Notes on the λ-calculus

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

November 30, 2017

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. We then continue to study the typed λ-calculus and its properties leading to Gödel's system T, Peano arithmetic, and the higher-order primitive recursive functions.

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 by 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, $\lambda x.M$ here x is being bound: this gives the calculus its name.

More formally the terms in the calculus are formed using the following rules:

$$\boxed{\begin{array}{c} \dfrac{x \ \ \mathsf{Variable}}{x \ \ \mathsf{Term}}\ \mathrm{Var} \\[2em] \dfrac{M \ \ \mathsf{Term} \quad N \ \ \mathsf{Term}}{MN \ \ \mathsf{Term}}\ \mathrm{app} \qquad \dfrac{M \ \ \mathsf{Term} \quad x \ \ \mathsf{Variable}}{\lambda x.M \ \ \mathsf{Term}}\ \mathrm{abst} \end{array}}$$

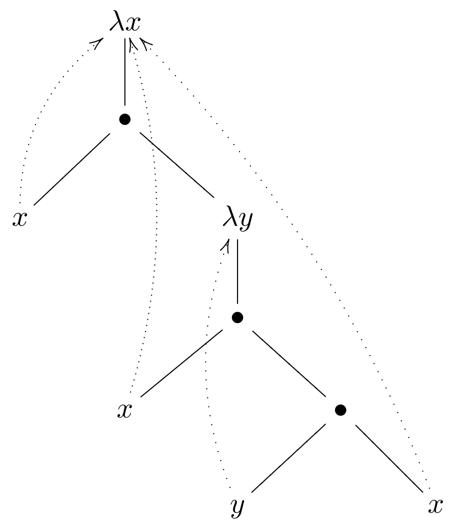
λ-term construction.

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 on 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 $\lambda.0(\lambda.1(01))$. This way of indexing the variables is sometimes known as a "De Bruijn indexing" after the Dutch mathematician, Nicholaas 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 (i.e. set of variables) in which we build can build the term. Below are the inference rules for building terms in a variable context. A term t can be built from a variable context V, written $V \vdash t \ \ \mathsf{Term}$, if and only if it can be built using the rules of the following system:

$$\frac{x \ \ \mathsf{Variable}}{\{x\} \vdash x \ \ \mathsf{Term}}\ \text{Var}$$

$$\frac{V_1 \vdash M \ \ \mathsf{Term} \quad V_2 \vdash N \ \ \mathsf{Term}}{V_1 \cup V_2 \vdash MN \ \ \mathsf{Term}}\ \text{app} \qquad \frac{V \vdash M \ \ \mathsf{Term} \quad x \ \ \mathsf{Variable}}{V - \{x\} \vdash \lambda x.M \ \ \mathsf{Term}}\ \text{abst}$$

$$\frac{W \vdash M \ \ \mathsf{Term} \quad W \subseteq V}{V \vdash M \ \ \mathsf{Term}}\ \text{Weaken}$$

Variable contexts and λ-term construction.

Two λ terms are **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:

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

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:

$$\frac{V \vdash M \quad \mathsf{Term}}{V \vdash M = M}\ \text{Reflexive}$$

$$\frac{V \vdash N = M}{V \vdash M = M}\ \text{Symmetry} \qquad \frac{V \vdash M = N \quad V \vdash N = P}{V \vdash M = P}\ \text{Transitivity}$$

$$\frac{V \vdash (\lambda x.M)N \quad \mathsf{Term}}{V \vdash (\lambda x.M)N = M[N/x]}\ \beta\text{-equality} \qquad \frac{V, x \vdash M = N}{V \vdash (\lambda x.M) = (\lambda x.N)}\ \text{abst}$$

$$\frac{V \vdash M = M'}{V \vdash MN = M'N}\ \text{Lapp} \qquad \frac{V \vdash N = N'}{V \vdash MN = MN'}\ \text{Rapp}$$

β-equality between λ-terms.

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:

$$\boxed{\begin{array}{c}
\dfrac{V \vdash (\lambda x.M)N \ \ \mathsf{Term}}{V \vdash (\lambda x.M)N \longrightarrow M[N/x]}\ \beta\text{-reduction} \\[2em]
\dfrac{V, x \vdash M \longrightarrow N}{V \vdash (\lambda x.M) \longrightarrow (\lambda x.N)}\ \text{abst} \\[2em]
\dfrac{V \vdash M \longrightarrow M'}{V \vdash MN \longrightarrow M'N}\ \text{Lapp} \qquad \dfrac{V \vdash N \longrightarrow N'}{V \vdash MN \longrightarrow MN'}\ \text{Rapp}
\end{array}}$$

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

$$N \xrightarrow{*} M := (N = M_0) \longrightarrow M_1 \longrightarrow ... \longrightarrow (M_n = M)$$

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 with a single hole where one can put a λ expression. This is written $C[\![\_]\!]$ when a term N is put at this position we write $C[\![N]\!]$. A context, in fact, also makes available some variables to the term which is to fill the hole. We shall indicate the variables which a context makes available in its hole by subscripting with a variable set: $C_\Gamma[\![\_]\!]$. We may then write the rules for building a context:

The first rule creates the hole immediately at the root: thus there is no term surrounding the hole in this case and, of course, only the free variable are available at this position. In the case of the λ-abstraction rule notice that the outside context has strictly fewer variables as the hole has the bound variable x available to it.

To fill the hole of a context one needs a term which can be built using the variables which are available in the hole, one can then fill the hole. For a filled context we shall drop the variable subscription:

$$\frac{\Gamma \vdash C_{\Gamma'}[\![\_]\!]\ \mathsf{context} \quad \Gamma' \vdash M\ \mathsf{term}}{\Gamma \vdash C[\![M]\!]\ \mathsf{term}}$$

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

$$\frac{}{\Gamma \vdash \_ = \mathsf{Id}_\Gamma[\![\_]\!] \text{ context}}\ \text{Empty}$$

$$\frac{\Gamma, x \vdash C_{\Gamma'}[\![\_]\!] \text{ context}}{\Gamma \vdash \lambda x.C_{\Gamma',x}[\![\_]\!] \text{ context}}\ \text{abst}$$

$$\frac{\Gamma \vdash M \text{ term} \quad \Gamma \vdash C_{\Gamma'}[\![\_]\!] \text{ context}}{\Gamma \vdash M\ C_{\Gamma'}[\![\_]\!] \ \text{ context}}\ \text{Lapp} \qquad \frac{C_{\Gamma'}[\![\_]\!] \text{ context} \quad N \text{ term}}{C_{\Gamma'}[\![\_]\!]\ N \ \text{ context}}\ \text{Rapp}$$

Table 1: Construction of contexts

notation of contexts allows a one step rewrite to be viewed as a top-level β-reduction in context:

$$C[\![(\lambda x.M)N]\!] \xrightarrow[\beta]{} C[\![M[N/x]]\!].$$

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*

$$\begin{tikzcd} M \arrow[dr, "*"'] & & N \arrow[dl, "*"] \\ & L & \end{tikzcd}$$

2.1 λ-terms in β-normal form

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 $\mathsf{True} := \lambda xy.x$ and $\mathsf{False} := \lambda xy.y$;

(b) Terms which contain no λ-abstraction are always in normal form:

$$x, xy, xx, yy, xyxx, ...$$

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

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

(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.

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

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

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

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

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

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

- $(\lambda x.x)(\lambda x.x) \longrightarrow (\lambda x.x)$ is a reduction to normal form.
- $\Omega := (\lambda x.xx)(\lambda x.xx) \longrightarrow xx[(\lambda x.xx)/x] = (\lambda x.xx)(\lambda x.xx)$ 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:

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

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. □

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 led 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:

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

(This is immediate!)

OBSERVATION II:

If we can prove:

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

$$\begin{tikzcd}
 & N \arrow[dl, "*"] \arrow[dr, "*"'] & \\
M_1 & & M_2
\end{tikzcd}$$

there is a convergence

$$\begin{tikzcd}
M_1 \arrow[dr, "*"'] & & M_2 \arrow[dl, "*"] \\
 & L &
\end{tikzcd}$$

Then the Church Rosser theorem holds!

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

$$\begin{tikzcd}
 & N_1 \arrow[dl, "8"] \arrow[dr, "*"] & & N_3 \arrow[dl, "*"'] \arrow[dr, "*"] & & N_5 \arrow[dl, "*"'] \arrow[dr, "*"] & \\
N_0 \arrow[dr, "*"'] & & N_2 \arrow[dl, "*"] & & N_4 \arrow[ddll, "*"] & & \ldots \\
 & M_1 \arrow[dr, "*"'] & & & & & \\
 & & M_2 \arrow[dr, "*"] & & & & \\
 & & & \ldots & & &
\end{tikzcd}$$

OBSERVATION III:

If we can prove:

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

$$\begin{tikzcd}
 & N \arrow[dl] \arrow[dr, "*"] & \\
M' & & M
\end{tikzcd}$$

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

$$\begin{tikzcd}
M' \arrow[dr, "*"'] & & M \arrow[dl, "*"] \\
 & L &
\end{tikzcd}$$

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

$$\begin{tikzcd}
N \arrow[r, "*"] \arrow[d] & M \arrow[d, "*"] \\
N_1 \arrow[r, "*"] \arrow[d] & M_1 \arrow[d, "*"] \\
N_2 \arrow[r, "*"] \arrow[d] & M_2 \arrow[d, "*"] \\
N_3 \arrow[d, dotted, "*"'] & \ldots. \quad M_3 \arrow[d, dotted, "*"] \\
M' \arrow[r, "*"] & M_n
\end{tikzcd}$$

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:

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

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 $N \rightsquigarrow V(N)$:

$$\boxed{\begin{array}{cc} \dfrac{x \ \mathsf{var}}{x \rightsquigarrow x} & \dfrac{M \rightsquigarrow M' \quad N \rightsquigarrow N'}{MN \rightsquigarrow M'N'} \\[2ex] \dfrac{N \rightsquigarrow N'}{\lambda x.N \rightsquigarrow \lambda x.N'} & \dfrac{N \rightsquigarrow N' \quad M \rightsquigarrow M'}{(\underline{\lambda} x.N)M \rightsquigarrow N'[M'/x]} \end{array}}$$

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;

$$\begin{tikzcd} M \arrow[d, mapsto] \arrow[dr, mapsto] & \\ U(M) \arrow[r, "*"'] & V(M) \end{tikzcd}$$

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

$$\begin{tikzcd} N \arrow[r, "*"] \arrow[d] & M \\ M' & \end{tikzcd} \qquad \begin{tikzcd} U(N') = N \arrow[r, "*"] \arrow[d] & U(M) \\ V(N') = M' & \end{tikzcd}$$

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:*

$$\begin{tikzcd} M \arrow[r] \arrow[d, "*"'] & M' \arrow[d, "*"] \\ V(M) \arrow[r, dotted, "*"'] & V(M') \end{tikzcd}$$

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

$$\begin{tikzcd} M \arrow[r] \arrow[d, "*"] & M_1 \arrow[r] \arrow[d, "*"] & M_2 \arrow[r] \arrow[d, "*"] & \ldots \arrow[r] & M' \arrow[d, "*"] \\ M \arrow[r, dotted] & M_1 \arrow[r, dotted] & M_2 \arrow[r, dotted] & \ldots \arrow[r, dotted] & M' \end{tikzcd}$$

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):

$$\begin{tikzcd}
(\lambda x.M)N \arrow[r] \arrow[d, "*"'] & M[N/x] \arrow[d, "*"] \\
(\lambda x.V(M))V(N) \arrow[r] & V(M)[V(N)/x] = V(M[N/x])
\end{tikzcd}$$

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

At the root(marked):

$$\begin{tikzcd}
(\underline{\lambda} x.M)N \arrow[r] \arrow[d, "*"'] & M[N/x] \arrow[dd, "*"] \\
(\underline{\lambda} x.V(M))V(N) \arrow[d] & \\
V(M)[V(N)/x] \arrow[r, equal] & V(M)[V(N)/x]
\end{tikzcd}$$

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

$$\begin{tikzcd}
C\llbracket(\lambda x.M)N\rrbracket \arrow[r] \arrow[d, "*"'] & C\llbracket M[N/x]\rrbracket \arrow[d, "*"] \\
C\llbracket(\lambda x.V(M))V(N)\rrbracket \arrow[r] \arrow[d, "*"'] & C\llbracket V(M)[V(N)/x]\rrbracket \arrow[d, "*"] \\
V(C\llbracket(\lambda x.V(M))V(N)\rrbracket) \arrow[r, "*"'] & V(C\llbracket V(M)[V(N)/x]\rrbracket)
\end{tikzcd}$$

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 the establish that the lower square can be completed: this done in lemma 2.9.

In context (marked):

$$\begin{tikzcd}
C\llbracket(\underline{\lambda} x.M)N\rrbracket \arrow[r] \arrow[d, "*"'] & C\llbracket M[N/x]\rrbracket \arrow[d, "*"] \\
C\llbracket V(M)[V(N)/x]\rrbracket \arrow[r, equal] \arrow[d, "*"'] & C\llbracket V(M)[V(N)/x]\rrbracket \arrow[d, "*"] \\
V(C\llbracket V(M)[V(N)/x]\rrbracket) \arrow[r, equal] & V(C\llbracket V(M)[V(N)/x]\rrbracket)
\end{tikzcd}$$

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

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

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

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

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

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

□

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) *$V(C[\![M]\!]) = V(C[\![V(M)]\!])$;*

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

$$\begin{tikzcd}
C[\![M]\!] \arrow[r] \arrow[d, "*"'] & C[\![N]\!] \arrow[d, "*"] \\
V(C[\![M]\!]) \arrow[r, "*"'] & V(C[\![N]\!])
\end{tikzcd}$$

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):

$$\begin{array}{rcll}
V(NC[\![M]\!]) & = & V(N)V(C[\![M]\!]) & \text{defn of marked evaluation} \\
& = & V(N)V(C[\![V(M)]\!]) & \text{structural induction} \\
& = & V(NV(C[\![V(M)]\!]) & \text{defn of marked evaluation}
\end{array}$$

For unmarked abstractions we have:

$$\begin{array}{rcll}
V(\lambda x.C[\![M]\!]) & = & \lambda x.V(C[\![M]\!]) & \text{defn of marked evaluation} \\
& = & \lambda x.V(C[\![V(M)]\!]) & \text{structural induction} \\
& = & V(\lambda x.C[\![V(M)]\!]) & \text{defn of marked evaluation}
\end{array}$$

Finally for the first case of marked abstraction we have:

$$\begin{array}{rcll}
V((\underline{\lambda} y.C[\![M]\!])N) & = & V(C[\![M]\!])[V(N)/y] & \text{defn of marked evaluation} \\
& = & V(C[\![V(M)]\!])[V(N)/y] & \text{structural induction} \\
& = & V((\underline{\lambda} y.C[\![V(M)]\!])N) & \text{defn of marked evaluation}
\end{array}$$

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

$$V(LC[\![M]\!]) \rightarrow V(LC[\![M]\!])$$

on the assumption that we have a sequence of rewrites $V(C[\![M]\!]) \longrightarrow V(C[\![M]\!])$. But as $V(LC[\![M]\!]) = V(L)V(C[\![M]\!])$ and $V(LC[\![M]\!]) = V(L)V(C[\![M]\!])$ 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:

$$\begin{tikzcd}
(\underline{\lambda} y.C[\![M]\!])M' \arrow[r] \arrow[d, "*"'] & (\underline{\lambda} y.C[\![N]\!])M' \arrow[d, "*"] \\
V(C[\![M]\!][M'/y]) \arrow[d, equal] & V(C[\![N]\!])[M'/y]) \arrow[d, equal] \\
V(C[\![M]\!])[V(M')/y] \arrow[r, "*"'] & V(C[\![N]\!])[V(M')/y]
\end{tikzcd}$$

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):

$$\begin{tikzcd}
(\underline{\lambda} y.M')C[\![M]\!] \arrow[r] \arrow[d, "*"'] & (\underline{\lambda} y.M')C[\![N]\!] \arrow[d, "*"] \\
V(M'[C[\![M]\!]/y]) \arrow[d, equal] & V(M'[C[\![N]\!]/y]) \arrow[d, equal] \\
V(M')[V(C[\![M]\!])/y] \arrow[r, "*"'] & V(M')[V(C[\![N]\!])/y]
\end{tikzcd}$$

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:

$$\begin{array}{rcl}
\mathsf{True} & := & \lambda xy.x \\
\mathsf{False} & := & \lambda xy.y \\
\mathsf{If}\ ztf & = & ztf \quad \Leftrightarrow \quad \mathsf{If} = \lambda ztf.ztf
\end{array}$$

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:

$$\begin{array}{rcl}
\mathsf{Not}\ b & = & \mathsf{If}\ b\ \mathsf{False}\ \mathsf{True} \\
\mathsf{And}\ b_1 b_2 & = & \mathsf{If}\ b_1(\mathsf{If}\ b_2\ \mathsf{True}\ \mathsf{False})\ \mathsf{False} \\
\mathsf{Or}\ b_1 b_2 & = & \mathsf{If}\ b_1\ \mathsf{True}\ (\mathsf{If}\ b_2\ \mathsf{True}\ \mathsf{False})
\end{array}$$

3.2 Products

Here we must encode the ability to form pairs:

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

Note that when we project a pair we have

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

Notice that with pairing and Boolean operations one can represent all finite stes (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:

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

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:

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

This allows us to define addition and multiplication:

$$\begin{array}{rcl}\mathsf{Add}\ n\ m & = & \mathsf{Fold}\ m\ \mathsf{Succ}\ n \\ \mathsf{Mult}\ n\ m & = & \mathsf{Fold}\ \mathsf{Zero}\ (\mathsf{Add}\ m)\ n\end{array}$$

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

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

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:

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

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 where both zero (using the iszero function. However, the easiest way to define truncated subtraction was to use the predecessor function. Here are the programs:

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

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

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

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:

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

Next we show how to implement this combinator:

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

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

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

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 to 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:

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

Here are the basic functions associated with lists:

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

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

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

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

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

Then $\mathsf{CaseList}$ can be programmed as follows:

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

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:

$$\mathsf{Tail} l \ \ := \ \ \mathsf{case}\ l\ \mathsf{of} \left| \begin{array}{lcl} \mathsf{Nil} & \mapsto & \mathsf{Nil} \\ \mathsf{Cons}\ a\ as & \mapsto & as \end{array} \right.$$

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:

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

The first step is to write down some simple trees:

$$\begin{array}{rcl} \mathsf{Leaf}\ x & = & \lambda\ l\ n.l\ x \\ \mathsf{Node}\ (\mathsf{Leaf} x_1)\ (\mathsf{Leaf} x_2) & = & \lambda\ l\ n.n\ (l\ x_1)\ (l\ x_2) \\ & & \ldots. \end{array}$$

Now one defines:

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

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:

$$\mathsf{case}\ t\ \mathsf{of} \left| \begin{array}{lcl} \mathsf{Leaf} x & \mapsto & v_1 \\ \mathsf{Node}\ t_1\ t_1 & \mapsto & v_2 \end{array} \right. := \mathsf{CaseTree}\ t\ (\lambda\ x.v_1)\ (\lambda\ t_1\ t_2.v_2)$$

Next write down the $\mathsf{CaseTree}$ function using products:

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

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:

$$\mathsf{data}\ \ \mathsf{D}\ a_1\ ...\ a_n = \left|\begin{array}{l}\mathsf{Cons}_1(T_{11}\ a_1\ ...\ a_n\ (\mathsf{D}\ a_1\ ...\ a_n))\ ...\ (T_{1m_1}\ a_1\ ...\ a_n\ (\mathsf{D}\ a_1\ ...\ a_n)) \\ ... \\ \mathsf{Cons}_p(T_{p1}\ a_1\ ...\ a_n\ (\mathsf{D}\ a_1\ ...\ a_n))\ ...\ (T_{pm_p}\ a_1\ ...\ a_n\ (\mathsf{D}\ a_1\ ...\ a_n))\end{array}\right.$$

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

$$\mathsf{Cons}_i :: (T_{i1}\ a_1\ ...\ a_n\ (\mathsf{D}\ a_1\ ...\ a_n)) \to ... \to (T_{im_i}\ a_1\ ...\ a_n\ (\mathsf{D}\ a_1\ ...\ a_n)) \to (\mathsf{D}\ a_1\ ...\ a_n)$$

   .

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

$$\frac{f :: x \to y}{\mathsf{MapT}_{ij}\ f :: (T_{ij}\ a_1\ ...\ a_n\ x) \to (T_{ij}\ a_1\ ...\ a_n\ y).}$$

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

$$f_i :: (T_{i1}\ a_1\ ...\ a_n\ b) \to ... \to (T_{im_i}\ a_1\ ...\ a_n\ b) \to b$$

   This makes the type of the fold look:

$$\mathsf{foldD} :: \begin{array}{l} ((T_{11}\ a_1\ ...\ a_n\ b) \to ... \to (T_{1m_1}\ a_1\ ...\ a_n\ b) \to b) \\ \to ... \\ \to ((T_{p1}\ a_1\ ...\ a_n\ b) \to ... \to (T_{pm_p}\ a_1\ ...\ a_n\ b) \to b) \\ \to (\mathsf{D}\ a_1\ ...\ a_n) \\ \to b \end{array}$$

   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:

$$\mathsf{foldD}\ c_1\ ...\ c_p\ d = d\ c_1\ ...\ c_p.$$

   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 be able to define inductively what it means to be a element of this datatype).

5. So what do the constructors look like?

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

   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 datatype 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 datatype. Clearly, as the datatype 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 datatype:

$$\mathsf{data\ Rose}\ f\ v = \left|\begin{array}{l}\mathsf{Var}\ v\\ \mathsf{Rs}\ f\ (\mathsf{List}\ (\mathsf{Rose}\ a\ v))\end{array}\right.$$

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

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

4 General 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 $\mathsf{Y}f = f(\mathsf{Y}f)$. this can be used to encode a recursive call to a function. As suppose we define $\mathsf{g}x = M\mathsf{g}$ where M is some combinator which reuses the function g recursively then we can define $\mathsf{g} = \mathsf{Y}(\lambda gx.Mg)$ this means:

$$\begin{array}{rcl} \mathsf{g}N & := & (\mathsf{Y}(\lambda gx.Mg)) \\ & \longrightarrow & (\lambda x.M(\mathsf{Y}(\lambda gx.Mg))x)N \\ & \longrightarrow & M[N/x](\mathsf{Y}(\lambda gx.Mg)) \end{array}$$

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!):

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

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

$$\begin{array}{rcl} (Yf) & := & (Af)(Af) := (\lambda x.f(xx)(Af) \\ & \longrightarrow & f((Af)(Af)) := f(Yf) \end{array}$$

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

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

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

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

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

$$Y \longrightarrow \lambda f.f((Af)(Af)) \longrightarrow \lambda f.f(f((Af)(Af))) \longrightarrow .. \longrightarrow \lambda f.f^n((Af)(Af)) \longrightarrow ...$$

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

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

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 for general recursion

(1) Rather than define append using a fold we can implement it recursively as

$$\mathsf{append}\ x\ y := \mathsf{case}\ x\ \mathsf{of} \left| \begin{array}{lll} \mathsf{nil} & \mapsto & y \\ \mathsf{cons}\ a\ as & \mapsto & \mathsf{cons}\ a\ (\mathsf{append}\ as\ y) \end{array} \right.$$

This translates to

$$\mathsf{append} := \mathsf{Y}(\lambda\ f\ x\ y.\mathsf{case}\ x\ \mathsf{of} \left| \begin{array}{lll} \mathsf{nil} & \mapsto & y \\ \mathsf{cons}\ a\ as & \mapsto & \mathsf{cons}\ a\ (f\ as\ y) \end{array} \right. .$$

Note that the correct behaviour of this relies on the case construct which when the first argument is nil will not recall the function.

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

$$\begin{array}{rcl} \mathtt{f}n & := & (\mathsf{Y}(\lambda fn.n : (f(n+1))))n \\ & \to & (\lambda fn.n : (f(n+1)))(\mathsf{Y}(\lambda fn.n : (f(n+1))))n \\ & \to & n : ((\mathsf{Y}(\lambda fn.n : (f(n+1))))n + 1) \\ & =: & n : (\mathtt{f}n + 1) \end{array}$$

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 $\mathbb{N}^n \longrightarrow \mathbb{N}$ on the natural numbers was "computable" – in the sense that it could be calculated by machine. Thus, the first notions of computability emerged as those functions *on the natural numbers* which were computable. To determine the power of the λ-calculus people therefore considered the question of which functions on the natural numbers were "represented".

A partial function $F : \mathbb{N}^n \longrightarrow \mathbb{N}$ is represented in the λ-calculus if there is a λ-term ψ_F such that whenever $F(m_1, ..., m_n) = m_0$ then $\psi_F \underline{m_1}...\underline{m_n} =_\beta \underline{m_0}$ (where $\underline{m_i}$ is the Church numeral for m_i) and, furthermore, when $F(m_1, ..., m_n)$ is undefined, then the corresponding λ-term does not have a normal form.

Kleene developed a simple theorem which we shall use to show that all recursive functions can be represented in the λ-calculus:

Theorem 5.1 *The computable (partial) functions on the natural numbers are exactly those generated by primitive recursion and (pure) "minimization" (Kleene's μ-operator).*

We shall now develop these ideas briefly.

5.1 Primitive recursion

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

Basic functions: The following functions are primitive recursive:

(a) $\mathsf{zero} : 1 \longrightarrow \mathbb{N}$

(b) $\mathsf{succ} : \mathbb{N} \longrightarrow \mathbb{N}$

(c) All projections $\pi_i^n : \mathbb{N}^n \longrightarrow N; (x_1, ..., x_n \mapsto x_i$ (here $1 \leq i \leq n$.

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

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

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

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

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 (assuming that g and h are represented) using the fold:

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

Notice the use of pairs in this definition.

Primitive recursive and total recursive functions

Primitive recursive functions are along way from representing all the computable (total) functions. This can be demonstrated by the using Cantor's diagonal argument.

Consider all the primitive recursive functions $\mathbb{N} \longrightarrow \mathbb{N}$: they can certainly be enumerated (here one must resort to some sort of Gödel numbering to establish this: we shall – for the moment – be happy to assume that such exists):

$$f_0, f_1, f_2, f_3, ...$$

We allow repetitions in this list – in fact we have to allow repetitions – but we can guarantee that every primitive recursive function will appear. Now define the function

$$g : \mathbb{N} \longrightarrow \mathbb{N}; i \mapsto f_i(i) + 1$$

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. Somewhat surprisingly – considering the total functions are a subset – it *is* possible to enumerate all the partial computable functions. Notice that Cantor's diagonal argument cannot be applied to an enumeration of partial functions as $f_i(i)$ may not be defined for many values of i.

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 enumeration we can use the diagonal argument to do so. By "eventually grows faster" (or "majorizes") 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 $m' > m$ $g(m') > f(m')$. Using Cantor's argument we can define g by:

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

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 functions in the set: this allows a more economical description and this idea underlay the development of Ackermann's function.

It turns out just using folds one can in the λ-calculus represent far more than just primitive recursive functions: one can represent all the *higher-order* primitive recursive functions. This is a *much* larger class. In particular, one can implement Ackermann's function as we now show.

Here is a first definition of Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

However, we can re-express this in a higher-order fashion:

$$\begin{array}{rcl} \mathsf{A}\ \mathsf{Zero} & = & \mathsf{Succ} \\ \mathsf{A}\ (\mathsf{Succ}\ m) & = & \mathsf{IT}(\mathsf{A}\ m) \\ \mathsf{IT}\ f\ \mathsf{Zero} & = & f\ (\mathsf{Succ}\ \mathsf{Zero}) \\ \mathsf{IT}\ f\ (\mathsf{Succ}\ n) & = & f\ (\mathsf{IT}\ f\ n) \end{array}$$

which translates into folding on the natural numbers as:

$$A\ m\ n := (\mathsf{FoldNat}\ \mathsf{Succ}\ [\lambda f.\mathsf{FoldNat}\ (\lambda x.f\ (\mathsf{Succ}\ \mathsf{Zero}))\ (\lambda hx.f\ (h\ x))\ n]\ m\ n$$

This also show how to represent Ackermann's function using higher-order primitive recursion ...

We now know that the λ-calculus can represent all the primitive recursive functions and much more: namely, the higher-order primitive recursive functions. Furthermore, we know that there are computable total functions which are definitely not primitive recursive functions or indeed higher-order primitive recursive!

5.2 Minimization

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

$$\frac{g : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}}{\mathsf{mini}\ g : \mathbb{N}^n \longrightarrow \mathbb{N}}$$

where mini g is, in general, a partial function defined by:

$$(\mathsf{mini}\ g)x_1...x_n := \mu n.g(n, x_1, ..., x_n) := \mathsf{min}\{n | g(n, x_1, ..., x_n) = 0\}$$

Here we regard μn as binding n in $g(n, x_1, .., x_n)$.

To give a concrete feel for the operation of minimization consider a function g with values:

$$\begin{array}{rcl} g(0,4) & = & 22 \\ g(1,4) & = & 1 \\ g(2,4) & = & 14 \\ g(3,4) & = & 0 \\ g(4,4) & = & 119 \\ & ... & \end{array}$$

then $\mu n.g(n, 4) = 3$. Importantly, $x \mapsto \mu n.g(n, x)$ can be a partial map, because g may never hit zero. The use of the binder μ is a useful notation as sometimes it is not the first argument over which we wish to minimize but rather some other argument. In this case we can write:

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

and, thus, we treat μ as a binding operator which binds variables much as λ-abstraction does.

Let us make some simple observations on the minimization operator used on primitive recursive functions:

Primitive recursive functions using minimization:

First, we observe that we can re-express any primitive recursive function as a minimization. Let $f : \mathbb{N}^k \longrightarrow \mathbb{N}$ be any primitive recursive function then set

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

then $\mu n.g(n, x_1, ..., x_k) = f(x_1, ..., x_k)$. This means minimization can express all primitive recursive functions as a minimization of a primitive recursive function.

Composition of minimized functions

Now something harder! Consider a composition of minimized primitive recursive functions

$$\begin{aligned}&(\lambda x_1...x_k.\mu n.g_0(n, x_1, .., x_k))\\ &\quad(\mu m_1.g_1(m_1, y_1, ..., y_p))\\ &\quad...\\ &\quad(\mu m_k.g_k(m_k, y_1, ..., y_p))\end{aligned}$$

we wish to represent it as a single minimization over a single primitive recursive function. First we will simplify notation a bit by representing $y_1, .., y_p$ by a single variable y so that we have a slightly more manageable term:

$$(\lambda x_1...x_k.\mu n.g_0(n, x_1, .., x_k))(\mu m_1.g_1(m_1, y))...(\mu m_k.g_k(m_k, y))$$

To achieve this composition we will employ an enumeration of $\mathbb{N}^{k+1}$ satisfying a special property.

We shall write the enumeration as $e_p : \mathbb{N} \longrightarrow \mathbb{N}^p$, and when $e_p(m) = (r_1, .., r_p)$ we will write $e_{p,i}(m) = r_i$ as the function which gives the i^{th}-coordinate. This enumeration to solve our problem must satisfy the following condition: when $e_p(m) = (r_1, ..., r_p)$ then, whenever $r'_i \leq r_i$, for each $i = 1, .., p$, then there must be an $m' \leq m$ with $e_p(m') = (r'_1, .., r'_p)$.

This certainly does not determine the enumeration and, in fact, there are a number of such enumerations which are primitive recursive. Let us consider one such enumeration which is due to Cantor. We need two functions one the inverse of the other:

$$e_2^{-1} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}; (i, j) \mapsto \frac{1}{2}(i + j)(i + j + 1) + j$$

This looks like

$e_2^{-1}(i, j)$	0	1	2	3	4	...
0	0	2	5	9	14	...
1	1	4	8	13	...	
2	3	7	12	...		
3	6	11	...			
4	10	...				
$\vdots$						

Notice that the condition which must be satisfied requires that the numbers in the rectangle determined by (i, j), that is every member of $\{e_2^{-1}(i', j') | i' \leq i \ \& \ j' \leq j\}$ must be less or equal to $e_2^{-1}(i, j)$. This is clearly the case with this enumeration and it can be easily generalized to arbitrary dimensions.

We actually need the inverse of this function which we can write as:

$$e_2 : \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}; n \longrightarrow (i, j) \quad \text{where} \quad \begin{array}{l} c = \lfloor \sqrt{2n} - \frac{1}{2} \rfloor \\ i = n - c(c+1) \\ j = c - i + 2 \end{array}$$

Given such an enumeration now one can define

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

Note that $q(m, y) = 0$ only if each component of the sum is zero – in which case the composite function will be defined. It is now easily checked that the composite required above can be written as

$$\mu n.h(n, y) := \mu n.g(e_{0,k+1}(n), h_1(e_{1,k+1}(n), .., h_k(e_{k,k+1}(n), y)).$$

This allows us to rewrite any composite of minimized primitive recursive functions as a single primitive recursive function with exactly one minimization. This is, thus, a "normal form" for recursive functions:

Proposition 5.2 *Every partial recursive function can be expressed in the form $\mu n.p(n, x_1, ..., x_k)$ where p is a primitive recursive function.*

5.2.1 Minimization in the λ-calculus

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

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

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:

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

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.

Remark: An alternative – and perhaps a simpler – 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.

The attraction of the above method is that it introduced two families of functions which did not require general recursion in their definition: primitive recursive and higher-order primitive recursive functions.

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 β-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 $P(A) = \mathsf{True}$ and $P(B) = \mathsf{False}$. Being **closed to β-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. Consider $P(X) \Leftrightarrow X =_\beta N$ for some is fixed term N. This is obviously closed to equality but it is also non-trivial as the property is true of N itself but cannot be true of both True and False as we know they are not β-equal. If there was a *recursive* predicate P then this would mean equality of λ-terms would be decidable: conversely if there were no such recursive predicate – as the Scott-Curry theorem states – this would mean that equality of λ-terms is undecidable.

2. Consider $P(X) \Leftrightarrow X =_\beta N \wedge \mathsf{nf}(N)$: that is P is true of X precisely when X has a normal form. We know Ω does not have a normal form and that True, being in normal form, certainly has a normal form: thus, this 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 also have a normal form. Thus this P is closed to β-equality. Having a normal form is equivalent to asking whether a program (evaluated by name) terminates ... so we would not expect this to be decidable.

3. A λ-term N is "solvable" when there are $M_1, .., M_n$ (for $n \in \mathbb{N}$ – so n could be 0 such that $N\ M_1 ...\ M_n = I$ (where $I = \lambda x.x$). Equivalently – and this is a theorem – N is solvable if and only if it has a head normal form. This is asking that the reduction of the term will produce *some* information. Consider the predicate $P(X) \Leftrightarrow \mathsf{solvable}(X)$.

   This predicate is non-trivial: as $P(I)$ is certainly true (use $n = 0$!) but also $\Omega := \lambda x.x\ x$ is equally clearly not solvable. To show the predicate is closed with respect to β-equality, note that any term M β-equal to a term N which is solvable must itself be solvable because if $NM_1...M_n =_\beta I$ then as $M =_\beta N$, certainly $MM_1...M_n =_\beta I$. Thus, M is solvable.

   Using Scott-Curry's theorem we can conclude that being solvable is undecidable.

4. The negation of any non-trivial property which is closed to β-equality is automatically non-trivial and closed to equality. Thus, for example we may observe that being insolvable or having no normal form is undecidable as well.

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 Scott-Curry theorem in its positive form:

Theorem 6.1 (Scott-Curry)

Given any recursive predicate P on the terms of the λ-calculus, for which there are terms A and B with $P(\underline{A}) = \neg P(\underline{B})$, then there are terms A' and B' with $A' =_\beta B'$ and $P(\underline{A'}) = \neg P(\underline{B'})$.

For the proof we represent λ-terms internally in the λ-calculus by using the datatype of λ-terms:

```
data Lambda a = Var a
              | App (Lambda a) (Lambda a)
              | Abst a (Lambda a)
```

We may represent the variables by natural numbers. We shall write a term so translated into the λ-calculus as $\underline{N}$. It is clear that these syntactic terms have an application which we will write as

$$\underline{N} \bullet \underline{M} := \mathsf{App}\ \underline{N}\ \underline{M}$$

and, furthermore, that so defined $\underline{N} \bullet \underline{M} = \underline{N\ M}$. Recall we are just translating syntax!

Furthermore, we can define a "translation" function T in the λ-calculus such that $T\underline{N} = \underline{\underline{N}}$. This is surprisingly simple as it is a primitive recursive function. By mapping over the λ-term we can replace the variables by their internal forms. To translate a Church numeral into its internal representation we simply fold (for natural numbers) over the Church numeral replacing Zero with $\underline{\mathsf{Zero}}$ and Succ with $\lambda x.\underline{\mathsf{Succ}} \bullet x$. It then remains to translate the λ-term itself: we can achieve this by a fold (for the datatype of λ-terms) which replaces Var with $\lambda x.\underline{\mathsf{Var}} \bullet x$, App with $\lambda xy.\underline{\mathsf{App}} \bullet x \bullet y$, and Abst with $\lambda xy.\underline{\mathsf{Abst}} \bullet x \bullet y$.

This amounts to setting

$$T := \lambda t.\mathsf{Fold}_\lambda\ (\lambda x.\underline{\mathsf{Var}} \bullet x)\ (\lambda xy.\underline{\mathsf{App}})\ (\lambda xy.\underline{\mathsf{Abst}} \bullet x \bullet y)\ (\mathsf{Map}_\lambda(\mathsf{Fold}_{\mathbb{N}}\ \underline{\mathsf{Zero}}\ (\lambda x.\underline{\mathsf{Succ}} \bullet x))\ t).$$

If the predicate P is recursive we may represent it as a λ-term P which normalizes on *all* the terms $\underline{N}$ of the datatype of λ-terms and produces either True or False depending on whether P is satisfied or not. As the predicate is non-trivial, we then have, without loss of generality, that $P\underline{A} = \mathsf{True}$ and $P\underline{B} = \mathsf{False}$ for some λ-terms A and B.

Lemma 6.2 (Second recursion theorem)

Given any F then there is a λ-term X such that $X =_\beta F\underline{X}$.

Proof: Define

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

where T is the translation function discussed above. This allows:

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

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

Proof: (of theorem 6.1) To complete the proof set

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

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

If $P\underline{X} = \mathsf{True}$: We get $X =_\beta F\underline{X} =_\beta B$ so that setting $A' = X$ and $B' = B$ for the theorem provides $P(\underline{A'}) = P(\underline{X}) = \mathsf{True}$ and $P(\underline{B'}) = P(\underline{B}) = \mathsf{False}$ while $X = \beta B$ proving the theorem.

If $P\underline{X} = \mathsf{False}$: We get $X = F\underline{X} = A$ so that $A' = A$ and $B' = X$ for the theorem $P(\underline{A'}) = P(\underline{A}) = \mathsf{True}$ and $P(\underline{B'}) = P(\underline{X}) = \mathsf{False}$ while $X = \beta A$ proving the theorem.

□

Notice that Theorem 6.1 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 Evaluation of λ-terms

An evaluation of a λ-term essentially involves repeatedly doing β-reductions until a desired normal form is reached. Of course, doing random reductions of a λ-term may not result in a normal form even if there is one. Recall that $\Omega := (\lambda x.xx)\ (\lambda x.xx)$ has a never terminating reduction sequence and this means that in the term

$$\mathsf{If\ True\ False}\ \Omega =: (\lambda xyz.x\ y\ z)(\lambda xy.x)(\lambda xy, y)((\lambda x.x\ x)\ (\lambda x.x\ x))$$

which clearly "should" evaluate to False – a normal form – we must clearly not try to evaluate Ω.

Evaluation strategies may be broadly classified by how one searches for the next **redex**, this is an occurrence of where a β-reduction step could be applied. Here are some basic strategies:

Innermost: Before evaluating a term (with a redex at the root) ensure that all its subterms have been evaluated.

Outermost: Reduce a term with a redex at the root before evaluating any of its subterms.

Leftmost: Reduce redexes to the left of the term before redexes on the right of the term: leftmost evaluation

Rightmost: Reduce redexes to the right of the term before redexes on the left of the term: rightmost evaluation.

A basic **by-value** strategy for evaluation is *rightmost innermost* : this means amongst the innermost redexes (those nearest the leaves) one always chooses the rightmost to reduce. The strategy means that, whenever there are redexes to reduce, there is exactly one redex which is the rightmost innermost redex. Clearly, a by-value reduction strategy will run into trouble when evaluating the term above!

A basic **by-name** evaluation strategy is a *leftmost outermost* strategy: this means that amongst the outermost redexes (i.e. those closest to the root) one should choose the leftmost redex. This, for the λ-calculus is called **normal order evaluation**. This strategy has the important property that if a term has a normal form this strategy will find the normal form.

Evaluation strategies can also be distinguished by where they are designed to stop. The two strategies by-value and by-name only halt when a normal form is reached. However, often it is useful to stop much earlier and this gives another dimension to the rewriting:

Normal form: A rewriting strategy which only halts on a normal form is sometimes called a "strong" rewriting strategy.

Head normal form: A rewriting strategy designed to terminate on a head normal form is called a "head" rewriting strategy. Such strategy repeatedly rewrite the head redex (see below).

Weak head normal form: A rewriting strategy designed to terminate on a weak head normal form is sometimes called a "weak" rewriting strategy. These strategies are the ones usually used by programming languages.

A term is in *normal form* if it contains no redexes. A term is in *head normal form* if it does not contain a **head redex**. A head redex is one of the form

$$\lambda x_1 ... x_n.(\underline{\lambda} z.M)\ N\ N_1\ ...\ N_m$$

where $n, m \in \mathbb{N}$ (which means they can both be zero). Notice that a head redex is necessarily a leftmost outermost redex and that it is found by burrowing through the outer λ-abstractions and then following the "application chain" to left. A term is, thus in **head** normal form if when searching for the head redex one encounters at the head a variable. So a head normal form looks like:

$$\lambda x_1 ... x_n.x_r\ N_1\ ...\ N_m$$

Notice that if a term is in head normal form any subsequent rewriting will take place in the N_i and, thus, will not effect the "head" of the term:

$$\lambda x_1 ... x_n.x_r\ ?\ ...\ ?$$

and this means that one can publish this head as an intermediate result. This is important in modelling potentially infinite computations which can give partial answers as they develop.

A term is in *weak head normal form* if it is a λ-abstraction or it is head normal form. This means that a weak head normal form has one of the two forms:

$$\lambda x.N \qquad \text{or} \qquad x\ N_1\ N_2\ ...\ N_m$$

As before we may characterize a weak head normal form as a term which does not have a **weak head redex**: this is a redex of the form:

$$(\underline{\lambda} x.M)\ N\ N_1\ N_2\ ...\ N_m$$

where again m can be zero. Thus, in searching for a weak head normal form one simply follows the application chain to the left to find a redex. Thus, unlike for a head normal form, one never looks inside λ-abstractions. This is also the basic strategy lazy functional languages like Haskell use for searching for the next rewrite, however, importantly, they also hold the term as a graph so that subterms are not duplicated on performing β-reductions. This makes their rewriting strategies "optimal" in the sense that they perform the minimum number of β-reductions. This may suggest that lazy rewriting is always the best option, however, this is not necessarily the case as there is an overhead associated with managing the graph and, in particular, in recording when a rewrite has been performed.

Here we are mostly concerned with the theoretical properties of normal order reduction: namely that it always manages to find a normal form if ther is one:

Theorem 7.1 (Standardization)
If M has a normal form then a normal order reduction will terminate on that normal form.

PROOF: We shall prove this in two steps: first we shall show that an outermost parallel reduction strategy will always terminate on the normal form. Thus, we shall suppose that we start with a rewriting of M to normal form N which consists of a sequence of parallel rewriting steps. A parallel rewriting step consists of rewriting simultaneously a finite number of redexes which are on different branches of the term (thus in a parallel rewrite no redex can be above another).

Suppose now that we perform two parallel rewrites one after another:

$$M_r \xrightarrow[\| \beta]{} M_{r+1} \xrightarrow[\| \beta]{} M_{r+2}$$

in which some rewrites of the second parallel rewrite are above those of the first parallel rewrite (there may be none). Wherever this happens we can exchange the rewrites which are below a rewrite in the second parallel set with modified rewrites. Suppose we have

$$(\lambda x.N)M \xrightarrow[\| \beta]{} (\lambda x.N')M' \xrightarrow[\beta]{} N'[M'/x]$$

then we may rewrite this as

$$(\lambda x.N)M \xrightarrow[\beta]{} N[M/x] \xrightarrow[\| \beta]{} N'[M'/x]$$

Where notice that as x can occur multiple times (or not at all) in N the parallel rewrite in the second form may, thus, introduce many more parallel redexes – however, the point is that they will all be parallel. This means that we can promote the redexes in the second parallel rewrite which are not dominated by a redex of the first parallel rewrite – and demote any dominated rewrites to obtain an outermost rewriting sequence.

We may then apply this swapping strategy to a given (parallel) rewrite sequence. The plan is to do this repeatedly until all outermost (parallel) rewrites are all performed before inner rewrites. However, we must argue that this process actually terminates. Fortunately, inspecting the term there are a parallel set of outermost redexes. One sweep through the rewriting sequence (starting at the bottom) will bring them into the first rewrite step. The result of this parallel rewrite also has outermost redexes and so another sweep will bring them into the next parallel rewrite. As the length of the rewriting sequence is fixed (or can decrease) this means that we may modify any rewrite sequence into a (parallel) rewrite sequence of the same (or shaorter) length but in which all outermost rewrites are always performed first.

The final step is then to sort the parallel sequence of rewrites so that all leftmost outermost rewrites are performed before rewrites which are further right. Clearly one can always sequentialize rewrites to be leftmost first: it simply involves involves swapping rewrites which are in the non-leftmost order until the reduction is done in leftmost outermost order.

Thus, any rewrite sequence can be modified to be a normal order reduction. □

8 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 k and s. These are subject to the following equations which we orient as rewrites:

$$\begin{array}{rcl} (\mathsf{k} \bullet x) \bullet y & \to & x \\ ((\mathsf{s} \bullet x) \bullet y) \bullet z) & \to & ((x \bullet z) \bullet (y \bullet z) \end{array}$$

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 $\mathsf{s}\bullet\mathsf{k} \neq \mathsf{k}$. 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

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

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

$$p \longrightarrow x_1 | x_2 | ... | x_n | a \in A | p \bullet p$$

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

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

Proposition 8.1 *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 $X = x_1, ..., x_n$ and then define $\lambda^* x_i.p$ as the following polynomial in which the variable x_i has been removed:

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

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

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

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

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

to obtain:

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

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! □

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:

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

Thus $(\mathsf{k} \bullet x) \bullet y = x$ but

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

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.

9 The typed λ-calculus

The typed λ-calculus, as opposed to the "untyped" λ-calculus – which is what we described above - has some rather striking and important properties. Notably, the pure typed λ-calculus has a word problem which is decidable and a rewriting system which is terminating. Once one adds datatypes one obtains a very expressive system which is still terminating but can be viewed as a basic "strong" (in the sense of terminating) programming language.

9.1 The simply typed λ-calculus

We shall start with a fragment which has both λ-abstraction and pairing which assumes a set of atomic types, $\mathcal{A}$, from which one can build all the types using the rules:

$$\boxed{\begin{array}{cc} \dfrac{A \in \mathcal{A}}{A \ \mathsf{type}} & \dfrac{}{1 \ \mathsf{type}} \\[2ex] \dfrac{X \ \mathsf{type} \quad Y \ \mathsf{type}}{X \times Y \ \mathsf{type}} & \dfrac{X \ \mathsf{type} \quad Y \ \mathsf{type}}{X \to Y \ \mathsf{type}} \end{array}}$$

One can then describe the term judgements for the system as follows:

$$\boxed{\begin{array}{cc} \multicolumn{2}{c}{\dfrac{A \in \mathcal{A}}{x : A, \Gamma \vdash x : A} \text{ projection}} \\[2ex] \dfrac{}{\Gamma \vdash () : 1} \text{ empty tuple} & \dfrac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 \times T_2} \text{ pairing} \\[2ex] \dfrac{\Gamma \vdash t : T_0 \times T_1}{\Gamma \vdash \pi_0 t : T_0} \text{ first} & \dfrac{\Gamma \vdash t : T_0 \times T_1}{\Gamma \vdash \pi_1 t : T_1} \text{ second} \\[2ex] \dfrac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \text{ abstraction} & \dfrac{\Gamma \vdash t : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash t\ s : B} \text{ application} \end{array}}$$

There are then just three equalities which can all be viewed as rewrites:

$$\begin{array}{rcl} \pi_0(t_0, t_1) & \longrightarrow & t_0 \\ \pi_1(t_0, t_1) & \longrightarrow & t_1 \\ (\lambda x.t)s & \longrightarrow & t[s/x] \end{array}$$

Proposition 9.1 *These rewrites form a confluent system.*

We sketch a proof which uses a marking argument which extends the rewriting of the λ-calculus with projections for pairs. The argument in fact shows that adding explicit pairs to the untyped λ-calculus results in a confluent system.

We start by introducing marked terms $\underline{\pi}_i(t_0,t_1)$ and, as before, note that there is a by-value evaluation of marked terms:

$$\begin{array}{rcl}
V(x) & = & x \quad \text{for } x \text{ a variable} \\
V(\pi_1) & = & \pi_i \quad \text{for unmarked projections} \\
V(\lambda x.N) & = & \lambda x.(V(N)) \\
V(NM) & = & V(N)V(M) \\
V((N,M)) & = & (V(N),V(M)) \\
V(\underline{\pi}_i(M_0,M_1) & = & V(M_i) \\
V((\underline{\lambda} x.M)N) & = & V(M)[V(N)/x]
\end{array}$$

It then suffices, for the marking argument to go through, to show that

$$\begin{tikzcd}
t_0 \arrow[r] \arrow[d, "*"'] & t_1 \arrow[d, "*"] \\
V(t_0) \arrow[r, dotted] & V(t_1)
\end{tikzcd}$$

As before we may consider the single step rewriting initially to be at the root and then to be in context. In the first case there are two cases: when the single rewrite is marked and when it is not. If it is marked it is easily seen that $V(t_0) = V(t_1)$ We shall reconsider the case when there is an unmarked projection or β-reduction at the root:

$$\begin{tikzcd}
\pi_0(t_1,t_2) \arrow[r] \arrow[d, "*"'] & t_2 \arrow[d, "*"] & (\lambda x.M)N \arrow[r] \arrow[d, "*"'] & M[n/x] \arrow[d, "*"] \\
\pi_0(V(t_1),V(t_2)) \arrow[r] & V(t_2) & (\lambda x.V(M))V(N) \arrow[r] & V(M)[V(N)/x] = V(M[N/x])
\end{tikzcd}$$

These diagram show these one-step diagrams work provided we have the substitution equality (which we leave to the reader). For a rewriting in context we must show:

$$\begin{tikzcd}
C[\![N]\!] \arrow[r] \arrow[d, "*"'] & C[\![M]\!] \arrow[d, "*"] \\
V(C[\![N]\!]) \arrow[r, dotted, "*"'] & V(C[\![M]\!])
\end{tikzcd}$$

and again we have to work by structural induction on the context. It is not hard to show, as before, the useful fact that $V(C[\![t]\!]) = V(C[\![V(t)]\!])$. Now if there is no marked material in the context the result is immediate as no rewriting of the context happens. This means there are two important inductive cases corresponding to building the context with a marked term. A marked projection redex or a marked β-reduction. For a marked projection redex the context must be in one of the terms of the pair: for example $\underline{\pi}_i(t_0, C[\![N]\!])$. When the projection is to the coordinate which does not contain the context his reduces to $V(t_0)$ and the rewrite step $N \longrightarrow M$ is completely removed. So the remaining case is when the projection is to the coordinate of the context: but in this case we have:

$$\begin{tikzcd}
\underline{\pi}_1(t_0, C[\![N]\!]) \arrow[r] \arrow[d, "*"'] \arrow[dr, phantom, "\text{ind}"] & \underline{\pi}_i(t_0, C[\![M]\!]) \arrow[d, "*"] \\
\underline{\pi}_1(V(t_0), V(C[\![V(N)]\!])) \arrow[r, dotted, "*"'] \arrow[d] & \underline{\pi}_i(V(t_0), V(C[\![V(M)]\!])) \arrow[d] \\
V(C[\![V(N)]\!]) \arrow[r, dotted, dash, "*"'] & V(C[\![V(M)]\!])
\end{tikzcd}$$

The argument for a β-reduction is then similar to the argument in the untyped λ-calculus.

9.2 Normalization

We shall first prove that in the simply typed λ-calculus that each term has a normal form. This is called the normalization theorem, but note that it presumes a prescribed order of reduction in order to reach the normal form. Thus, having normalization does not rule out the possibility that, using some strange rewriting order, one could avoid ever reach a normal form. If it is the case that no matter which rewriting order one uses one is *guaranteed* to reach a normal form then the system would be said to be *strongly* normalizing. In fact, the simply typed λ-calculus *is* strongly normalizing – and we shall return to prove this shortly. Meanwhile we will prove normalization.

We prove normalization by showing that there is a series of rewrites which is guaranteed to terminate and, thus, end in a normal form. The technical part of the argument uses a termination argument based on well-founded relations. We start therefore by reviewing well-founded relations.

9.2.1 Well-founded relations and termination

A rewriting system may be viewed as a relation on a set (usually a set of terms): it is **terminating** if the rewriting relation is *well-founded.* A relation $\longrightarrow \subseteq A \times A$ is **well-founded** in case every non-empty set has a minimal element: that is for each subset $S \subseteq A$ there is an $s \in S$ such that there is no $s' \in S$ with $s' \longrightarrow s$. This means, equivalently, that in any subset S there are no infinitely descending chains of the form

$$t \longleftarrow t_1 \longleftarrow t_2 \longleftarrow t_3 \longleftarrow ...$$

as such a chain would not have a minimal element and conversely given an S with no minimal element one would be able to choose an infinitely descending chain. So, for example, β-reduction on the untyped λ-calculus is *not* a well-founded relation as a normal form for a rewriting system is just a minimal element for the rewriting relation and we know there are terms that lack a normal form.

We observe:

Lemma 9.2 *If a relation $\longrightarrow$ is well-founded its transitive closure, $\xrightarrow{+}$, is also well-founded.*

Proof: Suppose S is a non-empty subset we must find a minimal element in the set with respect to $\xrightarrow{+}$. Notice that, we do not change this minimal element if we up-close the set

$$\Uparrow S = \{t \xrightarrow{+} t' | t' \in S\}.$$

Now this set has a minimal element, s, with respect to the original relation, as it is, by assumption, well-founded.

The claim is that it must also be minimal with respect to $\xrightarrow{+}$ in the up-closure. Suppose for contradiction that $t \xrightarrow{+} s$ and $t \in \Uparrow S$ then either $t \longrightarrow s$ (which cannot be) or $t \xrightarrow{+} t' \longrightarrow s$ but then $t' \in \Uparrow S$ and s is not minimal with respect to $\longrightarrow$.. $\square$

In a rewriting system, in which the number of possible rewrites leaving any term is always finite (i.e. each term contains a finite number of redexes), when the rewrite relation is well founded then there is for each term t a bound $\beta(t) \in \mathbb{N}$ on the number of rewrites which a chain leaving that term can have. Note having a bound is a strictly stronger property than being well-founded. However, in many terminating rewrite systems finding an explicit bound (that is a natural number) on the length of any chain of rewrites leaving a term can be quite straightforward.

Lemma 9.3 *A rewriting system $\mathcal{R}$ is terminating if and only if there is a well-founded set $(W, <)$ and a map $\alpha : T_\Omega(X) \longrightarrow W$ such that:*

- *$\alpha(r_i) > \alpha(c_i)$ for each $r_i \longrightarrow c_i \in \mathcal{R}$;*
- *If $\alpha(t) > \alpha(t')$ then $\alpha(t[\sigma]) > \alpha(t'[\sigma])$;*
- *If $\alpha(t_i) > \alpha(t'_i)$ then $\alpha(f(t_1, ..., t_i, ..., t_n)) > \alpha(f(t_1, ..., t'_i, ..., t_n))$.*

Proof: If the rewriting system is terminating then $t \longrightarrow t'$ is well founded so we take α to be the identity map. Conversely given such an α suppose $\emptyset \neq S \subseteq T_\Omega(X)$ then the set $\{\alpha(s) | s \in S\} \subseteq W$ is non-empty and therefore has a minimal element $\alpha(t_0)$. However if $t_0 \longrightarrow t'_0$ is a reduction step which has $t'_0 \in S$ then $\alpha(t'_0) < \alpha(t_0)$ contradicting the minimality of $\alpha(t_0)$. Thus, t_0 is a minimal element in S with respect to the reduction order showing this order is well-founded. $\square$

Thus, to demonstrate that a rewrite system terminates it suffices to exhibit a map α satisfying the conditions of this lemma.

The canonical well-founded set is the natural numbers with the usual relation $n < m$.

If P is a set with a well-founded relation then $P^* = \mathsf{List}(P)$ with the lexicographic relation is also well-founded. The lexicographic relation is given by setting $[]$ to be less that everything and $x : xs < y : ys$ if either $x < y$ or if $x = y$ and $xs < ys$ inductively.

Lemma 9.4 *The lexicographic relation on P^* derived from a well-founder relation on P is itself well-founded.*

Proof: The lexicographic relation is well-founded as given any $X \subset P^*$ if the empty list is in there is automatically a minimal element otherwise choose an element $x_0 \in P$ which is minimal in the first entry and consider all the elements with the same first entry. We argue that this set has a minimal element. Consider the tails $\mathsf{Tail}_{x_0}(X) = \{xs | x_0 : xs \in X\}$, if there is a minimal element in this set then there is a minimal element in X. In fact, if we continue this process of removing minimal first elements, one of two things can happen: either the set reduces to a one element set – and then we have found a minimal element – or we reach a set which has the empty list and again we have found a minimal element. $\square$

Another important source of well-founded relations uses an induced relation on "bags" which was introduced by Dershowitz and Manna.

A bag is an unordered list: this means repetitions are allowed: $\{\!\{x, y, x\}\!\} = \{\!\{y, x, x\}\!\} \neq \{\!\{x, y\}\!\}$ but the order in which the elements occur does not matter. The set $\mathsf{Bag}(P)$ of bags of elements of a set P, with a well-founded relation, can itself be endowed with a well-founded relation. One defines one bag to be less than another in case:

- $\{\!\{x\}\!\} \sqcup b_1 < \{\!\{x\}\!\} \sqcup b_2$ if $b_1 < b_2$;
- $\{\!\{x_1, .., x_n\}\!\} \sqcup b_1 < \{\!\{y\}\!\} \sqcup b_2$ for $n \in \mathbb{N}$ whenever each $x_i < y$ and $b_1 = b_2$ or $b_1 < b_2$.

Notice that the empty bag $\{\!\{\}\!\}$ is minimal because given any bag we can repeatedly remove all elements using the second rule with $n = 0$

Thus we may determine whether one bag is (strictly) less than the other by first removing the elements in common and then for each element in the bigger bag removing all elements in the smaller bag which are strictly less than it. The smaller bag will be emptied by this process.

Intuitively to construct an element strictly smaller than a given b we are allowed to pick an element from the bag and either remove it or replace it with a bag of elements each of which is individually strictly smaller than the element removed. It is not so obvious that this process must always end as you may replace the element with a very large bag of smaller elements!

Proposition 9.5 *If P is a well founded set then the induced relation on $\mathsf{Bag}(P)$ given above is well founded.*

Proof: To prove this is well-founded is not so easy (see Dershowitz and Manna). Here is a sketch: suppose there is an infinite descending chain of bags $b_0 < b_1 < b_2 < ...$ then we may build a forest with roots $x \in b_0$ the children of x are the bag $\{\!\{x_1, .., x_n\}\!\}$ of elements which eventually replace x that is $b_i < b_{i+1}$ (for some i) introduces this strict replacement of x (if this never happens then x is a leaf). Every step in the sequence must do one (or more) replacements of this nature. However, this tree is finitely branching and has all its paths of finite length so itself is finite (Konigs lemma). This means the sequence itself must be finite. □

9.2.2 The argument for weak normalization

The ideas in this section are a slight modification of the ideas in [5].

To show weak normalization we shall use the following measures:

Type height: The height of a type $\mathsf{hgt}(T)$ is defined by:

$$\begin{aligned}
\mathsf{hgt}(1) = \mathsf{hgt}(A) &= 1 \quad \text{for } A \text{ atomic (that is } A \in \mathcal{A})\\
\mathsf{hgt}(T_1 \times T_2) = \mathsf{hgt}(T_1 \to T_2) &= \max(\mathsf{hgt}(T_1), \mathsf{hgt}(T_2)) + 1.
\end{aligned}$$

Redex weight: The redex weight of a term is zero unless it starts with a redex:

$$\begin{aligned}
\mathsf{red}(\pi_i(t_1, t_2)) &= \mathsf{hgt}(T_1 \times T_2) \quad \text{where } (t_1, t_2) : T_1 \times T_2\\
\mathsf{red}((\lambda x.t)s) &= \mathsf{hgt}(T \to S) \quad \text{where } (\lambda x.t) : T \to S\\
\mathsf{red}(t) &= 0 \quad \text{otherwise}
\end{aligned}$$

Finally the bag of redex weights of a term is:

$$\mathsf{rbag}(t) = \{\!\!\{\mathsf{red}(s) | s \ll t \ \& \ \mathsf{red}(s) \neq 0\}\!\!\}$$

where $s \ll t$ means s is a subterm of t. We observe:

Lemma 9.6

(i) Suppose $(\lambda x.t)s \xrightarrow[\beta]{} t[s/x]$ *then provided the weight of this (top) redex is greater than any redex contained in* s *then* $\mathsf{rbag}((\lambda x.t)s) > \mathsf{rbag}(t[s/x])$

(ii) Suppose $\pi_i(t_0, t_1) \longrightarrow t_i$ *then* $\mathsf{rbag}(\pi_i(t_0, t_1)) > \mathsf{rbag}(t_i)$.

(iii) Suppose $C[\![r]\!] \longrightarrow C[\![c]\!]$ *where* $\mathsf{rbag}(r) > \mathsf{rbag}(c)$ *then* $\mathsf{rbag}(C[\![r]\!]) > \mathsf{rbag}(C[\![c]\!])$.

PROOF:

(i) Suppose $(\lambda x.t)s \xrightarrow[\beta]{} t[s/x]$ then the redexes in $t[s/x]$ are:

- redexes of t with s substituted for x, which have the same weight as those in t so are dominated;
- redexes of s: these may be proliferated when s is substituted for x in t, however, their weight is less than the weight of the top redex by assumption so they are dominated by that weight.
- new redexes created by the substitution of the form st'. The weight of these redexes is also dominated by the weight of the top redex as if $s : U$ then $\lambda x.t$ must have type $U \longrightarrow V$ for some type V and $\mathsf{hgt}(U) < \mathsf{hgt}(U \longrightarrow V)$.

(ii) Suppose $\pi_i(t_0, t_1) \longrightarrow t_i$ then $\mathsf{rbag}(\pi_i(t_0, t_1)) > \mathsf{rbag}(t_i)$ as the weights of the redexes of the term eliminated together with that of the top redex are removed.

(iii) This is also immediate as weights of the redexes in the context are unchanged.

□

Whenever we have a redex such that $\mathsf{red}(t) > \mathsf{red}(s)$ for all $s \neq t$ and $s \ll t$ then the lemma tells us that by reducing that redex we can descend in the bag ordering. As we can alway find such a redex – any redex with no redexes below it will do – we can prove that the term must eventually reduce a normal form if one repeatedly reduces such a redex. In particular, note that the by-value reduction strategy *always* has this form. This means every term has a normal form and this means:

Corollary 9.7 *In the typed* λ*-calculus (with products) both by-value and normal-order reduction always terminate.*

Normal order reduction terminates because, if there is a normal form it will be found by a normal order reduction! (Although we have not shown this for products!)

While knowing that terms have a normal form is important even more important is to know that whatever order the reduction is done in that it will always terminate. This, however, requires a more sophisticated argument.

9.2.3 The argument for strong normalization

To prove strong normalization we shall employ a much stronger argument introduced by Tait and modified by Girard. The idea of the proof is to introduce for each type a family of terms called the *reducible* terms. These terms are then proven to satisfy three conditions – called by Girard the "candidats de reductibilité". Finally it is shown by induction that all terms are reducible.

We define $\mathsf{RED}(T)$ for a type T to be the set of terms defined by:

[**Red.1**] For an atomic type $A \in \mathcal{A}$, $t \in \mathsf{RED}(A)$ if and only if it is strongly normalizing;

[**Red.2**] For a product type, $T \times S$, $t \in \mathsf{RED}(T \times S)$ if and only if $\pi_0(t) \in \mathsf{RED}(T)$ and $\pi_1(t) \in \mathsf{RED}(S)$;

[**Red.3**] For an arrow type $T \longrightarrow S$, $u \in \mathsf{RED}(T \longrightarrow S)$ if and only if for every $t \in T$, $ut \in \mathsf{RED}(S)$.

Say that a term is *neutral* if it is not a pair, $\langle u, v \rangle$, or an abstraction, $\lambda x.t$. Thus, a neutral term is either a variable, x, a projection, $\pi_0(t)$ or $\pi_1(t)$, or an application, $t\ s$. We shall now show that the reducible terms satisfy the following three conditions:

[**CR.1**] If $t \in \mathsf{RED}(T)$ then t is strongly normalizable;

[**CR.2**] If $t \in \mathsf{RED}(T)$ and $t \underset{*}{\longrightarrow} t'$ then $t' \in \mathsf{RED}(T)$;

[**CR.3**] If t is neutral and, for every reduction step $t \longrightarrow t'$, $t' \in \mathsf{RED}(T)$ then $t \in \mathsf{RED}(T)$.

In particular, importantly, [**CR.3**] implies that a neutral term, t, which is in normal form is always reducible.

Proposition 9.8 *Reducible terms satisfy* [**CR.1**]-[**CR.2**].

Proof: We work by structural induction:

Atomic types: [**CR.1**] and [**CR.2**] are obviously true. For [**CR.3**] notice that if every one step reduction of t takes one to a strongly normalizing term t' then the term t is certainly strongly normailizing and so in $\mathsf{RED}(A)$.

Product types: A term of product type is reducible if its projections are.

[**CR.1**] If $t \in \mathsf{RED}(U \times V)$ then $\pi_0(t) \in \mathsf{RED}(U)$ and $\pi_1(t) \in \mathsf{RED}(V)$ by assumption [**CR.1**] hold for U and V so both $\pi_0(t)$ and $\pi_1(t)$ are strongly normalizing. Thus, t is strongly normalizing.

[**CR.2**] If $t \in \mathsf{RED}(U \times V)$ and $t \underset{*}{\longrightarrow} t'$ then $\pi_0(t) \underset{*}{\longrightarrow} \pi_0(t')$ and $\pi_1(t) \underset{*}{\longrightarrow} \pi_1(t')$ but by assumption [**CR.2**] holds for U and V so that $\pi_0(t') \in \mathsf{RED}(U)$ and $\pi_1(t') \in \mathsf{RED}(V)$ and so $t' \in \mathsf{RED}(U \times V)$.

[**CR.3**] Now let t be neutral and such that for every one step reduction $t \longrightarrow t'$ we have $t' \in \mathsf{Red}(U \times V)$ then $\pi_0(t') \in \mathsf{RED}(U)$ and $\pi_1(t') \in \mathsf{RED}(V)$. We must show $\pi_0(t)$ and $\pi_1(t)$ are reducible. However, consider a one step reduction of these terms, as t is neutral such a step must be to $\pi_0(t')$ or $\pi_1(t')$ and these are reducible. Now using the inductive assumption this means $\pi_0(t)$ and $\pi_1(t)$ are reducible and thus that, as required, t is reducible.

Arrow types: Recall a term of arrow type is reducible if applied to any reducible term it is reducible.

[CR.1] Suppose $t \in \mathsf{RED}(U \to V)$ this means, letting $u \in \mathsf{RED}(U)$ be a reducible term (there is always such as variables are terms and are always reducible) that $tu \in \mathsf{RED}(V)$ but this means tu is strongly normalizing by assumption. This in turn means t must be strongly normalizing.

[CR.2] Suppose $t \underset{*}{\longrightarrow} t'$ and $t \in \mathsf{RED}(U \to V)$ then for any $u \in \mathsf{RED}(U)$ we have that $tu \in \mathsf{RED}(V)$ but $tu \underset{*}{\longrightarrow} t'u$ and by assumption on V this means $t'u$ is reducible. Thus, $t' \in \mathsf{RED}(U \to V)$.

[CR.3] Not let t be a neutral term and suppose for any $t \to t'$ that $t' \in \mathsf{RED}(U \to V)$. Taking any $u \in \mathsf{RED}(U)$ consider $t\ u$: a one step reduction, as t is neutral, is either of the form $t\ u \to t'\ u$ or $t\ u \to t\ u'$. The former is by assumption reducible the latter case needs some more work. By **[CR.1]** we know that u is strongly normalizing so we can argue for the reducibility of $t\ u$ by induction on the maximum reduction length for u. If it is 0, that is u is in normal form, we are done as either the reduction is in t or $t\ u$ has no reductions: but then as this of type V which satisfies **[CR.3]** it must be reducible. Suppose now we have the result for terms of maximum reduction length N and u has maximal reduction length $N + 1$ then applying the one step reduction $t\ u \to t\ u'$ produces the now by hypothesis reducible term $t\ u'$.

□

Now we proceed to prove that all terms are reducible by proving a slightly stronger statement which allows for the smooth handling of abstractions:

Proposition 9.9 *Any term t with its variables substituted by reducible terms is reducible.*

PROOF: The proof is by structural induction on terms:

Variables: A variable is a reducible term: substituting by a reducible term also produces a reducible term.

Projections: Consider $\pi_0(t)$ (the argument for $\pi_1(t)$ is similar) by inductive assumption $t[u_1/x_1, ..., u_n/x_n]$ is reducible for any reducible terms $u_1, ..., u_n$. However, for $t[u_1/x_1, ..., u_n/x_n]$ to be reducible it is necessary for its projections to be reducible. Thus, in particular $\pi_0(t[u_1/x_1, ..., u_n/x_n])$ is reducible.

Pairs: By assumption $\langle t_0, t_1 \rangle$ has t_0 and t_1 satisfying the hypothesis so $t_0[\tilde{u}/\tilde{x}]$ and $t_1[\tilde{u}/\tilde{x}]$ are reducible. To show $\langle t_0, t_1 \rangle[\tilde{u}/\tilde{x}] = \langle t_0[\tilde{u}/\tilde{x}], t_1[\tilde{u}/\tilde{x}] \rangle$ is reducible we must show that $\pi_0(\langle t_0, t_1 \rangle[\tilde{u}/\tilde{x}])$ and $\pi_1(\langle t_0, t_1 \rangle[\tilde{u}/\tilde{x}])$ are reducible. We shall argue for the former. It suffices to consider one step reductions: clearly the step $\pi_0(\langle t_0, t_1 \rangle[\tilde{u}/\tilde{x}]) \to t_0[\tilde{u}/\tilde{x}]$ gives a reducible term. We must consider reductions within the pair and here we argue on the (maximal) reduction length.

The base case is when the pair is in normal form: here the projection applies and the result follows. The inductive case allows a reduction of the pair to a case in which all one step reductions are reducible and so, by **[CR3]**, it is also reducible!

Applications: Consider $t\ s$: by assumption $t[\tilde{u}/\tilde{x}]$ and $s[\tilde{u}/\tilde{x}]$ are reducible terms. But $t[\tilde{u}/\tilde{x}]$ is reducible only if applying it to all reducible terms yields a reducible term. Thus, $t[\tilde{u}/\tilde{x}]\ s[\tilde{u}/\tilde{x}]$ is certainly reducible.

Abstractions: Consider $\lambda x.t$ where, by assumption $t[v/x, \tilde{u}/\tilde{x}]$ is reducible for any substitution. But then the one step reduction $(\lambda x.t)[\tilde{u}/\tilde{x}]v \longrightarrow t[v/x, \tilde{u}/\tilde{x}]$ produces a reducible term. The other one step reductions must be internal to $t[\tilde{u}/\tilde{x}]$ or to v. Both are reducible terms so one can argue by induction on their reduction length.

In the base case (both are in normal form) one must perform the β-reduction. For the inductive case the term reduces to a term which by **[CR.3]** is reducible. Thus, it is itself reducible.

□

Finally, by choosing the reducible terms to be variables, we then conclude that all terms are reducible. As reducible terms are strongly normalizing, by **[CR.1]**, we obtain the result we sought:

Theorem 9.10 *All terms in the typed λ-calculus with products are strongly normalizing.*

9.3 Higher-order primitive recursive functions

The basic system of the typed λ-calculus with products that we have considered so far is not very expressive. To correct this we shall start adding (inductive) data types. The strategy for strong termination that we shall outline works for *all* inductive data types. However, we shall illustrate the techniques on adding the Booleans and the natural numbers: this gets us to essentially Göedel's system T which is extremely expressive as it allows all higher-order primitive recursive functions. These are also the functions which are provably total in Peano arithmetic.

We introduce new terms and types into the system:

- Bool and Nat are new atomic types.
- There are three new terms involving Bool:

$$\frac{}{\Gamma \vdash \mathsf{True} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{False} : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash y : T \quad \Gamma \vdash b : \mathsf{Bool}}{\Gamma \vdash \mathsf{FoldBool}\ x\ y\ b : T}$$

  The fold can also be seen as the conditional (or case) for this non-recursive data type: $\mathsf{FoldBool}\ x\ y\ b := \mathsf{If}\ b\ x\ y$.

- There are three new terms involving Nat:

$$\frac{}{\Gamma \vdash \mathsf{Zero} : \mathsf{Nat}} \qquad \frac{}{\Gamma \vdash \mathsf{Succ} : \mathsf{Nat} \longrightarrow \mathsf{Nat}}$$

$$\frac{\Gamma \vdash v : T \quad \Gamma \vdash f : T \longrightarrow T \quad \Gamma \vdash n : \mathsf{Nat}}{\Gamma \vdash \mathsf{FoldNat}\ v\ f\ n : T}$$

The fold can be viewed to be an "iterator". Gödels original system had a "recursor" which implemented primitive recursion. The reason for this is that (as discussed above) implementing the "case" construct with just an iterator is rather inefficient (and, in fact, this inefficiency cannot be avoided in a non-higher-order system). However, the two systems are equally expressive and we shall be concerned (here) with only with expressivity (i.e. what functions $f : \mathsf{Nat} \longrightarrow \mathsf{Nat}$ can be expressed) and in this regard the systems are equivalent.

These new terms come with the following rewrites:

$$\begin{array}{rcl} \mathsf{FoldBool}\ x\ y\ \mathsf{True} & \longrightarrow & x \\ \mathsf{FoldBool}\ x\ y\ \mathsf{False} & \longrightarrow & y \\ \mathsf{FoldNat}\ v\ f\ \mathsf{Zero} & \longrightarrow & v \\ \mathsf{FoldNat}\ v\ f\ (\mathsf{Succ} n) & \longrightarrow & f\ (\mathsf{FoldNat}\ v\ f\ n) \end{array}$$

It is not hard to see that adding these rewrites to the basic rewrites of the typed λ-calculus with products gives a confluent system.

Theorem 9.11 *The typed λ-calculus with products, Booleans, and natural numbers with rewrites above gives a strongly normalizing system.*

PROOF: (Sketch)

Amazingly the same proof works!!

We extend the notion of neutrality to be any term which is not of the form $\langle u, v\rangle$, $\lambda x.t$, True, False, Zero, or $\mathsf{Succ}\ n$. Then we have the notion of a reducible term of a type as before and that **[CR.1]**-**[CR.3]** still hold.

We then can prove by structural induction that all terms are reducible. Here is the case for $\mathsf{FoldNat}$:

Suppose v, f, and n are reducible then all are strongly normalizing and so one can reason by induction on $|v| + |f| + |n| + \ell(n)$ where $|t|$ is the maximal reduction length of t and $\ell(n)$ is the size of the normal form of n. In a single reduction step $\mathsf{FoldNat}\ v\ f\ n$ reduces to:

- $\mathsf{FoldNat}\ v'\ f'\ n'$ which is reducible by induction;
- v if $n = \mathsf{Zero}$, which is reducible;
- $f\ (\mathsf{FoldNat}\ v\ f\ m)$ when $n = (\mathsf{Succ}\ m)$. The induction hypothesis tells us that $\mathsf{FoldNat}\ v\ f\ m$ is reducible as $\ell(n) = \ell(\mathsf{Succ}\ m) > \ell(m)$. Thus as f is reducible, $f\ (\mathsf{FoldNat}\ v\ f\ m)$ is reducible.

□

The proof that this system is strongly normalizing does not mean that we can determine equality of functions on the natural numbers! The point is that these rewrites allow the evaluation of functions on values but they do not allow the determination of the equality of functions. For example one cannot even prove that $x + y = y + z$, where $x + y := \mathsf{FoldNat}\ x\ \mathsf{Succ}\ y$ as this latter is already in normal form. Thus, the system is extremely weak in terms of giving a notion of equality but it is, nonetheless, computationally very expressive!

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)

[5] *Proofs and types.* J-Y Girard, Cambridge University Press (1989) (Translated by Y. Lafont and P.Taylor)