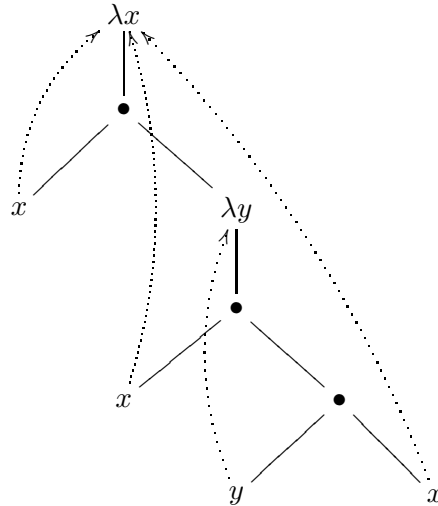
$$\boxed{\begin{array}{c} \frac{x \text{ Variable}}{x \text{ Term}} \text{Var} \\ \frac{M \text{ Term} \quad N \text{ Term}}{MN \text{ Term}} \text{app} \quad \frac{M \text{ Term} \quad x \text{ Variable}}{\lambda x.M \text{ Term}} \text{abst} \end{array}}$$

λ -term construction.

Two important notational conventions are used when writing λ -terms:

- Application associates to the left: $MNP := (MN)P$.
- Multiple abstractions are flattened: $\lambda xy.M := \lambda x(\lambda y.M)$.

In the term $\lambda x.M$ the variable x becomes bound and all free occurrences of x in the term M now refer to that binding. We may visualize this by drawing a λ -term as a tree with back arrows. For example the term $\lambda x.x(\lambda y.x(yx))$ may be portrayed as:



This suggests one could completely replace the bound variable names with a number which indicates how many λ -bindings one must pass through before one reaches the binding for the variable. These in programming language jargon are called “scope levels”. In this presentation the term $\lambda x.x(\lambda y.x(yx))$ would become $\lambda 0.0(\lambda 1.01)$. This way of indexing the variables is sometimes known as a “De Bruijn indexing” after the Dutch mathematician, Nicolaas de Bruijn who introduced a method of representing bound variables by numbers.

The free variables of a λ -term are those which occur in the term but are not bound. The free variables of a term are the smallest variable context in which we build can build the term. Here are the inference rules for variable contexts of a term: $V \vdash t$ Term means the term t can be built from the variable context V :

$\frac{x \text{ Variable}}{\{x\} \vdash x \text{ Term}} \text{ Var}$	
$\frac{V_1 \vdash M \text{ Term} \quad V_2 \vdash N \text{ Term}}{V_1 \cup V_2 \vdash MN \text{ Term}} \text{ app}$	$\frac{V \vdash M \text{ Term} \quad x \text{ Variable}}{V - \{x\} \vdash \lambda x.M \text{ Term}} \text{ abst}$
$\frac{W \vdash M \text{ Term} \quad W \subseteq V}{V \vdash M \text{ Term}} \text{ Weaken}$	

Variable contexts and λ -term construction.

Two λ terms are equivalent, this is called α -**equivalent** if they differ only in the way the bound variables are named. It should be clear that terms are α equivalent if and only if their de Bruijn indexed terms are the same.

1.2 Substitution

A key operation in the λ -calculus is substitution. By $M[N/x]$ shall be meant, intuitively, the term M with all free occurrences of x replaced by N . In practice this basic idea needs to be adjusted to avoid what is known as “variable capture” which occurs when a variable which is supposed to be free becomes bound as during substitution it is placed within the scope of a bound variable with the same name:

$$(\lambda x.xy)[x/y] \neq (\lambda x.xx)$$

Here the solution is to rename the bound variable (α -conversion) then do the replacement:

$$(\lambda x.xy)[x/y] =_{\alpha} (\lambda z.zy)[x/y] = \lambda z.zx$$

We may express the process of substitution as follows:

$$\begin{aligned} x[N/y] &= \begin{cases} N & x == y \\ x & \text{not } x == y \end{cases} \\ (MN)[P/x] &= M[P/x]N[P/x] \\ (\lambda y.M)[N/x] &= \begin{cases} \lambda y.M & x == y \\ \lambda y.(M[N/x]) & \text{not } y \in FV(N) \\ \lambda z.(M[z/y])[N/x] & z \text{ new} \end{cases} \end{aligned}$$

In the very last step we must do an α -conversion to avoid variable capture.

1.3 β -equality

We have already met α -equality: this, however, is really just a “book-keeping” equality due to having to make choices for variable names. The first significant equality is known as β -equality. It asserts:

$$(\lambda x.M)N = M[N/x].$$

From now on we shall take α -equality to be understood. This means we can define the β -equality relation by:

$\frac{V \vdash M \text{ Term}}{V \vdash M = M}$ Reflexive	
$\frac{V \vdash N = M}{V \vdash M = M}$ Symmetry	$\frac{V \vdash M = N \quad V \vdash N = P}{V \vdash M = P}$ Transitivity
$\frac{V \vdash (\lambda x.M)N \text{ Term}}{V \vdash (\lambda x.M)N = M[N/x]}$ β -equality	$\frac{V, x \vdash M = N}{V \vdash (\lambda x.M) = (\lambda x.N)}$ abst
$\frac{V \vdash M = M'}{V \vdash MN = M'N}$ Lapp	$\frac{V \vdash N = N'}{V \vdash MN = MN'}$ Rapp

β -equality between λ -terms.

$\frac{}{_ \text{context}} \text{ Empty}$	
$\frac{C[_] \text{ context}}{\lambda x.C[_] \text{ context}} \text{ abst}$	
$\frac{M \text{ term} \quad C[_] \text{ context}}{M C[_] \text{ context}} \text{ Lapp}$	$\frac{C[_] \text{ context} \quad N \text{ term}}{C[_] N \text{ context}} \text{ Rapp}$

Table 1: Construction of contexts

There is one other equality which is commonly considered called η -equality: $\lambda x.Mx = M$ (here x cannot occur in M). This is linked to “extensionality” because it says that if a function “behaves” just as N it must be N . For this note that $(\lambda x.Mx)N = MN$ for every N . These notes do not discuss this identity any further.

2 β -reduction and the Church-Rosser theorem

Using the above deduction system to determine the β -equality of λ -terms is highly inefficient as it involves a blind search. Church and his student Rosser determined that it was possible to direct the β -rule so it became a “ β -reduction”. A one step β -reduction is defined as follows:

$\frac{V \vdash (\lambda x.M)N \text{ Term}}{V \vdash (\lambda x.M)N \rightarrow M[N/x]} \beta\text{-reduction}$	
$\frac{V, x \vdash M \rightarrow N}{V \vdash (\lambda x.M) \rightarrow (\lambda x.N)} \text{ abst}$	
$\frac{V \vdash M \rightarrow M'}{V \vdash MN \rightarrow M'N} \text{ Lapp}$	$\frac{V \vdash N \rightarrow N'}{V \vdash MN \rightarrow MN'} \text{ Rapp}$

One step β -reduction.

We shall write $N \xrightarrow{*} M$ to indicate that to get from N to M requires 0 or more one step β reductions and $N \xrightarrow{+} M$ to indicate that to get from N to M requires 1 or more one step β reductions. Thus

$$N \xrightarrow{*} M := (N = M_0) \rightarrow M_1 \rightarrow \dots \rightarrow (M_n = M)$$

Another way of thinking of a one step β -reduction is to view it as a top-level β -reduction being performed at exactly one subterm. A *context* is a λ -term in which the root of a subtree has been selected. This is written $C[_]$ when a term N is put at this position we write $C[N]$:

Putting a term N in context is unlike a substitution as a free variable x of N can become bound by the context when the position of the root of the tree is in the scope of a binding for x . The notation of contexts allows a one step rewrite to be viewed as a top-level β -reduction in context:

$$C[(\lambda x.M)N] \rightarrow C[M[N/x]].$$

Church with his student Rosser proved the following theorem:

Theorem 2.1 (Church, Rosser) *In the λ -calculus two λ -terms N and M are equal (with respect to β -equality) if and only if there is a λ -term L and*



2.1 Normal form λ -terms

Before proving this theorem we discuss some very simple corollaries of it which – despite their simplicity – are very important and establish the consistency of the λ -calculus. A concern that one might have in dealing with the λ -calculus is that it is not consistent in the sense that one can prove that all terms are equal. The problem with the equality relation is that it really does not tell one when one *cannot* prove that two terms are equal. The Church-Rosser theorem, however, solves this problem rather neatly ...

Say that a λ -term is in **normal form** whenever there is no β -reduction leaving it. Here are some examples of terms in normal form:

- (a) Here are two very basic terms **True** := $\lambda xy.x$ and **False** := $\lambda xy.y$;
- (b) Terms which contain no λ -abstraction are always in normal form:

$$x, xy, xx, yy, xyxx, \dots$$

- (c) λ -abstractions of normal form terms are normal form terms:

$$\lambda x.x, \lambda yx.x, \lambda x.xx, \dots$$

- (d) Terms which are an application of a term M , which is in normal form *and* which has no λ -abstractions, applied to any term in normal form is itself in normal form.

$$x(\lambda x.x), y(\lambda x.x(\lambda x.xx)), \dots$$

- (e) Here is an infinite family of closed terms in normal form:

$$\lambda x.x, \lambda x.xx, \lambda x.xxx, \lambda x.xxxx, \lambda x.xxxxx, \dots$$

- (f) When two terms in normal form are applied to each other the result need not be in normal form:

$$(\lambda x.x)(\lambda x.x), \Omega := (\lambda x.xx)(\lambda x.xx), (\lambda x.xxx)(\lambda x.xxx), \dots$$

None of these are in normal form and in fact the second and third have infinite β -reduction sequences.

- $(\lambda x.x)(\lambda x.x) \rightarrow (\lambda x.x)$ is a reduction to normal form.
- $\Omega := (\lambda x.xx)(\lambda x.xx) \rightarrow xx[(\lambda x.xx)/x] = (\lambda x.xx)(\lambda x.xx)$ is a reduction of Ω to itself and thus Ω cannot have a normal form.
- The last term also has an infinite reduction sequence in which the term actually grows in size:

$$\begin{aligned} (\lambda x.xxx)(\lambda x.xxx) &\rightarrow xxx[(\lambda x.xxx)/x] \\ &= (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow (xxx[(\lambda x.xxx)/x])(\lambda x.xxx) \\ &= (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow \dots \end{aligned}$$

An important corollary of the Church-Rosser theorem is:

Corollary 2.2 *Two terms which are in normal form which are not α -equivalent are distinct in the λ -calculus.*

It is clear that True and False are closed terms which are not α -equivalent and so:

Corollary 2.3 *The λ -calculus is consistent in the sense that there are at least two unequal closed terms. In fact, there are infinitely many distinct terms.*

Another important result is the “normal form theorem” which says:

Theorem 2.4 *A λ -term is β -equivalent to at most one normal form term.*

PROOF: To prove this requires two observations: first if M is β -equivalent to a normal form term N then there is a reduction sequence $M \xrightarrow{*} N$. We may see this using the Church-Rosser theorem as follows. As M is β -equivalent to N there is an L to which they both reduce. But N cannot be reduced whence $N = L$ so that $M \xrightarrow{*} N$.

Next suppose that M reduces to two normal forms N_1 and N_2 then immediately N_1 and N_2 are β -equivalent so that there is an L to which both reduce. However as both are in normal form $N_1 = L = N_2$ establishing the result. \square

2.2 The proof of the Church-Rosser theorem

At the time it was proven the Church-Rosser theorem was regarded as a fairly major achievement. In fact, it lead to a whole field of study called *term rewriting* (which we discuss later). Furthermore, the original proof was quite long. Here we shall present a proof due to Barendregt [1] which uses a marking argument. The proof starts with some simple observations which reduce the problem:

OBSERVATION I:

Two λ -terms are equal if and only if there is a zig-zag of β -reductions:

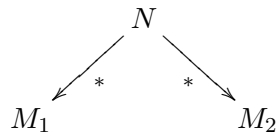
$$M = N_0 \xleftarrow{*} N_1 \xrightarrow{*} N_2 \xleftarrow{*} \dots \xrightarrow{*} N_p = N$$

(This is immediate!)

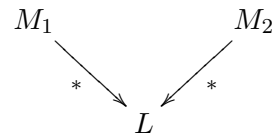
OBSERVATION II:

If we can prove:

Proposition 2.5 (Confluence) *Given any divergence of λ -terms and β reductions:*

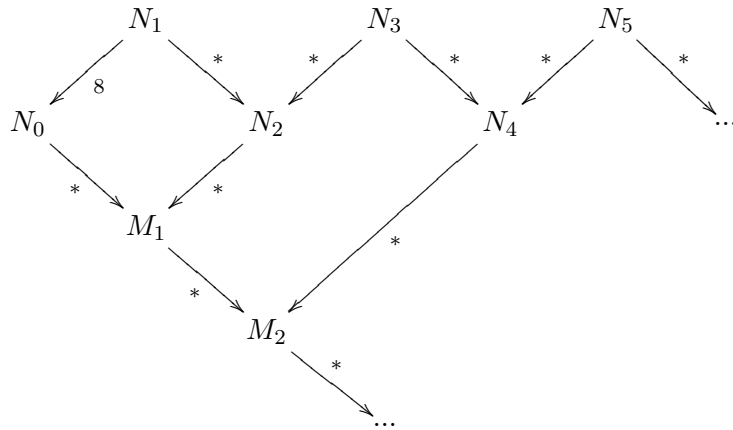


there is a convergence



Then the Church Rosser theorem holds!

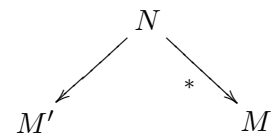
This is because we can fill in any chain as above:



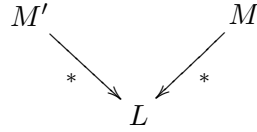
OBSERVATION III:

If we can prove:

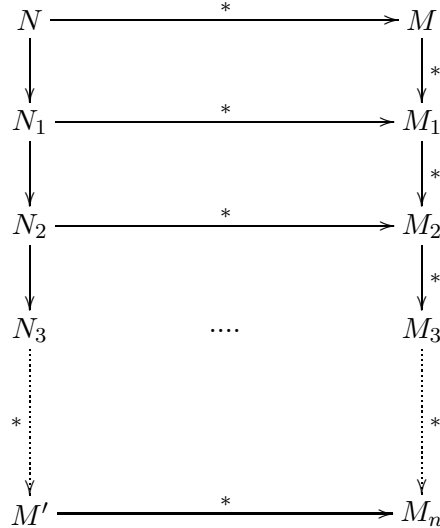
Lemma 2.6 (Strip lemma) *Given any divergence of λ -terms and β reductions:*



(note the one-step reduction) there is a convergence



Then we can prove confluence! This is because we can break down confluence into a series of one steps down the left hand reduction:



Unfortunately even the strip lemma is not so easy to prove. This is where we shall employ a marking argument originally due to Barendregt.

2.3 Barendregt's marking argument

We may define marked λ -terms by:

$$M \rightarrow x \mid \lambda x.M \mid MM \mid (\underline{\lambda}x.M)M$$

There is an obvious underlying map to the “unmarked” (ordinary) λ -terms by just dropping the marking:

$$U((\underline{\lambda}x.M)N) = (\lambda x.M)N$$

There is also a by-value evaluation of a marked λ -term:

$$\begin{aligned}
 V(x) &= x \\
 V(\lambda x.M) &= \lambda x.V(M) \\
 V(MN) &= V(M)V(N) \\
 V((\underline{\lambda}x.M)N) &= V(M)[V(N)/x]
 \end{aligned}$$

This can also be written as an inference system where the by-value reduction is presented as $N \rightsquigarrow V(N)$:

$$\boxed{
\begin{array}{c}
\frac{x \text{ var}}{x \rightsquigarrow x} \qquad \frac{M \rightsquigarrow M' \quad N \rightsquigarrow N'}{MN \rightsquigarrow M'N'} \\
\frac{N \rightsquigarrow N'}{\lambda x.N \rightsquigarrow \lambda x.N'} \qquad \frac{N \rightsquigarrow N' \quad M \rightsquigarrow M'}{(\lambda x.N)M \rightsquigarrow N'[M'/x]}
\end{array}
}$$

Marked by-value reduction

It is easy to see that this evaluation always terminates and $V(M)$ contains no marked terms. Furthermore there is a rewriting sequence mimicking the evaluation;

$$\begin{array}{ccc}
& M & \\
& \swarrow & \\
U(M) & \xrightarrow{*} & V(M)
\end{array}$$

Consider again the strip lemma we may view the one step rewrite as a marked reduction:

$$\begin{array}{ccc}
N & \xrightarrow{*} & M \\
\downarrow & & \\
M' & & \\
U(N') = N & \xrightarrow{*} & U(M) \\
\downarrow & & \\
V(N') = M' & &
\end{array}$$

we can then prove the strip lemma by proving the following proposition for marked reductions:

Proposition 2.7 *In the marked λ -calculus (with the evident reductions) we have:*

$$\begin{array}{ccc}
M & \longrightarrow & M' \\
\downarrow * & & \downarrow * \\
V(M) & \dashrightarrow_* & V(M')
\end{array}$$

Given this we may break down the version of the strip lemma above as:

$$\begin{array}{ccccccc}
M & \longrightarrow & M_1 & \longrightarrow & M_2 & \longrightarrow & \dots & \longrightarrow & M' \\
\downarrow * & & \downarrow * & & \downarrow * & & & & \downarrow * \\
M & \dashrightarrow_* & M_1 & \dashrightarrow_* & M_2 & \dashrightarrow_* & \dots & \dashrightarrow_* & M'
\end{array}$$

and thus obtain the desired result.

The proof of the proposition proceeds by considering a series of cases for the single step rewrite:

At the root(unmarked):

$$\begin{array}{ccc}
(\lambda x.M)N & \longrightarrow & M[N/x] \\
\downarrow * & & \downarrow * \\
(\lambda x.V(M))V(N) & \longrightarrow & V(M)[V(N)/x] = V(M[N/x])
\end{array}$$

where we must use the substitution lemma 2.8 below for the last step.

At the root (marked):

$$\begin{array}{ccc}
(\underline{\lambda}x.M)N & \longrightarrow & M[N/x] \\
\downarrow * & & \downarrow * \\
(\underline{\lambda}x.V(M))V(N) & & \\
\downarrow & & \\
V(M)[V(N)/x] & \equiv & V(M)[V(N)/x]
\end{array}$$

In context (unmarked): The argument is more complex if the reduction is not at the root:

$$\begin{array}{ccc}
C[(\underline{\lambda}x.M)N] & \longrightarrow & C[M[N/x]] \\
\downarrow * & & \downarrow * \\
C[(\underline{\lambda}x.V(M))V(N)] & \longrightarrow & C[V(M)[V(N)/x]] \\
\downarrow * & & \downarrow * \\
V(C[(\underline{\lambda}x.V(M))V(N)]) & \xrightarrow{*} & V(C[V(M)[V(N)/x]])
\end{array}$$

Here we use the fact that the by value evaluation of the term in (marked) context will evaluate the inner term before it evaluates the context. This allows us to separate the evaluation into evaluating the inner term and then evaluating the result in context. We must establish that the lower square can be completed: this done in lemma 2.9.

In context (marked):

$$\begin{array}{ccc}
C[(\underline{\lambda}x.M)N] & \longrightarrow & C[M[N/x]] \\
\downarrow * & & \downarrow * \\
C[V(M)[V(N)/x]] & \equiv & C[V(M)[V(N)/x]] \\
\downarrow * & & \downarrow * \\
V(C[V(M)[V(N)/x]]) & \equiv & V(C[V(M)[V(N)/x]])
\end{array}$$

To complete the proof we have two lemmas to prove:

Lemma 2.8

$$V(M[N/x]) = V(M)[V(N)/x].$$

PROOF: We shall prove this by structural induction on M :

M is a variable: If $M = x$ we have $V(x[N/x]) = V(N) = x[V(N)/x] = V(x)[V(N)/x]$. If $M = y$ and $y \neq x$ then we have $V(y[N/x]) = V(y) = y = y[V(N)/x] = V(y)[V(N)/y]$.

M is an application: If $M = N_1N_2$ then

$$\begin{aligned}
V((N_1N_2)[N/x]) &= V(N_1[N/x]N_2[N/x]) \text{ defn of substitution} \\
&= V(N_1[N/x])V(N_2[N/x]) \text{ defn of marked evaluation} \\
&= V(N_1)[V(N)/x]V(N_2)[V(N)/x] \\
&\quad \text{structural induction } (N_1 \text{ and } N_2 \text{ are smaller terms)} \\
&= (V(N_1)V(N_2))[V(N)/x] \text{ defn of substitution} \\
&= (V(N_1N_2))[V(N)/x] \text{ defn of marked evaluation}
\end{aligned}$$

M is an unmarked abstraction: Suppose $M = \lambda y.N'$ then (assuming without loss of generality $x \neq y$):

$$\begin{aligned}
V((\lambda y.N')[N/x]) &= V(\lambda y.(N'[N/x])) \text{ defn of substitution} \\
&= \lambda y.V(N'[N/x]) \text{ defn of marked evaluation} \\
&= \lambda y.V(N')[V(N)/x] \text{ structural induction } (N' \text{ is a smaller term)} \\
&= V(\lambda y.N')[V(N)/x] \text{ defn of marked evaluation}
\end{aligned}$$

M is a marked λ -abstraction: Suppose $M = (\underline{\lambda}y.M')N'$ then

$$\begin{aligned}
V(((\underline{\lambda}y.M')N')[N/x]) &= V(((\underline{\lambda}y.M')[N/x])(N'[N/x])) \text{ defn of substitution} \\
&= V((\underline{\lambda}y.(M'[N/x]))(N'[N/x])) \text{ defn of substitution} \\
&= V(M'[N/x])[V(N')[V(N)/x]/y] \text{ defn of marked evaluation} \\
&= (V(M')[V(N)/x])[V(N')[V(N)/x]/y] \text{ structural induction} \\
&= (V(M'))[(V(N')/y)][V(N)/x] \text{ property of substitution} \\
&= V((\underline{\lambda}y.M')N')[V(N)/x] \text{ defn of marked evaluation}
\end{aligned}$$

□

We now have to examine more carefully how to build a marked context, in particular, it is useful to have an inductive definition of the construction of these as then we can use proofs by structural induction. Here is the way in which a marked context is built (note the change from table 1):

- The empty context is a context for M : $C[[M]] = M$
- Adding application: if $C[[M]]$ is a context for M then

$$NC[[M]] \text{ and } C[[M]]L$$

are contexts for M .

- Unmarked abstraction: if $C[[M]]$ is a context for M then $\lambda x.C[[M]]$ is a context for M . Note here any $x \in M$ becomes bound.
- Marked abstraction: if $C[[M]]$ is a context for M then

$$(\underline{\lambda}x.C[[M]])N \text{ and } (\underline{\lambda}x.N)C[[M]]$$

are context for M . Note in the first of these terms x becomes bound in M .

Lemma 2.9

(i) $V(C\llbracket M \rrbracket) = V(C\llbracket V(M) \rrbracket)$;

(ii) If $M \rightarrow N$ is a rewrite between unmarked terms then:

$$\begin{array}{ccc} C\llbracket M \rrbracket & \longrightarrow & C\llbracket N \rrbracket \\ * \downarrow & & \downarrow * \\ V(C\llbracket M \rrbracket) & \xrightarrow{*} & V(C\llbracket N \rrbracket) \end{array}$$

PROOF: We prove both parts by a structural induction on building the context:

(i) If the context is empty then as $V(V(M)) = V(M)$ the result follows. If the context is built using an application we have (for left application):

$$\begin{aligned} V(NC\llbracket M \rrbracket) &= V(N)V(C\llbracket M \rrbracket) \text{ defn of marked evaluation} \\ &= V(N)V(C\llbracket V(M) \rrbracket) \text{ structural induction} \\ &= V(NV(C\llbracket V(M) \rrbracket)) \text{ defn of marked evaluation} \end{aligned}$$

For unmarked abstractions we have:

$$\begin{aligned} V(\lambda x.C\llbracket M \rrbracket) &= \lambda x.V(C\llbracket M \rrbracket) \text{ defn of marked evaluation} \\ &= \lambda x.V(C\llbracket V(M) \rrbracket) \text{ structural induction} \\ &= V(\lambda x.C\llbracket V(M) \rrbracket) \text{ defn of marked evaluation} \end{aligned}$$

Finally for the first case of marked abstraction we have:

$$\begin{aligned} V((\lambda y.C\llbracket M \rrbracket)N) &= V(C\llbracket M \rrbracket)[V(N)/y] \text{ defn of marked evaluation} \\ &= V(C\llbracket V(M) \rrbracket)[V(N)/y] \text{ structural induction} \\ &= V((\lambda y.C\llbracket V(M) \rrbracket)N) \text{ defn of marked evaluation} \end{aligned}$$

The second is similar.

(ii) The result is clear for the empty context. For abstraction on the left (on the right is similar) we must produce a rewrite sequence

$$V(LC\llbracket M \rrbracket) \rightarrow V(LC\llbracket M \rrbracket)$$

on the assumption that we have a sequence of rewrites $V(C\llbracket M \rrbracket) \rightarrow V(C\llbracket M \rrbracket)$. But as $V(LC\llbracket M \rrbracket) = V(L)V(C\llbracket M \rrbracket)$ and $V(LC\llbracket M \rrbracket) = V(L)V(C\llbracket M \rrbracket)$ then we may use the original rewrite sequence as applied on the right of the application.

A similar argument works for extending the context by an unmarked lambda.

The difficult case is when the context is extended by a marked λ . There are two cases:

$$\begin{array}{ccc}
(\underline{\lambda}y.C[[M]])M' & \longrightarrow & (\underline{\lambda}y.C[[N]])M' \\
\downarrow * & & \downarrow * \\
V(C[[M]][M'/y]) & & V(C[[N]][M'/y]) \\
\parallel & & \parallel \\
V(C[[M]][V(M')/y]) & \xrightarrow{*} & V(C[[N]][V(M')/y])
\end{array}$$

Here by assumption we have a rewrite sequence $C[[M]] \xrightarrow{*} C[[N]]$ and the last line simply substitutes this (note the step down uses the substitution lemma above).

Finally we have the last case (which is also the hardest case):

$$\begin{array}{ccc}
(\underline{\lambda}y.M')C[[M]] & \longrightarrow & (\underline{\lambda}y.M')C[[N]] \\
\downarrow * & & \downarrow * \\
V(M'[C[[M]]/y]) & & V(M'[C[[N]]/y]) \\
\parallel & & \parallel \\
V(M')[V(C[[M]]/y)] & \xrightarrow{*} & V(M')[V(C[[N]]/y)]
\end{array}$$

In this case wherever y occurs we must do the rewrite sequence $C[[M]] \xrightarrow{*} C[[N]]$. It is then a matter of going through each occurrence of y and doing the rewrites required at that occurrence. This gives a possibly huge number of rewrites as y can occur often in the term $V(M')$... but it is still a finite number of occurrences so the parallel rewriting of subterms can be turned into a sequential rewriting!

□

This completes the proof of the Church-Rosser theorem in some detail.

3 Representing data in the λ -calculus

To represent computations in the λ -calculus one must first agree on how to represent data. The two most basic datatype are the **Booleans** and the ability to form **products**. The element (of the Booleans are **True** and **False**. The elements of a product of two sets are all the pairs $\langle a, b \rangle$ where a is from the first set and b is from the second. However, having data is of no value if one cannot *use* the data. Thus, for Booleans, a key feature is that one can use them as a condition to control a program; for products one must be able to extract the information in the components – that is *project* to obtain the entries.

Next one wants to be able to represent more general (infinite) inductive data such as the natural numbers, lists, and trees. These, it turns out can all be represented very uniformly in the λ -calculus. At the time these ideas were being developed - in the 1930s - representing the natural numbers was of crucial significance as it was for functions between the natural numbers that the first notions of being “computable” or (partial) “recursive” arose. We shall trace these developments showing that, in the λ -calculus, one can represent *all* the computable functions.

Another important way to encode computable functions was to use a Turing machines. Once one has general inductive data (lists in particular) and general recursion it is straightforward to program Turing machines in the λ -calculus. However, this does mean that it is useful to know how these encodings work in the λ -calculus.

3.1 Booleans

In the λ -calculus it is traditional to set:

$$\begin{aligned}\text{True} &:= \lambda xy.x \\ \text{False} &:= \lambda xy.y \\ \text{If } zt f &= zt f \Leftrightarrow \text{If} = \lambda zt f.zt f\end{aligned}$$

In the last of these we are using a the more natural form for the definition of **If**: this uncurries function symbol being defined to remove the outer lambda abstractions. The curried for, which gives **If** as a close expression is given on the right.

Given the conditional one may then define all the usual functions on Booleans:

$$\begin{aligned}\text{Not } b &= \text{If } b \text{ False True} \\ \text{And } b_1 b_2 &= \text{If } b_1 (\text{If } b_2 \text{ True False}) \text{ False} \\ \text{Or } b_1 b_2 &= \text{If } b_1 \text{ True } (\text{If } b_2 \text{ True False})\end{aligned}$$

3.2 Products

Here we must encode the ability to form pairs:

$$\begin{aligned}\langle f, g \rangle &:= \lambda h.hfg \\ \pi_0 &:= \lambda z.z(\lambda xy.x) \\ \pi_1 &:= \lambda z.z(\lambda xy.y)\end{aligned}$$

Note that when we project a pair we have

$$\begin{aligned}
 \pi_0 \langle f, g \rangle &:= (\lambda z. z(\lambda xy. x)) \langle f, g \rangle \\
 &\rightarrow \langle f, g \rangle (\lambda xy. x) \\
 &:= (\lambda h. hfg) (\lambda xy. x) \\
 &\rightarrow (\lambda xy. x) fg \\
 &\rightarrow (\lambda y. f) g \\
 &\rightarrow f
 \end{aligned}$$

Notice that with pairing and Boolean operations one can represent all finite sets (as binary numbers of a fixed width) and then one can define any function using Boolean expressions. This does not allow one to define functions on infinite sets, however.

3.3 The natural numbers

The first significant difficulty is to represent inductive data such as the natural numbers in the λ -calculus. The following is the Church encoding for the natural numbers:

$$\begin{aligned}
 0 &:= \lambda xy. x \\
 1 &:= \lambda xy. yx \\
 2 &:= \lambda xy. y(yx) \\
 3 &:= \lambda xy. y(y(yx)) \\
 &\dots
 \end{aligned}$$

They are called the “Church numerals”.

This is by no means the only way to do it but it is the most natural way in a sense we shall soon make clear. But what are **Zero**, **Succ**, and the **fold** or iteration functions over numbers? Here they are:

$$\begin{aligned}
 \text{Zero} &= \lambda xy. x \\
 \text{Succ } n &= \lambda xy. y(nxy) \\
 \text{Fold } zfn &= nzf
 \end{aligned}$$

This allows us to define addition and multiplication:

$$\begin{aligned}
 \text{Add } n m &= \text{Fold } m \text{ Succ } n \\
 \text{Mult } n m &= \text{Fold } \text{Zero} (\text{Add } m) n
 \end{aligned}$$

To illustrate how this works let us add one and one:

$$\begin{aligned}
\text{Add } \mathbf{1} \ \mathbf{1} &:= \text{Fold } \mathbf{1} \ \text{Succ } \mathbf{1} \\
&:= \mathbf{1} \ \mathbf{1} \ \text{Succ} \\
&:= (\lambda xy.yx) \ \mathbf{1} \ \text{Succ} \\
&\rightarrow \text{Succ } \mathbf{1} \\
&:= (\lambda xy.y(\mathbf{1} \ xy)) \\
&:= (\lambda xy.y((\lambda xy.yx)xy)) \\
&\rightarrow (\lambda xy.y(yx)) \\
&=: \mathbf{2}
\end{aligned}$$

A problem which gave Church and his student Kleene pause was the question of how to encode the predecessor function. This seems like an easy problem, however, removing one successor is not so easy when you are trying to do it purely from the primitives the λ -calculus provides. The story goes that Kleene had the inspiration when he was sitting in the dentists chair ... here is what he did:

$$\text{Pred } n = \pi_0(\text{Fold } \langle \mathbf{0}, \mathbf{0} \rangle (\lambda z.\langle \text{Succ}(\pi_1 z), \pi_1 z \rangle) n)$$

Why is the predecessor function so important? A basic ability which is required in many programs is the ability to determine whether two numbers are equal. The basic scheme for this was to use truncated subtraction, or the `monus` function, and to test whether the two ways of doing this subtraction were both zero (using the `iszero` function. However, the easiest way to define truncated subtraction was to use the predecessor function. Here are the programs:

$$\begin{aligned}
\text{iszero } n &= \text{Fold True } (\lambda x.\text{False}) \ n \\
\text{monus } n \ m &= \text{Fold } n \ \text{Pred } m \\
\text{Eqnm} &= \text{And } (\text{iszero } (\text{monus } n \ m)) \ (\text{iszero } (\text{monus } m \ n)).
\end{aligned}$$

The predecessor function, of course, is easily programmed if one has a case function as in Haskell:

$$\text{Pred } x = \text{case } x \ \text{of} \ \left| \begin{array}{l} \text{Zero} \quad \mapsto \ \text{Zero} \\ \text{Succ } x' \mapsto \ x' \end{array} \right.$$

Thus, a more fundamental problem is to program the `case` construct. Not surprisingly this requires a similar trick to the predecessor function. Our first step is to remove the “syntactic sugar” from the case statement to obtain a combinator:

$$\text{case } n \ \text{of} \ \left| \begin{array}{l} \text{Zero} \quad \mapsto \ t_1 \\ \text{Succ } n' \mapsto \ t_2 \end{array} \right. := \text{Case } n \ t_1 \ (\lambda n'.t_2)$$

Next we show how to implement this combinator:

$$\text{Case } n \ t_1 \ f := \pi_1(\text{fold}(\text{Nil}, t_1)(\lambda x.\langle \text{Succ}(\pi_0 x), f(\pi_0 x) \rangle)).$$

If we allow pattern matching for products we can present this a little more neatly as:

$$\text{Case } n \ t_1 \ f := \pi_1 (\text{fold} \langle \text{Nil}, t_1 \rangle (\lambda \langle x_0, x_1 \rangle. \langle \text{Succ } x_0, f \ x_0 \rangle)).$$

Note that there is an essential use of products in defining this function too..

3.4 Representing list

One way to prove that one can program all computable functions is to show that one can program a Turing machine. To do this the most difficult aspect is to represent the tape: a standard way of doing this is to view it as a list of cells left of head, the cell under the head, and a list of cells to the right of the head. Moving the head to the left involves, for example, pushing the current cell under the head onto the right list and popping the top element off the left list to become the new cell under the head. If the list to be popped is empty then one inserts a black cell under the head. Thus, once one has lists one can, in principle, program the basic mechanism of any Turing machine.

There are two problems to surmount: first one must be able to represent lists in the λ -calculus and second one must be able to iterate its “step” until the machine reaches a halt state. We shall see how this can be done soon. Modulo this aspect one will then be confident that all computable functions can be programmed.

Here is how lists are represented:

$$\begin{aligned} [] &= \lambda nc.n \\ [a_1] &= \lambda nc.ca_1n \\ [a_1, a_2] &= \lambda nc.ca_1(ca_2n) \\ [a_1, a_2, a_3] &= \lambda nc.ca_1(ca_2(ca_3n)) \\ &\dots \end{aligned}$$

Here are the basic functions associated with lists:

$$\begin{aligned} \text{Nil} &= \lambda nc.n \\ \text{Cons } a \ l &= \lambda nc.ca(lnc) \\ \text{FoldList } v \ g \ l &= lvg \end{aligned}$$

We can than define the basic functions associated with list such as appending lists, mapping over lists, etc.

$$\begin{aligned} \text{append } l_1 \ l_2 &= \text{FoldList } l_2 \ \text{Cons } l_1 \\ \text{MapList } f \ l &= \text{FoldList } \text{Nil} \ (\lambda az.\text{Cons } (f \ a) \ z)l \end{aligned}$$

Of course, as for the natural numbers, there is still the problem of programming the case construct. We start by removing the syntactic sugar:

$$\text{case } l \ \text{of} \ \left| \begin{array}{ll} \text{Nil} & \mapsto \ t_1 \\ \text{Cons } a \ as & \mapsto \ t_2 \end{array} \right. := \text{CaseList } l \ t_1 \ (\lambda a \ as.t_2)$$

Then `CaseList` can be programmed as follows:

$$\text{CaseList } l \ t \ f := \pi_1 (\text{FoldList } \langle \text{Nil}, t \rangle (\lambda a \langle l, _ \rangle. \langle \text{Cons } a \ l, f \ a \ l \rangle) \ l)$$

Again notice the essential use of the product and the use of (product) patterns. This allow us, for example, to program the “tail” function for a list as follows:

$$\text{Tail } l := \text{case } l \text{ of } \left\{ \begin{array}{l} \text{Nil} \quad \mapsto \text{Nil} \\ \text{Cons } a \ as \mapsto as \end{array} \right.$$

3.5 Representing trees

How do we represent trees? By now you may be suspecting that there is a definite pattern emerging! Before describing the general pattern let us do one more simple example. Start with the Haskell data declaration:

$$\text{data Tree } a = \left\{ \begin{array}{l} \text{Leaf } a \\ \text{Node } (\text{Tree } a) (\text{Tree } a) \end{array} \right.$$

The first step is to write down some simple trees:

$$\begin{aligned} \text{Leaf } x &= \lambda l \ n. l \ x \\ \text{Node } (\text{Leaf } x_1) (\text{Leaf } x_2) &= \lambda l \ n. n \ (l \ x_1) \ (l \ x_2) \\ &\dots \end{aligned}$$

Now one defines:

$$\begin{aligned} \text{Leaf } x &= \lambda l \ n. l \ x \\ \text{Node } t_1 \ t_2 &= \lambda l \ n. n \ (t_1 \ l \ n) \ (t_2 \ l \ n) \\ \text{FoldTree } f \ g \ t &= t \ f \ g. \end{aligned}$$

The last definition indicates that the objective of the representatin is to make a tree into a curried fold function for the tree. Thus, in a very real sense data in the λ -calculus is potential computation – as it curried computation.

Finally, and somewhat more tricky, is to write down the `case` function for a tree. First, remove the syntactic sugar:

$$\text{case } t \text{ of } \left\{ \begin{array}{l} \text{Leaf } x \quad \mapsto v_1 \\ \text{Node } t_1 \ t_1 \mapsto v_2 \end{array} \right. := \text{CaseTree } t \ (\lambda x. v_1) \ (\lambda t_1 \ t_2. v_2)$$

Next write down the `CaseTree` function using products:

$$\text{CaseTree } t \ f \ g := \pi_1 (\text{FoldTree } (\lambda x. \langle \text{Leaf } x, f \ x \rangle) (\lambda \langle t_1, _ \rangle \langle t_1, _ \rangle. \langle \text{Node } t_1 \ t_2, g t_1 t_2 \rangle) \ t)$$

3.6 Inductive data in general

Representing data in the λ -calculus is just a translation process. Unfortunately, this does not mean it is particularly easy BUT once you have tried it a few times you will probably be able to appreciate the general process. Here is an explicit description of the process:

1. Start with an arbitrary datatype definition:

$$\text{data } D \ a_1 \ \dots \ a_n = \begin{cases} \text{Cons}_1 T_{11} \ a_1 \ \dots \ a_n \ (D \ a_1 \ \dots \ a_n) \ \dots \ T_{1m_1} \ a_1 \ \dots \ a_n \ (D \ a_1 \ \dots \ a_n) \\ \dots \\ \text{Cons}_p T_{p1} \ a_1 \ \dots \ a_n \ (D \ a_1 \ \dots \ a_n) \ \dots \ T_{pm_p} \ a_1 \ \dots \ a_n \ (D \ a_1 \ \dots \ a_n) \end{cases}$$

First notice that there are p constructors: each constructor has a type. This is rather hidden in the Haskell syntax so lets make it explicit:

$$\text{Cons}_i :: (T_{i1} \ a_1 \ \dots \ a_n \ (D \ a_1 \ \dots \ a_n)) \rightarrow \dots \rightarrow (T_{im_i} \ a_1 \ \dots \ a_n \ (D \ a_1 \ \dots \ a_n)) \rightarrow (D \ a_1 \ \dots \ a_n)$$

2. Notice also that we are using types T_{ij} which we are assuming have already been introduced. Furthermore, we shall assume that with each of these already introduced types there is a MapT_{ij} function:

$$\frac{f :: x \rightarrow y}{\text{MapT}_{ij} \ f :: T_{ij} \ a_1 \ \dots \ a_n \ x \rightarrow T_{ij} \ a_1 \ \dots \ a_n \ y.}$$

3. Now we know we must construct the fold function: in this each constructor turns into a function. It is useful to think about the types of these functions and of the fold. The former are obtained by replacing the type of the data being defined by an arbitrary type, b :

$$c_i :: (T_{i1} \ a_1 \ \dots \ a_n \ b) \rightarrow \dots \rightarrow (T_{im_i} \ a_1 \ \dots \ a_n \ b) \rightarrow b$$

This makes the type of the fold look:

$$\text{foldD} :: \begin{array}{l} ((T_{11} \ a_1 \ \dots \ a_n \ b) \rightarrow \dots \rightarrow (T_{1m_1} \ a_1 \ \dots \ a_n \ b) \rightarrow b) \\ \rightarrow \dots \\ \rightarrow ((T_{p1} \ a_1 \ \dots \ a_n \ b) \rightarrow \dots \rightarrow (T_{pm_p} \ a_1 \ \dots \ a_n \ b) \rightarrow b) \\ \rightarrow b \end{array}$$

This just sorts out the typing involved but you must be clear on this to start with!

4. Now the next step is to represent elements of this data type in the λ -calculus and the basic idea is to represent it as an abstracted fold. This means that in the end you will know that the actual fold becomes:

$$\text{foldD} \ c_1 \ \dots \ c_p \ d = d \ c_1 \ \dots \ c_p.$$

Of course you are done if you can write down the constructors as then you can inductively construct representing elements of the datatype (and so say inductively what it means to be a element of this datatype).

5. So what do the constructors look like?

$$\text{Cons}_i \ y_1 \ \dots \ y_{m_i} := \lambda \ c_1 \ \dots \ c_p. c_i(\text{MapT}_{i1}(\lambda y. y \ c_1 \ \dots \ c_m)y_1) \ \dots \ (\text{MapT}_{im_i}(\lambda y. y \ c_1 \ \dots \ c_m)y_{m_i})$$

The use of the map functions may initially be surprising but, recall, that one needs to transmit the “internal” names of the constructors to the occurrences of the data type inside each type: this is what the map does.

So with this pattern in mind it is possible to write down representative elements of any data type. Clearly, as the data type becomes more complex so the structure of the elements becomes more complex. Of course, this is exactly why you do not want to write down such elements but rather be able to rely on the pattern!

3.7 Rose trees

A small example to illustrate the necessity of the map in describing the constructors. Consider the data type:

$$\text{data Rose } f \ v = \begin{cases} \text{Var } v \\ \text{Rs } f \ (\text{List } (\text{Rose } a \ v)) \end{cases}$$

This is a Rose tree with “function symbols” at the nodes and “variables” at the leaves. The constructors, using the above process, are:

$$\begin{aligned} \text{Var } x &= \lambda v \ r. \text{vx} \\ \text{Rs } a \ l &= \lambda v \ r. \text{r } a \ (\text{MapList } (\lambda t. t \ v \ r) \ l) \end{aligned}$$

4 Recursion using fixed points

So far we have only shown how to represent all the (higher-order) primitive recursive functions. What we have not done is to mimic the ability a program has to recursively call itself or to iterate (possibly forever). Fundamental in this is the use of fixed point combinators.

4.1 Fixed points

A fixed point combinator Y has the property that for every f we have $Yf = f(Yf)$. this can be used to encode a recursive call to a function. As suppose we define $gx = Mg$ where M is some combinator which reuses the function g recursively then we can define $g = Y(\lambda gx.Mg)$ this means:

$$\begin{aligned} gN &:= (Y(\lambda gx.Mg)) \\ &\rightarrow (\lambda x.M(Y(\lambda gx.Mg))x)N \\ &\rightarrow M[N/x](Y(\lambda gx.Mg)) \end{aligned}$$

Note how this presents the body of the recursive call to act on the argument N once the Y combinator has been unwrapped if we do a leftmost outermost reduction.

There are two well known examples of fixed point combinators (although there are infinitely many!):

$$\begin{aligned} Y &= \lambda f.(Af)(Af) \\ Af &= \lambda x.f(xx) \\ Y' &= A'A' \\ A' &= \lambda xy.y(xxy) \end{aligned}$$

The first is due to Church the second is due to Turing. To show that Y is a fixed point combinator we have:

$$\begin{aligned} (Yf) &:= (Af)(Af) := (\lambda x.f(xx))(Af) \\ &\rightarrow f((Af)(Af)) := f(Yf) \end{aligned}$$

To show that Y' is a fixed point combinator we have:

$$\begin{aligned} Y'f &:= \underline{(\lambda xy.y(xxy))(\lambda xy.y(xxy))}f \\ &\rightarrow (\lambda y.y(Y'y))f \\ &\rightarrow f(Y'f) \end{aligned}$$

How do we know that these two expressions are not equivalent?

$$\begin{aligned} Y &= \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \quad \text{due to Curry} \\ Y' &= (\lambda xy.y(xxy))(\lambda xy.y(xxy)) \quad \text{due to Turing} \end{aligned}$$

To see that they are not the same look at the β -reductions of Y setting $Af = \lambda x.f(xx)$

$$Y \rightarrow \lambda f.f((Af)(Af)) \rightarrow \lambda f.f(f((Af)(Af))) \rightarrow \dots \rightarrow \lambda f.f^n((Af)(Af)) \rightarrow \dots$$

Now consider Turing's fixed point combinator set, $X = \lambda x f.f(xxf)$, then

$$Y' = XX \rightarrow \lambda f.f(XXf) \rightarrow \dots \rightarrow \lambda f.f^n(XXf) \rightarrow \dots$$

so they never have a common β -reduction and so they cannot be equal.

Here is a challenge: can you prove that there are infinitely many distinct fixed point?

4.2 Using fixed points in recursion

To implement the recursive (non-terminating) function $\mathbf{f} \ n = \mathbf{n} : (\mathbf{f} \ (\mathbf{n}+1))$ we set $\mathbf{f} = Y(\lambda f n.n : (f(n+1)))$ reducing gives the desired recursive behavior:

$$\begin{aligned} \mathbf{f} n &:= (Y(\lambda f n.n : (f(n+1))))n \\ &\rightarrow (\lambda f n.n : (f(n+1)))(Y(\lambda f n.n : (f(n+1))))n \\ &\rightarrow n : ((Y(\lambda f n.n : (f(n+1))))n + 1) \\ &=: n : (\mathbf{f} n + 1) \end{aligned}$$

5 Computability and the λ -calculus

When people first started studying computability it seemed very natural for them to focus on the question of whether a function $\mathbb{N}^n \rightarrow \mathbb{N}$ on the natural numbers was “computable” – in the sense that it could be calculated by machine. Thus, the first notions of computability emerged here and indeed Kleene developed a simple theorem which we shall use:

Theorem 5.1 *The computable (partial) functions on the natural numbers are exactly those generated by primitive recursion and (pure) “minimalization” – or the μ -operator.*

We shall now develop these ideas briefly.

5.1 Primitive recursion

The primitive recursive functions are (total) functions $f : \mathbb{N}^n \rightarrow \mathbb{N}$ (this include *constants* such as $\text{zero} : \mathbb{N}^0 = 1 \rightarrow \mathbb{N}$) which are generated by:

Basic functions: The following functions are primitive recursive:

- (a) $\text{zero} : 1 \rightarrow \mathbb{N}$
- (b) $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$
- (c) All projections $\pi_i^n : \mathbb{N}^n \rightarrow \mathbb{N}; (x_1, \dots, x_n \mapsto x_i$ (here $1 \leq i \leq n$).

Composition If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ and $(g_i : \mathbb{N}^m \rightarrow \mathbb{N})_{i=1, \dots, n}$ are primitive recursive functions then their composite h is primitive recursive:

$$h : \mathbb{N}^m \rightarrow \mathbb{N}; (x_1, \dots, x_m) \mapsto f(g_1(x_1, \dots, x_m), \dots, g_n(x_1, \dots, x_m))$$

Primitive recursion: If $g : \mathbb{N}^n \rightarrow \mathbb{N}$ and $h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ are primitive recursive functions then $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ defined by:

$$\begin{aligned} f(\text{zero}, x_1, \dots, x_n) &= g(x_1, \dots, x_n) \\ f(\text{succ}(n), x_1, \dots, x_n) &= h(f(n, x_1, \dots, x_n), n, x_1, \dots, x_n) \end{aligned}$$

is primitive recursive.

All the basic arithmetic functions (addition, multiplication, exponentiation, predecessor, monus, ..) are all primitive recursive. Indeed primitive recursion already contains so much power that any reasonable (terminating) program will be primitive recursive. In particular, all the complexity classes such as polynomial time, polynomial space, exponential-time are all within primitive recursion.

We can quite easily simulate primitive recursion in the λ -calculus. At this stage all except for the last requirement, that is actually producing a new function using primitive recursion, are clearly satisfied by the λ -calculus. Here is how we can produce the function f defined by primitive recursion using the fold:

$$f(n, x_1, \dots, x_n) = \pi_1^2(\text{foldNat}\langle g(x_1, \dots, x_n), 0 \rangle(\lambda z. \langle h(\pi_1^2 z, \pi_2^2 z, x_1, \dots, x_n), \pi_2^2 z + 1 \rangle)))$$

Notice the use of pairs in this definition.

Primitive recursive functions are along way from representing all the computable (total) functions. This can be demonstrated by the following sketched diagonal argument.

Consider all the primitive recursive functions $\mathbb{N} \rightarrow \mathbb{N}$: they can certainly be enumerated (here one must resort to some sort of Gödel numbering to establish this but I am content here to assume it is so):

$$f_0, f_1, f_2, f_3, \dots$$

We necessarily have to allow repetitions in this list but we can guarantee that every primitive recursive function will appear. Now define the function $g : \mathbb{N} \rightarrow \mathbb{N}; i \mapsto f_i(i) + 1$ then g is not in the collection as for all i we have $g(i) = f_i(i) + 1 \neq f_i(i)$. In this way, so long as the enumeration was computable, we can build a new computable function from our old collection. This sketched argument shows that it is not possible to generate an enumeration of all the *total* computable functions. By contrast, and somewhat surprisingly – considering the total functions are a subset, it *is* possible to enumerate all the partial computable functions.

The diagonal argument can be used in a number of slightly more sophisticated ways: for example if we wish to build a function which eventually grows faster than any function in our enumerable collection we can use the diagonal argument to do so. By “eventually grows faster” we mean a function g such that for each f in the enumerable collection there is an $m \in \mathbb{N}$ such that for all $m' > m$ $g(m') > f(m')$. We define g by:

$$g(m) = \max\{f_i(m) \mid i \leq m\} + 1$$

the function g thus defined has $g(m) > f_n(m)$ for all $n < m$. Clearly, to build such fast growing functions we need not enumerate *all* the functions in the set but merely a subset which dominates any given the functions in the set: this allows a more economical description and underlay the development of Ackermann’s function.

At any rate we now know something is definitely missing!

5.2 Minimalization

To capture all computable functions Kleene introduced the following minimalization operator over the class of all primitive recursive functions:

$$\frac{g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}}{\mu g : \mathbb{N}^n \rightarrow \mathbb{N}}$$

where μg is, in general, a partial function defined by:

$$\mu g(x_1, \dots, x_n) = \min\{n \mid g(n, x_1, \dots, x_n) = 0\}$$

Note this is, in general partial, because g may never hit zero.

We shall introduce two “small” modifications to Kleene’s minimalization (the second not so small!). First, sometimes it is not the first argument over which we wish to minimalize but rather some other argument. In this case we write:

$$\mu n.g(x_1, n, x_2, \dots, x_n) = \min\{n \mid g(x_1, n, x_2, \dots, x_n) = 0\}$$

Second, sometimes we do not want to return precisely the first argument but rather a (primitive recursive) function of that argument. Thus we shall therefore allow the output of a μ -function to be modified by a (primitive recursive) decreasing function h (thus, $h(n, x_1, \dots, x_n) \leq n$). We shall write this modified minimalization by:

$$\mu n.h(n, x_1, \dots, x_n) \mid g(n, x_1, \dots, x_n) = h(\min\{n \mid g(n, x_1, \dots, x_n) = 0\}, x_1, \dots, x_n)$$

We shall refer to $\mu n.g(n, x)$ as *pure* minimalization.

It is worth mentioning that one can also define $\mu n.g(n, x)$ for a *partial* function g . However, now one must be careful: when one finds a minimal N such that $g(N, x) = 0$, each $g(n, x)$ with $n \leq N$ must be defined (and non-zero). One way to say this is by computing $\min_{i=1, \dots, N} g(i, x)$ and asking this to be equal to zero. In this way if any of the smaller calculations does not terminate then the whole will not terminate.

Let us make some simple observations on these operators:

1. Let $f : \mathbb{N}^k \rightarrow \mathbb{N}$ be any primitive recursive function then set

$$g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}; (n, x_1, \dots, x_k) \mapsto (f(x_1, \dots, x_k) \dot{-} n) + (n \dot{-} f(x_1, \dots, x_k))$$

then $\mu n.g(n, x_1, \dots, x_k) = f(x_1, \dots, x_k)$. This means pure minimalization can immediately generate all the primitive recursive functions which one has so far ...

2. Now something harder: consider a composite partial function

$$\mu n.h(n, x) \mid g(n, \mu m_1.h_1(m_1, x)) \mid f_1(m_1, x), \dots, \mu m_k.h_k(m_k, x) \mid f_k(m_k, x)$$

where we have, to simplify notation, represented the parameters by a single variable, x . We note that we can obtain this in one step by employing a special enumeration of \mathbb{N}^{k+1} which we shall write as $e^p : \mathbb{N} \rightarrow \mathbb{N}^p$ and whenever $e^p(m) = (r_1, \dots, r_p)$ we will write $e_i^p(m) = r_i$. This enumeration must satisfy the following condition: when $e^p(m) = (r_1, \dots, r_p)$ then, whenever $r'_i \leq r_i$ then there is an $m' \leq m$ with $e_p(m') = (r'_1, \dots, r'_p)$. This does not determine the enumeration and, in fact, there are a number of such enumerations which are primitive recursive.

Given such an enumeration now one can define

$$\begin{aligned} q(r, x) &= g(e_1^{k+1}(r), h_1(e_2^{k+1}(r), x), \dots, h_k(e_{k+1}^{k+1}(r), x)) \\ &\quad + \sum_{j=1}^k f_j(e_j^{k+1}(r), x) \end{aligned}$$

and it is easily checked that

$$\begin{aligned} \mu n.h(e_1^k(n), x) \mid q(n, x) \\ = \mu n.h(n, x) \mid g(n, \mu m_1.h_1(m_1, x)) \mid f_1(m_1, x), \dots, \mu m_k.h_k(m_k, x) \mid f_k(m_k, x)). \end{aligned}$$

3. We now want to show that one can obtain functions defined by primitive recursion over minimalized partial functions using (extended) minimalization. To do this we shall use the fact that there is one primitive recursive function which combines the behavior of all the

above enumeration operators, $e(k, i, m) = e_i^k(m)$, whose behavior when $i > n$ will not matter (one may suppose it returns zero).

Now suppose we have

$$\begin{aligned} f(0, x) &= \mu n. h(n, x) \mid g(n, x) \\ f(n+1, x) &= \mu m. h'(m, x) \mid q(m, f(n, x), n, x) \end{aligned}$$

then we define

$$\begin{aligned} f'(m, 0, p, x) &= g(e_1^p(m), x) \\ f'(m, n+1, p, x) &= q(e_{n+1}^p(m), h''(e_n^p(m), n, x), n, x) \\ &\quad + f'(e_n^p(m), n, p, x) \end{aligned}$$

where $h''(m, 0, x) = h(m, x)$ and $h''(m, n+1, x) = h'(m, x)$. Then we observe $\mu m. h''(m, n, x) \mid f'(m, n, n, x_1, \dots, x_k) = f(n, x_1, \dots, x_k)$

Essentially we have shown that the computable partial functions as expressed by (extended) minimalization can be combined by composition and primitive recursion: this should not be surprising!

Minimalization is often characterized as the ability to do “unbounded search”. This is because we can implement the μ -operator by

$$\begin{aligned} \mu n. h(n, x) \mid g(n, x) &= \text{if } g(0, x) == 0 \text{ then } h(0, x) \\ &\quad \text{elseif } g(1, x) == 0 \text{ then } h(1, x) \\ &\quad \text{elseif } g(2, x) == 0 \text{ then } h(2, x) \\ &\quad \dots \end{aligned}$$

where we simply search the numerals in ascending order for a zero of the function. This, of course can be implemented using general recursion as:

$$\begin{aligned} f(x) &= h(0, x) \quad \text{where} \\ h(n, x) &= \text{if } g(n, x) \text{ then } n \text{ else } h(n+1, x) \end{aligned}$$

Because general recursion can be implemented in the λ -calculus, as discussed above, we can certainly implement this function! This allows us to conclude that all computable functions can be written in the λ -calculus.

An alternative way to show that all computable functions can be programmed in the λ -calculus is to directly program a Turing machine. The fact that we have lists allows us to do this in a quite straightforward manner: thus we have two ways of verifying that all computable functions are present.

6 Undecidability of the λ -calculus

The purpose of this section is to prove what is sometimes known as the Scott-Curry undecidability theorem for the λ -calculus: it is an analogue of Rice’s theorem for partial recursive functions. It has the consequence that every non-trivial computable (or recursive) property P on λ -terms which is closed under equality is recursively undecidable.

A property on the terms of the λ -calculus is **non-trivial** in case there are terms A and B such that $P(A) = \text{True}$ and $P(B) = \text{False}$. Being **closed to equality** means that if $P(N)$ is true and $N =_{\beta} M$ then $P(M)$ must also hold. Clearly for this to make sense P must be a predicate on terms (that is the syntax) rather than the “meaning” of those terms: being closed to equality means it is also a predicate on the meaning.

Examples of a properties which are close to equality and which we know are non-trivial include:

1. That N is equal to any fixed term M . This is obviously closed to equality but it is also non-trivial as the property is true of M but cannot be true of both True and False . If there was no such predicate this would mean that equality of λ -terms is undecidable.
2. N has a normal form. We know Ω does not have a normal form and True being in normal form certainly has a normal form, thus, the property is non-trivial. The confluence of β -reduction, tells us that if two terms are equal and one has a normal form then the other must so it is closed to equality. Having a normal form is equivalent to asking whether a program (evaluated by name) terminates.
3. N has a head normal form, that is it is “solvable”. This is asking much less: namely that the computation the term represents will produce *some* information. It is non-trivial for the reasons given for the normal form. To show it is closed with respect to equality note that any term equal to a term in head normal form can be reduced to a term to which the term in head normal form can be reduced. But this term is then certainly in head normal form.
4. The negation of any non-trivial property which is closed to equality is automatically non-trivial and closed to equality. Thus, for example we may observe that being insolvable is undecidable as well (as it is the negation of having a head normal form).

The first example may seem rather counter-intuitive as surely, if one chooses a term M in normal form, say True , one can simply reduce the other term to normal form! However, a little thought about this will make one realize that there is a serious flaw: if the term is not equal to the other term it may not even have a normal form so that it is not clear when one should abandon the attempt to reduce the term to a normal form. If one abandons the attempt too soon you may actually miss that it does have a normal form which, worse, may actually equal True .

This also has the consequence that there can be no computable total function which determines a number corresponding to terms such that, if the term could be reduced to normal form, the process could be completed in less steps than that number. Clearly access to such a number would allow one to decide the equality and indeed whether there was a normal form. Thus, no such number can be computed. However, there is definitely a way to associate such numbers with terms: simply do the reductions and count! Here is the point: this is a partial function.

We shall state the theorem in its positive form:

Theorem 6.1 (Scott-Curry) *Given any recursive predicate on the terms of the λ -calculus for which there are terms A and B with $\psi(A) = \neg\psi(B)$ then there is a term X such that $A' =_{\beta} X =_{\beta} B'$ and $P(A') = \neg P(B')$.*

For the proof we represent the λ -terms internally in the λ -calculus by representing the datatype of λ -terms. We shall write this term as \underline{N} and it is clear that they have an application (recall we are just translating syntax) $\underline{NM} = \underline{N} \bullet \underline{M}$. It is clear that we can define a function T in the λ -calculus such that $\underline{N} = T\underline{N}$.

If the predicate is recursive we may represent it as a λ -term P which normalizes on the terms of the datatype and produces either `True` or `False` depending on whether ψ is satisfied or not. We then have, without loss of generality, $PA = \text{True}$ and $PB = \text{False}$.

Lemma 6.2 (Second recursion theorem) *Given any F then there is a λ -term X such that $X =_{\beta} F\underline{X}$*

PROOF: This allows us to define:

$$H := \lambda x.F(x \bullet (Tx))$$

so that

$$H\underline{H} =_{\beta} F(\underline{H} \bullet (T\underline{H})) =_{\beta} F(\underline{H} \bullet (\underline{H})) =_{\beta} F(\underline{H}\underline{H})$$

and so we may set $X := H\underline{H}$. □

PROOF: (of the theorem) To complete the proof set

$$F := \lambda x.\text{if } Px \text{ then } B \text{ else } A$$

and X to be such that $X = F\underline{X}$ then

$P\underline{X} = \text{True}$: We get $X = F\underline{X} = B$ so that $A' = X$ and $B' = B$.

$P\underline{X} = \text{False}$: We get $X = F\underline{X} = A$ so that $A' = A$ and $B' = X$. □

Notice that the theorem says something even stronger:

Corollary 6.3 *Given any two disjoint sets of terms which are non-empty and closed to β -equality then there is no recursive predicate which separates the sets.*

7 Combinatory algebra

One may think that the λ -calculus is a remarkably simple system for expressing computability but it turns out there is an even simpler system!! This is often called “combinatory logic” but it is a very simple algebraic system: it has a binary operation, called application, and two constants \mathbf{k} and \mathbf{s} . These are subject to the following equations which we orient as rewrites:

$$\begin{aligned} (\mathbf{k} \bullet x) \bullet y &\rightarrow x \\ ((\mathbf{s} \bullet x) \bullet y) \bullet z &\rightarrow ((x \bullet z) \bullet (y \bullet z)) \end{aligned}$$

As we will see these rewrite rules form an orthogonal left-linear rewriting system. Therefore, this is a confluent system in which standard reductions are guaranteed to find normal forms. It is also a non-trivial system as $\mathbf{s} \bullet \mathbf{k} \neq \mathbf{k}$. The system, in fact, satisfies some quite remarkable properties which we now briefly explore.

An **applicative system** is a set A with a single binary operation

$$_ \bullet _ : A \times A \rightarrow A; (x, y) \mapsto x \bullet y.$$

Clearly a combinatory algebra is an example of a applicative system as one can simply forget about \mathbf{k} and \mathbf{s} . An applicative system is said to be functionally complete if whenever there is a polynomial expression $p(x_1, \dots, x_n)$ in n variable

$$p \rightarrow x_1|x_2|\dots|x_n|a \in A|p \bullet p$$

then there is an element \widehat{p} such that for every substitution of the variables by elements of A

$$(\dots((\widehat{p} \bullet x_1) \bullet \dots) \bullet x_n = p(x_1, \dots, x_n))$$

Proposition 7.1 *Combinatory algebra is functionally complete. Furthermore, a combinatory complete applicative system is a combinatory algebra*

PROOF: The proof involves the introduction of an abstraction mechanism which is a mechanism for building constants with the desired property for functional completeness. Suppose, therefore, we have a polynomial p which involves free variables $X = x_1, \dots, x_n$ and then define $\lambda^*x_i.p$ as the following polynomial in which the variable x_i has been removed:

$$\begin{aligned} \lambda^*x.x &= (\mathbf{s} \bullet \mathbf{k}) \bullet \mathbf{k} \\ \lambda^*x.z &= \mathbf{k} \bullet z \quad z \text{ is } \mathbf{s} \text{ or } \mathbf{k} \text{ or a variable} \\ \lambda^*x.(p_1 \bullet p_2) &= (\mathbf{s} \bullet (\lambda^*x.p_1)) \bullet (\lambda^*x.p_2) \end{aligned}$$

Observe that $(\lambda^*x.p) \bullet M = p[M/x]$ which can be seen by a structural induction:

$$\begin{aligned} (\lambda^*x.x) \bullet M &= ((\mathbf{s} \bullet \mathbf{k}) \bullet \mathbf{k}) \bullet M \\ &= (\mathbf{k}) \bullet M \bullet (\mathbf{k}) \bullet M = M = x[M/x] \\ (\lambda^*x.z) \bullet M &= (\mathbf{k} \bullet z) \bullet M = z = z[M/x] \\ \lambda^*x.(p_1 \bullet p_2) \bullet M &= ((\mathbf{s} \bullet (\lambda^*x.p_1)) \bullet (\lambda^*x.p_2)) \bullet M \\ &= ((\lambda^*x.p_1) \bullet M) \bullet ((\lambda^*x.p_2) \bullet M) \\ &= p_1[M/x] \bullet p_2[M/x] = (p_1 \bullet p_2)[M/x] \end{aligned}$$

Also we observe that $(\lambda^*x.p)[M/y] = (\lambda^*x.p[M/y])$ when $x \notin FV(p)$ again a proof by structural induction is required which we leave to the reader). This now shows that we can take

$$\hat{p} := (\lambda^*x_1.(\lambda^*x_2\dots(\lambda^*x_n.p)\dots))$$

to obtain:

$$\begin{aligned} & (((\lambda^*x_1.(\lambda^*x_2\dots(\lambda^*x_n.p)\dots)) \bullet M_1) \bullet M_2) \bullet \dots \bullet M_n \\ &= (((\lambda^*x_2.(\dots(\lambda^*x_n.p)\dots)[M_1/x_1]) \bullet M_2) \bullet \dots) \bullet M_n \\ &= ((\lambda^*x_2.(\dots(\lambda^*x_n.p[M_1/x_1])\dots)) \bullet M_2) \bullet \dots \bullet M_n \\ &= p[M_1/x_1, \dots, M_n/x_n] \end{aligned}$$

For the converse it suffices to show that a combinatory algebra which is functionally complete has a k and an s . However, the equations for these combinators are functional completeness equations so this must be so! \square

These observations suggest that in combinatory algebra, following the λ -calculus

$$\Omega = (\lambda^*x.x \bullet x) \bullet (\lambda^*x.x \bullet x)$$

should have a non-terminating reduction. This is easily checked to be the case. However, it should not be imagined that the two systems are equivalent: in combinatory algebra:

$$N = M \not\equiv \lambda^*x.M = \lambda^*x.N$$

Thus $(k \bullet x) \bullet y = x$ but

$$\begin{aligned} \lambda^*x.(k \bullet x) \bullet y &= (s \bullet (\lambda^*x.k \bullet x)) \bullet (\lambda^*x.y) \\ &= (s \bullet ((s \bullet (\lambda^*x.k)) \bullet (\lambda^*x.x))) \bullet (\lambda^*x.y) \\ &= (s \bullet ((s \bullet (k \bullet k)) \bullet ((s \bullet k) \bullet k))) \bullet (k \bullet y) \\ &\neq (s \bullet k) \bullet k = \lambda^*x.x \end{aligned}$$

So that combinatory logic is much weaker. Combinatory logic is important as a system as it one of the simplest systems in which all (partial) computable functions can be simulated. The encoding technique follows the techniques of the λ -calculus.

References

- [1] *The λ -calculus: its syntax and semantics*. H. Barendregt, North Holland (1984)
- [2] *Introduction to Combinators and λ -calculus*. J.R. Hindley and J.P. Seldin, London Mathematical Society (1988)
- [3] *Lambda Calculus and Combinators: an introduction*. J.R. Hindley and J.P. Seldin, Cambridge University Press (2008)
- [4] *Term rewriting systems*. J.W. Klop in "Handbook of logic for Computer Science" (Vol. 2) Clarendon Press (1992)