Computer Science Tripos, Part IB

Foundations of Functional Programming

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Contents

1 Introduction

This course is concerned with the λ-calculus and its close relative, combinatory logic. The λ-calculus is important to functional programming and to computer science generally:

1. *Variable binding* and *scoping* in block-structured languages can be modelled.

2. Several function calling mechanisms — *call-by-name*, *call-by-value*, and *call-by-need* — can be modelled. The latter two are also known as *strict evaluation* and *lazy evaluation*.

3. The λ-calculus is Turing universal, and is probably the most natural model of computation. *Church's Thesis* asserts that the 'computable' functions are precisely those that can be represented in the λ-calculus.

4. All the usual *data structures* of functional programming, including infinite lists, can be represented. Computation on infinite objects can be defined formally in the λ-calculus.

5. Its notions of *confluence* (Church-Rosser property), *termination*, and *normal form* apply generally in rewriting theory.

6. *Lisp*, one of the first major programming languages, was inspired by the λ-calculus. Many functional languages, such as ML, consist of little more than the λ-calculus with additional syntax.

7. The two main implementation methods, the *SECD machine* (for strict evaluation) and *combinator reduction* (for lazy evaluation) exploit properties of the λ-calculus.

8. The λ-calculus and its extensions can be used to develop better type systems, such as *polymorphism*, and to investigate theoretical issues such as *program synthesis*.

9. *Denotational semantics*, which is an important method for formally specifying programming languages, employs the λ-calculus for its notation.

Hindley and Seldin [7] is a concise introduction to the λ-calculus and combinators. Gordon [6] is oriented towards computer science, overlapping closely with this course. Barendregt [1] is the last word on the λ-calculus.

Acknowledgements. Reuben Thomas pointed out numerous errors in a previous version of these notes.

1.1 The λ-Calculus

Around 1924, Schönfinkel developed a simple theory of functions. In 1934, Church introduced the λ-calculus and used it to develop a formal set theory, which turned out to be inconsistent. More successfully, he used it to formalize the syntax of Whitehead and Russell's massive *Principia Mathematica*. In the 1940s, Haskell B. Curry introduced *combinatory logic*, a variable-free theory of functions.

More recently, Roger Hindley developed what is now known as *type inference*. Robin Milner extended this to develop the polymorphic type system of ML, and published a proof that a well-typed program cannot suffer a run-time type error. Dana Scott developed models of the λ-calculus. With his *domain theory*, he and Christopher Strachey introduced denotational semantics.

Peter Landin used the λ-calculus to analyze Algol 60, and introduced ISWIM as a framework for future languages. His SECD machine, with extensions, was used to implement ML and other strict functional languages. Christopher Wadsworth developed *graph reduction* as a method for performing lazy evaluation of λ-expressions. David Turner applied graph reduction to *combinators*, which led to efficient implementations of lazy evaluation.

Definition 1 The *terms* of the λ-calculus, known as λ-*terms*, are constructed recursively from a given set of *variables* x, y, z, ... They may take one of the following forms:

$$\begin{array}{rl} x & \text{variable} \\ (\lambda x.M) & \text{abstraction, where } M \text{ is a term} \\ (MN) & \text{application, where } M \text{ and } N \text{ are terms} \end{array}$$

We use capital letters like L, M, N, ... for terms. We write $M \equiv N$ to state that M and N are identical λ-terms. The equality between λ-terms, $M = N$, will be discussed later.

1.2 Variable Binding and Substitution

In $(\lambda x.M)$, we call x the *bound variable* and M the *body*. Every occurrence of x in M is *bound* by the abstraction. An occurrence of a variable is *free* if it is not bound by some enclosing abstraction. For example, x occurs bound and y occurs free in $(\lambda z.(\lambda x.(yx))$.

Notations involving free and bound variables exist throughout mathematics. Consider the integral $\int_a^b f(x)dx$, where x is bound, and the product $\Pi_{k=0}^n p(k)$, where k is bound. The quantifiers $\forall$ and $\exists$ also bind variables.

The abstraction $(\lambda x.M)$ is intended to represent the function f such that $f(x) = M$ for all x. Applying f to N yields the result of substituting N for all free occurrences of x in M. Two examples are

$(\lambda x.x)$	The *identity* function, which returns its argument unchanged. It is usually called $\mathbf{I}$.
$(\lambda y.x)$	A *constant* function, which returns x when applied to any argument.

Let us make these concepts precise.

Definition 2 $\mathrm{BV}(M)$, the set of all *bound variables* in M, is given by

$$\begin{aligned}
\mathrm{BV}(x) &= \emptyset \\
\mathrm{BV}(\lambda x.M) &= \mathrm{BV}(M)\bigcup\{x\} \\
\mathrm{BV}(MN) &= \mathrm{BV}(M)\bigcup\mathrm{BV}(N)
\end{aligned}$$

Definition 3 $\mathrm{FV}(M)$, the set of all *free variables* in M, is given by

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) - \{x\} \\
\mathrm{FV}(MN) &= \mathrm{FV}(M)\bigcup\mathrm{FV}(N)
\end{aligned}$$

Definition 4 $M[L/y]$, the result of substituting L for all free occurrences of y in M, is given by

$$\begin{aligned}
x[L/y] &\equiv \begin{cases} L & \text{if } x \equiv y \\ x & \text{otherwise} \end{cases} \\
(\lambda x.M)[L/y] &\equiv \begin{cases} (\lambda x.M) & \text{if } x \equiv y \\ (\lambda x.M[L/y]) & \text{otherwise} \end{cases} \\
(MN)[L/y] &\equiv (M[L/y]\ N[L/y])
\end{aligned}$$

The notations defined above are not themselves part of the λ-calculus. They belong to the metalanguage: they are for talking about the λ-calculus.

1.3 Avoiding Variable Capture in Substitution

Substitution must not disturb variable binding. Consider the term $(\lambda x.(\lambda y.x))$. It should represent the function that, when applied to an argument N, returns the constant function $(\lambda y.N)$. Unfortunately, this does not work if $N \equiv y$; we have defined substitution such that $(\lambda y.x)[y/x] \equiv (\lambda y.y)$. Replacing x by y in the constant function transforms it into the identity function. The free occurrence of x turns into a bound occurrence of y — an example of *variable capture*. If this were allowed to happen, the

λ-calculus would be inconsistent. The substitution $M[N/x]$ is safe provided the bound variables of M are disjoint from the free variables of N:

$$\mathrm{BV}(M) \bigcap \mathrm{FV}(N) = \emptyset.$$

We can always rename the bound variables of M, if necessary, to make this condition true. In the example above, we could change $(\lambda y.x)$ into $(\lambda z.x)$, then obtain the correct substitution $(\lambda z.x)[y/x] \equiv (\lambda z.y)$; the result is indeed a constant function.

1.4 Conversions

The idea that λ-abstractions represent functions is formally expressed through conversion rules for manipulating them. There are α-conversions, β-conversions and η-conversions.

The α-conversion $(\lambda x.M) \to_\alpha (\lambda y.M[y/x])$ renames the abstraction's bound variable from x to y. It is valid provided y does not occur (free or bound) in M. For example, $(\lambda x.(xz)) \to_\alpha (\lambda y.(yz))$. We shall usually ignore the distinction between terms that could be made identical by performing α-conversions.

The β-conversion $((\lambda x.M)N) \to_\beta M[N/x]$ substitutes the argument, N, into the abstraction's body, M. It is valid provided $\mathrm{BV}(M) \bigcap \mathrm{FV}(N) = \emptyset$. For example, $(\lambda x.(xx))(yz) \to_\beta ((yz)(yz))$. Here is another example: $((\lambda z.(zy))(\lambda x.x)) \to_\beta ((\lambda x.x)y) \to_\beta y$.

The η-conversion $(\lambda x.(Mx)) \to_\eta M$ collapses the trivial function $(\lambda x.(Mx))$ down to M. It is valid provided $x \notin \mathrm{FV}(M)$. Thus, M does not depend on x; the abstraction does nothing but apply M to its argument. For example, $(\lambda x.((zy)x)) \to_\eta (zy)$.

Observe that the functions $(\lambda x.(Mx))$ and M always return the same answer, (MN), when applied to any argument N. The η-conversion rule embodies a principle of *extensionality*: two functions are equal if they always return equal results given equal arguments. In some situations, this principle (and η-conversions) are dispensed with.

1.5 Reductions

We say that $M \to N$, or M *reduces to* N, if $M \to_\beta N$ or $M \to_\eta N$. (Because α-conversions are not directional, and are not interesting, we generally ignore them.) The reduction $M \to N$ may consist of applying a conversion to some subterm of M in order to create N. More formally, we could introduce inference rules for $\to$:

$$\frac{M \to M'}{(\lambda x.M) \to (\lambda x.M')} \qquad \frac{M \to M'}{(MN) \to (M'N)} \qquad \frac{M \to M'}{(LM) \to (LM')}$$

If a term admits no reductions then it is in *normal form*. For example, $\lambda xy.y$ and xyz are in normal form. To *normalize* a term means to apply reductions until a normal form is reached. A term *has a normal form* if it can be reduced to a term in normal form. For example, $(\lambda x.x)y$ is not in normal form, but it has the normal form y.

Many λ-terms cannot be reduced to normal form. For instance, $(\lambda x.xx)(\lambda x.xx)$ reduces to itself by β-conversion. Although it is unaffected by the reduction, it is certainly not in normal form. This term is usually called $\mathbf{\Omega}$.

1.6 Curried Functions

The λ-calculus has only functions of one argument. A function with multiple arguments is expressed using a function whose result is another function.

For example, suppose that L is a term containing only x and y as free variables, and we wish to formalize the function $f(x, y) = L$. The abstraction $(\lambda y.L)$ contains x free; for each x, it stands for a function over y. The abstraction $(\lambda x.(\lambda y.L))$ contains no free variables; when applied to the arguments M and N, the result is obtained by replacing x by M and y by N in L. Symbolically, we perform two β-reductions (any necessary α-conversions are omitted):

$$(((\lambda x.(\lambda y.L))M)N) \to_\beta ((\lambda y.L[M/x])N) \to_\beta L[M/x][N/y]$$

This technique is known as *currying* after Haskell B. Curry, and a function expressed using nested λs is known as a *curried function*. In fact, it was introduced by Schönfinkel. Clearly, it works for any number of arguments.

Curried functions are popular in functional programming because they can be applied to their first few arguments, returning functions that are useful in themselves.

1.7 Bracketing Conventions

Abbreviating nested abstractions and applications will make curried functions easier to write. We shall abbreviate

$$\begin{aligned} (\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.M)\ldots)) &\quad \text{as} \quad (\lambda x_1 x_2 \ldots x_n.M) \\ (\ldots(M_1 M_2)\ldots M_n) &\quad \text{as} \quad (M_1 M_2 \ldots M_n) \end{aligned}$$

Finally, we drop outermost parentheses and those enclosing the body of an abstraction. For example,

$$(\lambda x.(x(\lambda y.(yx)))) \text{ can be written as } \lambda x.x(\lambda y.yx).$$

It is vital understand how bracketing works. We have the reduction

$$\lambda z.(\lambda x.M)N \to_\beta \lambda z.M[N/x]$$

but the similar term $\lambda z.z(\lambda x.M)N$ admits no reductions except those occurring within M and N, because $\lambda x.M$ is not being applied to anything. Here is what the application of a curried function (see above) looks like with most brackets omitted:

$$(\lambda xy.L)MN \to_\beta (\lambda y.L[M/x])N \to_\beta L[M/x][N/y]$$

Note that $\lambda x.MN$ abbreviates $\lambda x.(MN)$ rather than $(\lambda x.M)N$. Also, xyz abbreviates $(xy)z$ rather than $x(yz)$.

Exercise 1 What happens in the reduction of $(\lambda xy.L)MN$ if y is free in M?

Exercise 2 Give two different reduction sequences that start at $(\lambda x.(\lambda y.xy)z)y$ and end with a normal form. (These normal forms must be identical: see below.)

2 Equality and Normalization

The λ-calculus is an equational theory: it consists of rules for proving that two λ-terms are equal. A key property is that two terms are equal just if they both can be reduced to the same term.

2.1 Multi-Step Reduction

Strictly speaking, $M \to N$ means that M reduces to N by exactly one reduction step, possibly applied to a subterm of M. Frequently, we are interested in whether M can be reduced to N by any number of steps. Write $M \twoheadrightarrow N$ if

$$M \to M_1 \to M_2 \to \cdots \to M_k \equiv N \qquad (k \geq 0)$$

For example, $((\lambda z.(zy))(\lambda x.x)) \twoheadrightarrow y$. Note that $\twoheadrightarrow$ is the relation $\to^*$, the reflexive/transitive closure of $\to$.

2.2 Equality Between λ-Terms

Informally, $M = M'$ if M can be transformed into M' by performing zero or more reductions and expansions. (An *expansion* is the inverse of a reduction, for instance $y \leftarrow (\lambda x.x)y$.) A typical picture is the following:

$$\begin{tikzcd}[column sep=small]
M \arrow[dr, ->>] & & M_1 \arrow[dl, ->>] \arrow[dr, ->>] & & M_2 \cdots M_{k-1} \arrow[dl, ->>] \arrow[drr, ->>] & & & M_k = M' \arrow[dl, ->>] \\
& N_1 & & N_2 & & \cdots & N_k &
\end{tikzcd}$$

For example, $a((\lambda y.by)c) = (\lambda x.ax)(bc)$ because both sides reduce to $a(bc)$. Note that $=$ is the relation $(\to \cup \to^{-1})^*$, the least equivalence relation containing $\to$.

Intuitively, $M = M'$ means that M and M' have the same value. Equality, as defined here, satisfies all the standard properties. First of all, it is an *equivalence relation* — it satisfies the reflexive, symmetric and associative laws:

$$M = M \qquad \frac{M = N}{N = M} \qquad \frac{L = M \quad M = N}{L = N}$$

Furthermore, it satisfies congruence laws for each of the ways of constructing λ-terms:

$$\frac{M = M'}{(\lambda x.M) = (\lambda x.M')} \qquad \frac{M = M'}{(MN) = (M'N)} \qquad \frac{M = M'}{(LM) = (LM')}$$

The six properties shown above are easily checked by constructing the appropriate diagrams for each equality. They imply that two terms will be equal if we construct them in the same way starting from equal terms. Put another way, if $M = M'$ then replacing M by M' in a term yields an equal term.

Definition 5 *Equality of λ-terms* is the least relation satisfying the six rules given above.

2.3 The Church-Rosser Theorem

This fundamental theorem states that reduction in the λ-calculus is *confluent*: no two sequences of reductions, starting from one λ-term, can reach distinct normal forms. The normal form of a term is independent of the order in which reductions are performed.

Theorem 6 (Church-Rosser) *If $M = N$ then there exists L such that $M \twoheadrightarrow L$ and $N \twoheadrightarrow L$.*

Proof See Barendregt [1] or Hindley and Seldin [7].

For instance, $(\lambda x.ax)((\lambda y.by)c)$ has two different reduction sequences, both leading to the same normal form. The affected subterm is underlined at each step:

$$\begin{array}{c}
\underline{(\lambda x.ax)((\lambda y.by)c)} \to a(\underline{(\lambda y.by)c}) \to a(bc) \\
(\lambda x.ax)(\underline{(\lambda y.by)c}) \to \underline{(\lambda x.ax)(bc)} \to a(bc)
\end{array}$$

The theorem has several important consequences.

- If $M = N$ and N is in normal form, then $M \twoheadrightarrow N$; if a term can transform into normal form using reductions and expansions, then the normal form can be reached by reductions alone.

- If $M = N$ where both terms are in normal form, then $M \equiv N$ (up to renaming of bound variables). Conversely, if M and N are in normal form and are distinct, then $M \neq N$; there is no way of transforming M into N. For example, $\lambda xy.x \neq \lambda xy.y$.

An equational theory is *inconsistent* if all equations are provable. Thanks to the Church-Rosser Theorem, we know that the λ-calculus is consistent. There is no way we could reach two different normal forms by following different reduction strategies. Without this property, the λ-calculus would be of little relevance to computation.

2.4 The Diamond Property

The key step in proving the Church-Rosser Theorem is demonstrating the diamond property — if $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$ then there exists a term L such that $M_1 \twoheadrightarrow L$ and $M_2 \twoheadrightarrow L$. Here is the diagram:

$$\begin{tikzcd}[column sep=small]
 & M \arrow[dl, two heads] \arrow[dr, two heads] & \\
M_1 \arrow[dr, two heads] & & M_2 \arrow[dl, two heads] \\
 & L &
\end{tikzcd}$$

The diamond property is vital: it says that no matter how far we go reducing a term by two different strategies it will always be possible to come together again by further reductions. As for the Church-Rosser Theorem, look again at the diagram for $M = M'$ and note that we can tile the region underneath with diamonds, eventually reaching a common term:

$$\begin{tikzcd}[column sep=small]
M \arrow[dr, two heads] & & M_1 \arrow[dl, two heads] \arrow[dr, two heads] & & M_2 \cdots M_{k-1} \arrow[dl, two heads] & \arrow[dr, two heads] & & M_k = M' \arrow[dl, two heads] \\
 & N_1 \arrow[dr, two heads] & & N_2 \arrow[dl, two heads] \arrow[dr, two heads] & & \cdots & N_k \arrow[dl, two heads] & \\
 & & L_1 \arrow[dr, two heads] & & L_2 \cdots L_{k-1} \arrow[dl, two heads] & {} & & \\
 & & & K_1 & \iddots & & & \\
 & & \ddots & & \iddots & & & \\
 & & & E & & & &
\end{tikzcd}$$

2.5 Proving the Diamond Property

Note that $\rightarrow$ (one-step reduction) does *not* satisfy the diamond property

$$\begin{tikzcd}[column sep=small]
 & M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr] & & M_2 \arrow[dl] \\
 & L &
\end{tikzcd}$$

Consider the term $(\lambda x.xx)(\,\mathbf{I}\,a)$, where $\mathbf{I} \equiv \lambda x.x$. In one step, it reduces to $(\lambda x.xx)a$ or to $(\,\mathbf{I}\,a)(\,\mathbf{I}\,a)$. These both reduce eventually to aa, but there is no way to complete the diamond with a single-step reduction:

$$\begin{array}{ccccc} & & (\lambda x.xx)(\mathbf{I}\,a) & & \\ & \swarrow & & \searrow & \\ (\mathbf{I}\,a)(\mathbf{I}\,a) & & & & (\lambda x.xx)a \\ & \ddots & & \swarrow & \\ & & a\,a & & \end{array}$$

The problem, of course, is that $(\lambda x.xx)$ replicates its argument, which must then be reduced twice. Note also that the difficult cases involve one possible reduction contained inside another. Reductions that do not overlap, such as $M \to M'$ and $N \to N'$ in the term xMN, commute trivially to produce $xM'N'$.

The diamond property for $\twoheadrightarrow$ can be proved with the help of a 'strip lemma', which considers the case where $M \to M_1$ (in *one* step) and also $M \twoheadrightarrow M_2$ (possibly *many* steps):

$$\begin{tikzcd} & M \arrow[dl] \arrow[dr, two heads] & \\ M_1 \arrow[dr, two heads] & & M_2 \arrow[dl, two heads] \\ & L & \end{tikzcd}$$

The 'strips' can then be pasted together to complete a diamond. The details involve an extremely tedious case analysis of the possible reductions from various forms of terms.

2.6 Possibility of Nontermination

Although different reduction sequences cannot yield different normal forms, they can yield completely different outcomes: one could terminate while the other runs forever! Typically, if M has a normal form and admits an infinite reduction sequence, it contains a subterm L having no normal form, and L can be erased by a reduction.

For example, recall that $\mathbf{\Omega}$ reduces to itself, where $\mathbf{\Omega} \equiv (\lambda x.xx)(\lambda x.xx)$. The reduction

$$\underline{(\lambda y.a)\mathbf{\Omega}} \to a$$

reaches normal form, erasing the $\mathbf{\Omega}$. This corresponds to a *call-by-name* treatment of functions: the argument is not reduced but substituted 'as is' into the body of the abstraction.

Attempting to normalize the argument generates a nonterminating reduction sequence:

$$(\lambda y.a)\mathbf{\Omega} \to (\lambda y.a)\mathbf{\Omega} \to \cdots$$

Evaluating the argument before substituting it into the body corresponds to a *call-by-value* treatment of function application. In this example, the call-by-value strategy never reaches the normal form.

2.7 Normal Order Reduction

The *normal order* reduction strategy is, at each step, to perform the leftmost outermost β-reduction. (The η-reductions can be left until last.) *Leftmost* means, for instance, to reduce L before N in LN. *Outermost* means, for instance, to reduce $(\lambda x.M)N$ before reducing M or N.

Normal order reduction corresponds to call-by-name evaluation. By the Standardization Theorem, it always reaches a normal form if one exists. The proof is omitted. However, note that reducing L first in LN may transform L into an abstraction, say $\lambda x.M$. Reducing $(\lambda x.M)N$ may erase N.

2.8 Lazy Evaluation

From a theoretical standpoint, normal order reduction is the optimal, since it always yields a normal form if one exists. For practical computation, it is hopelessly inefficient. Assume that we have a coding of the natural numbers (for which see the next section!) and define a squaring function $\mathbf{sqr} \equiv \lambda n.\, \mathbf{mult}\, nn$. Then

$$\mathbf{sqr}\,(\,\mathbf{sqr}\,N) \rightarrow\, \mathbf{mult}\,(\,\mathbf{sqr}\,N)(\,\mathbf{sqr}\,N) \rightarrow\, \mathbf{mult}\,(\,\mathbf{mult}\,NN)(\,\mathbf{mult}\,NN)$$

and we will have to evaluate four copies of the term N! Call-by-value would have evaluated N (only once) beforehand, but, as we have seen, it can result in nontermination.

Note: multi-letter identifiers (like $\mathbf{sqr}$) are set in bold type, or underlined, in order to prevent confusion with a series of separate variables (like sqr).

Lazy evaluation, or *call-by-need*, never evaluates an argument more than once. An argument is not evaluated unless the value is actually required to produce the answer; even then, the argument is only evaluated to the extent needed (thereby allowing infinite lists). Lazy evaluation can be implemented by representing the term by a graph rather than a tree. Each shared graph node represents a subterm whose value is needed more than once. Whenever that subterm is reduced, the result overwrites the node, and the other references to it will immediately have access to the replacement.

Graph reduction is inefficient for the λ-calculus because subterms typically contain free variables. During each β-reduction, the abstraction's body must be copied. Graph reduction works much better for combinators, where there are no variables. We shall return to this point later.

3 Encoding Data in the λ-Calculus

The λ-calculus is expressive enough to encode boolean values, ordered pairs, natural numbers and lists — all the data structures we may desire in a functional program. These encodings allow us to model virtually the whole of functional programming within the simple confines of the λ-calculus.

The encodings may not seem to be at all natural, and they certainly are not computationally efficient. In this, they resemble Turing machine encodings and programs. Unlike Turing machine programs, the encodings are themselves of mathematical interest, and return again and again in theoretical studies. Many of them involve the idea that the data can carry its control structure with it.

3.1 The Booleans

An encoding of the booleans must define the terms $\mathbf{true}$, $\mathbf{false}$ and $\mathbf{if}$, satisfying (for all M and N)

$$\begin{aligned} \mathbf{if}\ \mathbf{true}\, MN &= M \\ \mathbf{if}\ \mathbf{false}\, MN &= N. \end{aligned}$$

The following encoding is usually adopted:

$$\begin{aligned} \mathbf{true} &\equiv \lambda xy.x \\ \mathbf{false} &\equiv \lambda xy.y \\ \mathbf{if} &\equiv \lambda pxy.pxy \end{aligned}$$

We have $\mathbf{true} \neq \mathbf{false}$ by the Church-Rosser Theorem, since $\mathbf{true}$ and $\mathbf{false}$ are distinct normal forms. As it happens, $\mathbf{if}$ is not even necessary. The truth values are their own conditional operators:

$$\begin{aligned} \mathbf{true}\, MN \equiv (\lambda xy.x)MN \twoheadrightarrow M \\ \mathbf{false}\, MN \equiv (\lambda xy.y)MN \twoheadrightarrow N \end{aligned}$$

These reductions hold for all terms M and N, whether or not they possess normal forms. Note that $\mathbf{if}\, LMN \twoheadrightarrow LMN$; it is essentially an identity function on L. The equations given above even hold as reductions:

$$\begin{aligned} \mathbf{if}\ \mathbf{true}\, MN &\twoheadrightarrow M \\ \mathbf{if}\ \mathbf{false}\, MN &\twoheadrightarrow N. \end{aligned}$$

All the usual operations on truth values can be defined as conditional operator. Here are negation, conjunction and disjunction:

$$\begin{aligned} \mathbf{and} &\equiv \lambda pq.\, \mathbf{if}\, p\ q\, \mathbf{false} \\ \mathbf{or} &\equiv \lambda pq.\, \mathbf{if}\, p\, \mathbf{true}\, q \\ \mathbf{not} &\equiv \lambda p.\, \mathbf{if}\, p\, \mathbf{false}\ \mathbf{true} \end{aligned}$$

3.2 Ordered Pairs

Assume that $\mathbf{true}$ and $\mathbf{false}$ are defined as above. The function $\mathbf{pair}$, which constructs pairs, and the projections $\mathbf{fst}$ and $\mathbf{snd}$, which select the components of a pair, are encoded as follows:

$$\begin{array}{rcl} \mathbf{pair} & \equiv & \lambda xyf.fxy \\ \mathbf{fst} & \equiv & \lambda p.p\,\mathbf{true} \\ \mathbf{snd} & \equiv & \lambda p.p\,\mathbf{false} \end{array}$$

Clearly, $\mathbf{pair}\,MN \twoheadrightarrow \lambda f.fMN$, packaging M and N together. A pair may be applied to any 2-place function of the form $\lambda xy.L$, returning $L[M/x][N/y]$; thus, each pair is its own unpackaging operation. The projections work by this unpackaging operation (which, perhaps, is more convenient in programming than are the projections themselves!):

$$\begin{array}{rcl} \mathbf{fst}\,(\,\mathbf{pair}\,MN) & \twoheadrightarrow & \mathbf{fst}\,(\lambda f.fMN) \\ & \rightarrow & (\lambda f.fMN)\,\mathbf{true} \\ & \rightarrow & \mathbf{true}\,MN \\ & \twoheadrightarrow & M \end{array}$$

Similarly, $\mathbf{snd}\,(\,\mathbf{pair}\,MN) \twoheadrightarrow N$. Observe that the components of $\mathbf{pair}\,MN$ are completely independent; either may be extracted even if the other has no normal form.

Ordered n-tuples could be defined analogously, but nested pairs are a simpler encoding.

3.3 The Natural Numbers

The following encoding of the natural numbers is the original one developed by Church. Alternative encodings are sometimes preferred today, but Church's numerals continue our theme of putting the control structure in with the data structure. Such encodings are elegant; moreover, they work in the second-order λ-calculus (presented in the Types course by Andrew Pitts).

Define

$$\begin{array}{rcl} \underline{0} & \equiv & \lambda fx.x \\ \underline{1} & \equiv & \lambda fx.fx \\ \underline{2} & \equiv & \lambda fx.f(fx) \\ \vdots & \vdots & \vdots \\ \underline{n} & \equiv & \lambda fx.\underbrace{f(\cdots(f}_{n \text{ times}}x)\cdots) \end{array}$$

Thus, for all $n \geq 0$, the Church numeral $\underline{n}$ is the function that maps f to f^n. Each numeral is an iteration operator.

3.4 Arithmetic on Church Numerals

Using this encoding, addition, multiplication and exponentiation can be defined immediately:

$$
\begin{aligned}
\mathbf{add} &\equiv \lambda mnfx.mf(nfx) \\
\mathbf{mult} &\equiv \lambda mnfx.m(nf)x \\
\mathbf{expt} &\equiv \lambda mnfx.nmfx
\end{aligned}
$$

Addition is not hard to check:

$$
\begin{aligned}
\mathbf{add}\ \underline{m}\ \underline{n} &\twoheadrightarrow \lambda fx.\,\underline{m}\,f(\,\underline{n}\,fx) \\
&\twoheadrightarrow \lambda fx.f^m(f^n x) \\
&\equiv \lambda fx.f^{m+n}x \\
&\equiv \underline{m+n}
\end{aligned}
$$

Multiplication is slightly more difficult:

$$
\begin{aligned}
\mathbf{mult}\ \underline{m}\ \underline{n} &\twoheadrightarrow \lambda fx.\,\underline{m}\,(\,\underline{n}\,f)x \\
&\twoheadrightarrow \lambda fx.(\,\underline{n}\,f)^m x \\
&\twoheadrightarrow \lambda fx.(f^n)^m x \\
&\equiv \lambda fx.f^{m\times n}x \\
&\equiv \underline{m\times n}
\end{aligned}
$$

These derivations hold for all Church numerals $\underline{m}$ and $\underline{n}$, but not for all terms M and N.

Exercise 3 Show that $\mathbf{expt}$ performs exponentiation on Church numerals.

3.5 The Basic Operations for Church Numerals

The operations defined so far are not sufficient to define all computable functions on the natural numbers; what about subtraction? Let us begin with some simpler definitions: the successor function and the zero test.

$$
\begin{aligned}
\mathbf{suc} &\equiv \lambda nfx.f(nfx) \\
\mathbf{iszero} &\equiv \lambda n.n(\lambda x.\,\mathbf{false}\,)\,\mathbf{true}
\end{aligned}
$$

The following reductions hold for every Church numeral $\underline{n}$:

$$
\begin{aligned}
\mathbf{suc}\ \underline{n} &\twoheadrightarrow \underline{n+1} \\
\mathbf{iszero}\ \underline{0} &\twoheadrightarrow \mathbf{true} \\
\mathbf{iszero}\,(\,\underline{n+1}\,) &\twoheadrightarrow \mathbf{false}
\end{aligned}
$$

For example,

$$\begin{aligned}
\mathbf{iszero}\,(\underline{n+1}) &\twoheadrightarrow \underline{n+1}\,(\lambda x.\,\mathbf{false}\,)\,\mathbf{true}\\
&\twoheadrightarrow (\lambda x.\,\mathbf{false}\,)^{n+1}\,\mathbf{true}\\
&\equiv (\lambda x.\,\mathbf{false}\,)((\lambda x.\,\mathbf{false}\,)^{n}\,\mathbf{true}\,)\\
&\rightarrow \mathbf{false}
\end{aligned}$$

The predecessor function and subtraction are encoded as follows:

$$\begin{aligned}
\mathbf{prefn} &\equiv \lambda f p.\,\mathbf{pair}\,(f(\,\mathbf{fst}\,p))\,(\,\mathbf{fst}\,p)\\
\mathbf{pre} &\equiv \lambda n f x.\,\mathbf{snd}\,(n(\,\mathbf{prefn}\,f)(\,\mathbf{pair}\,x x))\\
\mathbf{sub} &\equiv \lambda m n.n\mathbf{pre}\,m
\end{aligned}$$

Defining the predecessor function is difficult when each numeral is an iterator. We must reduce an $n+1$ iterator to an n iterator. Given f and x, we must find some g and y such that $g^{n+1}y$ computes $f^n x$. A suitable g is a function on pairs that maps (x,z) to $(f(x),x)$; then

$$g^{n+1}(x,x) = (f^{n+1}(x), f^n(x)).$$

The pair behaves like a one-element delay line.

Above, $\mathbf{prefn}\,f$ constructs the function g. Verifying the following reductions should be routine:

$$\begin{aligned}
\mathbf{pre}\,(\underline{n+1}) &\twoheadrightarrow \underline{n}\\
\mathbf{pre}\,(\underline{0}) &\twoheadrightarrow \underline{0}
\end{aligned}$$

For subtraction, $\mathbf{sub}\ \underline{m}\ \underline{n}$ computes the nth predecessor of $\underline{m}$.

Exercise 4 Show that $\lambda m n.m\,\mathbf{suc}\,n$ performs addition on Church numerals.

3.6 Lists

Church numerals could be generalized to represent lists. The list $[x_1, x_2, \ldots, x_n]$ would essentially be represented by the function that takes f and y to $f x_1(f x_2 \ldots (f x_n y)\ldots)$. Such lists would carry their own control structure with them.

As an alternative, let us represent lists rather as Lisp and ML do — via pairing. This encoding is easier to understand because it is closer to real implementations. The list $[x_1, x_2, \ldots, x_n]$ will be represented by $x_1 :: x_2 :: \ldots :: \mathbf{nil}$. To keep the operations as simple as possible, we shall employ two levels of pairing. Each 'cons cell' $x :: y$ will be represented by $(\,\mathbf{false}\,, (x,y))$, where the $\mathbf{false}$ is a distinguishing tag field. By rights, $\mathbf{nil}$ should be represented by a pair whose first component is $\mathbf{true}$, such as $(\,\mathbf{true}\,,\ \mathbf{true}\,)$, but a simpler definition happens to work. In fact, we could dispense with the tag field altogether.

Here is our encoding of lists:

$$\begin{aligned}
\mathbf{nil} &\equiv \lambda z.z \\
\mathbf{cons} &\equiv \lambda xy.\,\mathbf{pair}\;\mathbf{false}\,(\,\mathbf{pair}\;xy) \\
\mathbf{null} &\equiv \mathbf{fst} \\
\mathbf{hd} &\equiv \lambda z.\,\mathbf{fst}\,(\,\mathbf{snd}\;z) \\
\mathbf{tl} &\equiv \lambda z.\,\mathbf{snd}\,(\,\mathbf{snd}\;z)
\end{aligned}$$

The following properties are easy to verify; they hold for all terms M and N:

$$\begin{aligned}
\mathbf{null}\;\mathbf{nil} &\twoheadrightarrow \mathbf{true} \\
\mathbf{null}\,(\,\mathbf{cons}\;MN) &\twoheadrightarrow \mathbf{false} \\
\mathbf{hd}\,(\,\mathbf{cons}\;MN) &\twoheadrightarrow M \\
\mathbf{tl}\,(\,\mathbf{cons}\;MN) &\twoheadrightarrow N
\end{aligned}$$

Note that $\mathbf{null}\;\mathbf{nil} \twoheadrightarrow \mathbf{true}$ happens really by chance, while the other laws hold by our operations on pairs.

Recall that laws like $\mathbf{hd}\,(\,\mathbf{cons}\;MN) \twoheadrightarrow M$ and $\mathbf{snd}\,(\,\mathbf{pair}\;MN) \twoheadrightarrow N$ hold for all M and N, even for terms that have no normal forms! Thus, $\mathbf{pair}$ and $\mathbf{cons}$ are 'lazy' constructors — they do not 'evaluate their arguments'. Once we introduction recursive definitions, we shall be able to compute with infinite lists.

Exercise 5 Modify the encoding of lists to obtain an encoding of the natural numbers.

4 Writing Recursive Functions in the λ-calculus

Recursion is obviously essential in functional programming. With Church numerals, it is possible to define 'nearly all' computable functions on the natural numbers.[1] Church numerals have an inbuilt source of repetition. From this, we can derive primitive recursion, which when applied using higher-order functions defines a much larger class than the primitive recursive functions studied in Computation Theory. Ackermann's function is not primitive recursive in the usual sense, but we can encode it using Church numerals. If we put

$$\mathbf{ack} \equiv \lambda m.m(\lambda fn.nf(f\,\underline{1}\,))\,\mathbf{suc}$$

then we can derive the recursion equations of Ackermann's function, namely

$$\begin{aligned}
\mathbf{ack}\;\underline{0}\;\underline{n} &= \underline{n+1} \\
\mathbf{ack}\,(\,\underline{m+1}\,)\,\underline{0} &= \mathbf{ack}\;\underline{m}\;\underline{1} \\
\mathbf{ack}\,(\,\underline{m+1}\,)(\,\underline{n+1}\,) &= \mathbf{ack}\;\underline{m}\,(\,\mathbf{ack}\,(\,\underline{m+1}\,)\,\underline{n}\,)
\end{aligned}$$

[1] The precise meaning of 'nearly all' involves heavy proof theory, but all 'reasonable' functions are included.

Let us check the first equation:

$$\begin{aligned}\mathbf{ack}\,\underline{0}\,\underline{n} &\twoheadrightarrow \underline{0}\,(\lambda fn.nf(f\,\underline{1}))\,\mathbf{suc}\,\underline{n}\\ &\twoheadrightarrow \mathbf{suc}\,\underline{n}\\ &\twoheadrightarrow \underline{n+1}\end{aligned}$$

For the other two equations, note that

$$\begin{aligned}\mathbf{ack}\,(\,\underline{m+1}\,)\,\underline{n} &\twoheadrightarrow (\,\underline{m+1}\,)(\lambda fn.nf(f\,\underline{1}))\,\mathbf{suc}\,\underline{n}\\ &\twoheadrightarrow (\lambda fn.nf(f\,\underline{1}))(\,\underline{m}\,(\lambda fn.nf(f\,\underline{1}))\,\mathbf{suc}\,)\,\underline{n}\\ &= (\lambda fn.nf(f\,\underline{1}))(\,\mathbf{ack}\,\underline{m}\,)\,\underline{n}\\ &\twoheadrightarrow \underline{n}\,(\,\mathbf{ack}\,\underline{m}\,)(\,\mathbf{ack}\,\underline{m}\,\underline{1}\,)\end{aligned}$$

We now check

$$\begin{aligned}\mathbf{ack}\,(\,\underline{m+1}\,)\,\underline{0} &\twoheadrightarrow \underline{0}\,(\,\mathbf{ack}\,\underline{m}\,)(\,\mathbf{ack}\,\underline{m}\,\underline{1}\,)\\ &\twoheadrightarrow \mathbf{ack}\,\underline{m}\,\underline{1}\end{aligned}$$

and

$$\begin{aligned}\mathbf{ack}\,(\,\underline{m+1}\,)(\,\underline{n+1}\,) &\twoheadrightarrow \underline{n+1}\,(\,\mathbf{ack}\,\underline{m}\,)(\,\mathbf{ack}\,\underline{m}\,\underline{1}\,)\\ &\twoheadrightarrow \mathbf{ack}\,\underline{m}\,(\,\underline{n}\,(\,\mathbf{ack}\,\underline{m}\,)(\,\mathbf{ack}\,\underline{m}\,\underline{1}\,))\\ &= \mathbf{ack}\,\underline{m}\,(\,\mathbf{ack}\,(\,\underline{m+1}\,)\,\underline{n}\,)\end{aligned}$$

The key to this computation is the iteration of the function $\mathbf{ack}\,\underline{m}$.

4.1 Recursive Functions using Fixed Points

Our coding of Ackermann's function works, but it hardly could be called perspicuous. Even worse would be the treatment of a function whose recursive calls involved something other than subtracting one from an argument — performing division by repeated subtraction, for example.

General recursion can be derived in the λ-calculus. Thus, we can model all recursive function definitions, even those that fail to terminate for some (or all) arguments. Our encoding of recursion is completely uniform and is independent of the details of the recursive definition and the representation of the data structures (unlike the above version of Ackermann's function, which depends upon Church numerals).

The secret is to use a *fixed point combinator* — a term $\mathbf{Y}$ such that $\mathbf{Y}\,F = F(\,\mathbf{Y}\,F)$ for all terms F. Let us explain the terminology. A *fixed point* of the function F is any X such that $FX = X$; here, $X \equiv \mathbf{Y}\,F$. A *combinator* is any λ-term containing no free variables (also called a closed term). To code recursion, F represents the body of the recursive definition; the law $\mathbf{Y}\,F = F(\,\mathbf{Y}\,F)$ permits F to be unfolded as many times as necessary.

4.2 Examples Using Y

We shall encode the factorial function, the append function on lists, and the infinite list $[0, 0, 0, \ldots]$ in the λ-calculus, realising the recursion equations

$$\begin{array}{rcl}
\mathbf{fact}\, N & = & \mathbf{if}\,(\,\mathbf{iszero}\, N)\,\underline{1}\,(\,\mathbf{mult}\, N(\,\mathbf{fact}\,(\mathbf{pre}\, N)))\\
\mathbf{append}\, ZW & = & \mathbf{if}\,(\,\mathbf{null}\, Z)W(\,\mathbf{cons}\,(\,\mathbf{hd}\, Z)(\,\mathbf{append}\,(\,\mathbf{tl}\, Z)W))\\
\mathbf{zeroes} & = & \mathbf{cons}\ \underline{0}\ \mathbf{zeroes}
\end{array}$$

To realize these, we simply put

$$\begin{array}{rcl}
\mathbf{fact} & \equiv & \mathbf{Y}\,(\lambda gn.\,\mathbf{if}\,(\,\mathbf{iszero}\, n)\,\underline{1}\,(\,\mathbf{mult}\, n(g(\mathbf{pre}\, n))))\\
\mathbf{append} & \equiv & \mathbf{Y}\,(\lambda gzw.\,\mathbf{if}\,(\,\mathbf{null}\, z)w(\,\mathbf{cons}\,(\,\mathbf{hd}\, z)(g(\,\mathbf{tl}\, z)w)))\\
\mathbf{zeroes} & \equiv & \mathbf{Y}\,(\lambda g.\,\mathbf{cons}\ \underline{0}\, g)
\end{array}$$

In each definition, the recursive call is replaced by the variable g in $\mathbf{Y}\,(\lambda g.\ldots)$. Let us verify the recursion equation for $\mathbf{zeroes}$; the others are similar:

$$\begin{array}{rcl}
\mathbf{zeroes} & \equiv & \mathbf{Y}\,(\lambda g.\,\mathbf{cons}\ \underline{0}\, g)\\
 & = & (\lambda g.\,\mathbf{cons}\ \underline{0}\, g)(\,\mathbf{Y}\,(\lambda g.\,\mathbf{cons}\ \underline{0}\, g))\\
 & = & (\lambda g.\,\mathbf{cons}\ \underline{0}\, g)\,\mathbf{zeroes}\\
 & \to & \mathbf{cons}\ \underline{0}\ \mathbf{zeroes}
\end{array}$$

4.3 Usage of Y

In general, the recursion equation $M = PM$, where P is any λ-term, is satisfied by defining $M \equiv YP$. Let us consider the special case where M is to be an n-argument function. The equation $Mx_1 \ldots x_n = PM$ is satisfied by defining

$$M \equiv \mathbf{Y}\,(\lambda g x_1 \ldots x_n.Pg)$$

for then

$$\begin{array}{rcl}
Mx_1 \ldots x_n & \equiv & \mathbf{Y}\,(\lambda g x_1 \ldots x_n.Pg)x_1 \ldots x_n\\
 & = & (\lambda g x_1 \ldots x_n.Pg)Mx_1 \ldots x_n\\
 & \twoheadrightarrow & PM
\end{array}$$

Let us realize the mutual recursive definition of M and N, with corresponding bodies P and Q:

$$\begin{array}{rcl}
M & = & PMN\\
N & = & QMN
\end{array}$$

The idea is to take the fixed point of a function F on pairs, such that $F(X, Y) = (PXY, QXY)$. Using our encoding of pairs, define

$$\begin{array}{rcl} L & \equiv & \mathbf{Y}\,(\lambda z.\,\mathbf{pair}\,(P(\,\mathbf{fst}\,z)(\,\mathbf{snd}\,z)) \\ & & \qquad\qquad\quad (Q(\,\mathbf{fst}\,z)(\,\mathbf{snd}\,z))) \\ M & \equiv & \mathbf{fst}\,L \\ N & \equiv & \mathbf{snd}\,L \end{array}$$

By the fixed point property,

$$\begin{array}{rl} L = & \mathbf{pair}\,(P(\,\mathbf{fst}\,L)(\,\mathbf{snd}\,L)) \\ & \qquad\quad (Q(\,\mathbf{fst}\,L)(\,\mathbf{snd}\,L)) \end{array}$$

and by applying projections, we obtain the desired

$$M = P(\,\mathbf{fst}\,L)(\,\mathbf{snd}\,L) = PMN$$

$$N = Q(\,\mathbf{fst}\,L)(\,\mathbf{snd}\,L) = QMN.$$

4.4 Defining Fixed Point Combinators

The combinator $\mathbf{Y}$ was discovered by Haskell B. Curry. It is defined by

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Let us calculate to show the fixed point property:

$$\begin{array}{rcl} \mathbf{Y}\,F & \to & (\lambda x.F(xx))(\lambda x.F(xx)) \\ & \to & F((\lambda x.F(xx))(\lambda x.F(xx))) \\ & = & F(\,\mathbf{Y}\,F) \end{array}$$

This consists of two β-reductions followed by a β-expansion. No reduction $\mathbf{Y}\,F \twoheadrightarrow F(\,\mathbf{Y}\,F)$ is possible! There are other fixed point combinators, such as Alan Turing's Θ:

$$\begin{array}{rcl} A & \equiv & \lambda xy.y(xxy) \\ \Theta & \equiv & AA \end{array}$$

We indeed have the reduction $\Theta F \twoheadrightarrow F(\Theta F)$:

$$\Theta F \equiv AAF \twoheadrightarrow F(AAF) \equiv F(\Theta F)$$

Here is a fixed point combinator discovered by Klop:

$$\begin{array}{rcl} \pounds & \equiv & \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator) \\ \$ & \equiv & \pounds \end{array}$$

The proof of $\$F \twoheadrightarrow F(\$F)$ is left as an exercise. Hint: look at the occurrences of r!

Any fixed point combinator can be used to make recursive definitions under call-by-name reduction. Later, we shall modify $\mathbf{Y}$ to get a fixed point combinator that works with a call-by-value interpreter for the λ-calculus. In practical compilers, recursion should be implemented directly because fixed point combinators are inefficient.

4.5 Head Normal Form

If $M = xM$ then M has no normal form. For if $M \twoheadrightarrow N$ where N is in normal form, then $N = xN$. Since xN is also in normal form, the Church-Rosser Theorem gives us $N \equiv xN$. But clearly N cannot contain itself as a subterm!

By similar reasoning, if $M = PM$ then M usually has no normal form, unless P is something like a constant function or identity function. So anything defined with the help of a fixed point combinator, such as $\mathbf{fact}$, is unlikely to have a normal form.

Although $\mathbf{fact}$ has no normal form, we can still compute with it; $\mathbf{fact}\ \underline{5}$ does have a normal form, namely $\underline{120}$. We can use infinite objects (including functions as above, and also lazy lists) by computing with them for a finite time and requesting a finite part of the result. To formalize this practice, let us define the notion of *head normal form* (hnf).

Definition 7 A term is in *head normal form* (hnf) if and only if it looks like this:

$$\lambda x_1 \ldots x_m . y M_1 \ldots M_k \qquad (m,\, k \geq 0)$$

Examples of terms in hnf include

$$x \qquad \lambda x.y\Omega \qquad \lambda x\, y.x \qquad \lambda z.z((\lambda x.a)c)$$

But $\lambda y.(\lambda x.a)y$ is not in hnf because it admits the so-called *head reduction*

$$\lambda y.(\lambda x.a)y \to \lambda y.a.$$

Let us note some obvious facts. A term in normal form is also in head normal form. Furthermore, if

$$\lambda x_1 \ldots x_m . y M_1 \ldots M_k \twoheadrightarrow N$$

then N *must* have the form

$$\lambda x_1 \ldots x_m . y N_1 \ldots N_k$$

where $M_1 \twoheadrightarrow N_1$, ..., $M_k \twoheadrightarrow N_k$. Thus, a head normal form fixes the outer structure of any further reductions and the final normal form (if any!). And since the arguments M_1, ..., M_k cannot interfere with one another, they can be evaluated independently.

By reducing a term M to hnf we obtain a finite amount of information about the value of M. By further computing the hnfs of M_1, ..., M_k we obtain the next layer of this value. We can continue evaluating to any depth and can stop at any time.

For example, define $\mathbf{ze} \equiv \Theta(\,\mathbf{pair}\ \underline{0}\,)$. This is analogous to $\mathbf{zeroes}$, but uses pairs: $\mathbf{ze} = (0,(0,(0,\ldots)))$. We have

$$\begin{aligned}
\mathbf{ze} &\twoheadrightarrow \mathbf{pair}\ \underline{0}\,\mathbf{ze} \\
&\equiv (\lambda xyf.fxy)\,\underline{0}\,\mathbf{ze} \\
&\twoheadrightarrow \lambda f.f\,\underline{0}\,\mathbf{ze} \\
&\twoheadrightarrow \lambda f.f\,\underline{0}\,(\lambda f.f\,\underline{0}\,\mathbf{ze}) \\
&\twoheadrightarrow \cdots
\end{aligned}$$

With $\lambda f.f\,\underline{0}\,\mathbf{ze}$ we reached a head normal form, which we continued to reduce. We have $\mathbf{fst}\,(\mathbf{ze}) \twoheadrightarrow \underline{0}$ and $\mathbf{fst}\,(\,\mathbf{snd}\,(\mathbf{ze})) \twoheadrightarrow \underline{0}$, since the same reductions work if $\mathbf{ze}$ is a function's argument. These are examples of useful finite computations involving an infinite value.

Some terms do not even have a head normal form. Recall Ω, defined by $\Omega = (\lambda x.xx)(\lambda x.xx)$. A term is reduced to hnf by repeatedly performing leftmost reductions. With Ω we can only do $\Omega \to \Omega$, which makes no progress towards an hnf. Another term that lacks an hnf is $\lambda y.\Omega$; we can only reduce $\lambda y.\Omega \to \lambda y.\Omega$.

It can be shown that if MN has an hnf then so does M. Therefore, if M has no hnf then neither does any term of the form $MN_1N_2\ldots N_k$. A term with no hnf behaves like a *totally undefined function*: no matter what you supply as arguments, evaluation never returns any information. It is not hard to see that if M has no hnf then neither does $\lambda x.M$ or $M[N/x]$, so M really behaves like a black hole. The only way to get rid of M is by a reduction such as $(\lambda x.a)M \to a$. This motivates the following definition.

Definition 8 A term is *defined* if and only if it can be reduced to head normal form; otherwise it is *undefined*.

The exercises below, some of which are difficult, explore this concept more deeply.

Exercise 6 Are the following terms defined? (Here $\mathbf{K} \equiv \lambda xy.x$.)

$$\mathbf{Y} \qquad \mathbf{Y}\;\mathbf{not} \qquad \mathbf{K} \qquad \mathbf{Y}\;\mathbf{I} \qquad x\Omega \qquad \mathbf{Y}\;\mathbf{K} \qquad \mathbf{Y}\,(\,\mathbf{K}\,x) \qquad \underline{n}$$

Exercise 7 A term M is called *solvable* if and only if there exist variables x_1, …, x_m and terms N_1, …, N_n such that

$$(\lambda x_1 \ldots x_m.M)N_1 \ldots N_n = \mathbf{I}.$$

Investigate whether the terms given in the previous exercise are solvable.

Exercise 8 Show that if M has an hnf then M is solvable. Wadsworth proved that M is solvable if and only if M has an hnf, but the other direction of the equivalence is much harder.

4.6 Aside: An Explanation of $\mathbf{Y}$

For the purpose of expressing recursion, we may simply exploit $\mathbf{Y}\,F = F(\,\mathbf{Y}\,F)$ without asking why it holds. However, the origins of $\mathbf{Y}$ have interesting connections with the development of mathematical logic.

Alonzo Church invented the λ-calculus to formalize a new set theory. Bertrand Russell had (much earlier) demonstrated the inconsistency of naive set theory. If we are allowed to construct the set $R \equiv \{x \mid x \notin x\}$, then $R \in R$ if and only if $R \notin R$. This became known as Russell's Paradox.

In his theory, Church encoded sets by their characteristic functions (equivalently, as predicates). The membership test $M \in N$ was coded by the application $N(M)$, which might be true or false. The set abstraction $\{x \mid P\}$ was coded by $\lambda x.P$, where P was some λ-term expressing a property of x.

Unfortunately for Church, Russell's Paradox was derivable in his system! The Russell set is encoded by $R \equiv \lambda x.\,\mathbf{not}\,(xx)$. This implied $RR = \mathbf{not}\,(RR)$, which was a contradiction if viewed as a logical formula. In fact, RR has no head normal form: it is an undefined term like Ω.

Curry discovered this contradiction. The fixed point equation for $\mathbf{Y}$ follows from $RR = \mathbf{not}\,(RR)$ if we replace $\mathbf{not}$ by an arbitrary term F. Therefore, $\mathbf{Y}$ is often called the Paradoxical Combinator.

Because of the paradox, the λ-calculus survives only as an equational theory. The *typed* λ-calculus does not admit any known paradoxes and is used to formalize the syntax of higher-order logic.

4.7 Summary: the λ-Calculus Versus Turing Machines

The λ-calculus can encode the common data structures, such as booleans and lists, such that they satisfy natural laws. The λ-calculus can also express recursive definitions. Because the encodings are technical, they may appear to be unworthy of study, but this is not so.

- The encoding of the natural numbers via Church numerals is valuable in more advanced calculi, such as the second-order λ-calculus.
- The encoding of lists via ordered pairs models their usual implementation on the computer.
- As just discussed, the definition of $\mathbf{Y}$ formalizes Russell's Paradox.
- Understanding recursive definitions as fixed points is the usual treatment in semantic theory.

These constructions and concepts are encountered throughout theoretical computer science. That cannot be said of any Turing machine program!

5 The λ-Calculus and Computation Theory

The λ-calculus is one of the classical models of computation, along with Turing machines and general recursive functions. *Church's Thesis* states that the computable functions are precisely those that are λ-definable. Below, we shall see that the λ-calculus has the same power as the (total) recursive functions. We shall also see some strong undecidability results for λ-terms. The following definition is fundamental.

Definition 9 If f is an n-place function over the natural numbers, then f is *λ-definable* if there exists some λ-term F such that, for all $k_1, \ldots, k_n \in N$,

$$F\,\underline{k_1}\,\ldots\,\underline{k_n} = \underline{f(k_1, \ldots, k_n)}\,.$$

In other words, F maps numerals for arguments to numerals for f's results. By the Church-Rosser Theorem, since numerals are in normal form, we have the reduction

$$F\,\underline{k_1}\,\ldots\,\underline{k_n} \twoheadrightarrow \underline{f(k_1, \ldots, k_n)}\,.$$

Thus, we can compute the value of $f(k_1, \ldots, k_n)$ by normalizing the term $F\,\underline{k_1}\,\ldots\,\underline{k_n}$.

5.1 The Primitive Recursive Functions

In computation theory, the primitive recursive functions are built up from the following basic functions:

0	the constant zero
suc	the successor function
U_n^i	the projection functions, $U_n^i(x_1, \ldots, x_n) = x_i$

New functions are constructed by *substitution* and *primitive recursion*. Substitution, or generalized composition, takes an m-place function g and the n-place functions $h_1, \ldots, h_m$; it yields the n-place function f such that

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)).$$

Primitive recursion takes an n-place function g and an $(n+2)$-place function h; it yields the $(n+1)$-place function f such that

$$\begin{aligned} f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\ f(\,\mathbf{suc}\,(y), x_1, \ldots, x_n) &= h(f(y, x_1, \ldots, x_n), y, x_1, \ldots, x_n) \end{aligned}$$

5.2 Representing the Primitive Recursive Functions

The functions are defined in the λ-calculus in the obvious way. The proof that they are correct (with respect to our definition of λ-definability) is left as an exercise.

Here are the basic functions:

- For 0 use $\underline{0}$, namely $\lambda fx.x$.
- For suc use $\mathbf{suc}$, namely $\lambda nfx.nf(fx)$.
- For U_n^i use $\lambda x_1 \ldots x_n.x_i$.

To handle substitution, suppose that the m-place function g and n-place functions $h_1, \ldots, h_m$ are λ-defined by $G, H_1, \ldots, H_m$, respectively. Then, their composition f is λ-defined by

$$F \equiv \lambda x_1 \ldots x_n . G(H_1 x_1 \ldots x_n) \ldots (H_m x_1 \ldots x_n).$$

To handle primitive recursion, suppose that the n-place function g and $(n+2)$-place function h are λ-defined by G and H, respectively. The primitive recursive function f is λ-defined by

$$\begin{aligned} F \equiv \mathbf{Y} \Big(\lambda f y x_1 \ldots x_n .\, & \textbf{if} \, (\, \mathbf{iszero}\, y) \\ & (G x_1 \ldots x_n) \\ & (H(f(\mathbf{pre}\, y) x_1 \ldots x_n)(\mathbf{pre}\, y) x_1 \ldots x_n)) \Big). \end{aligned}$$

5.3 The General Recursive Functions

Starting with the primitive recursive functions, the so-called *general recursive functions* are obtained by adding the *minimisation operator*, or function inversion. Given an n-place function g it yields the n-place function f such that

$$f(x_1, \ldots, x_n) = \text{the least } y \text{ such that } [g(y, x_2, \ldots, x_n) = x_1]$$

and is undefined if no such y exists.

Thus, minimisation may yield a partial function. This leads to certain difficulties.

- The notion of *undefined* is complicated in the λ-calculus. It is tedious to show that the λ-encoding of minimisation yields an undefined term if and only if the corresponding function fails to terminate.
- Composition in the λ-calculus is non-strict. For example, consider the partial functions r and s such that $r(x) = 0$ and $s(x)$ is undefined for all x. We may λ-define r and s by $R \equiv \lambda x. \underline{0}$ and $S \equiv \lambda x. \Omega$. Now, $r(s(0))$ should be undefined, but $R(S\, \underline{0}\,) \to \underline{0}$. Defining the strict composition of R and S is tricky.

Let us restrict attention to the *total* recursive functions. If for all $x_1, \ldots, x_n$ there is some y such that $g(y, x_2, \ldots, x_n) = x_1$, then the inverse of g is total. Suppose that G is a term that λ-defines g. Then the inverse is λ-defined by

$$\begin{aligned} F \equiv \lambda x_1 \ldots x_n . \mathbf{Y} \Big(\lambda h y .\, & \textbf{if} \, (\mathbf{equals}\, x_1 (G y x_2 \ldots x_n) \\ & y \\ & (h(\, \mathbf{suc}\, y)) \Big) \ \underline{0} \end{aligned}$$

This sets up a recursive function that tests whether $(G y x_2 \ldots x_n)$ equals x_1 for increasing values of y. To start the search, this function is applied to $\underline{0}$. The equality test for natural numbers can be defined in many ways, such as

$$\mathbf{equals} \equiv \lambda m\, n .\, \mathbf{iszero}\, (\, \mathbf{add}\, (\mathbf{sub}\, m\, n)(\mathbf{sub}\, n\, m))$$

This works because $\mathtt{sub}\, mn = \underline{0}$ if $m \leq n$. The equality relation between arbitrary λ-terms (not just for Church numerals) is undecidable and cannot be expressed in the λ-calculus.

Exercise 9 Find another definition of the equality test on the natural numbers.

5.4 The λ-Definable Functions are Recursive

We have just shown that all total recursive functions are λ-definable. The converse is not particularly difficult and we only sketch it. The key is to assign a unique (and recursively computable!) Gödel number $\#M$ to each λ-term M. First, assume that the set of variables has the form $x_1, x_2, x_3, \ldots$, so that each variable is numbered. The definition of $\#M$ is quite arbitrary; for example, we could put

$$\begin{aligned} \#x_i &= 2^i \\ \#(\lambda x_i.M) &= 3^i 5^{\#M} \\ \#(MN) &= 7^{\#M} 11^{\#N} \end{aligned}$$

To show that all λ-definable functions are recursive, suppose that we are given a λ-term F; we must find a recursive function f such that $f(k_1, \ldots, k_n) = k$ if and only if $F\, \underline{k_1}\, \ldots\, \underline{k_n} = \underline{k}$. We can do this in a uniform fashion by writing an interpreter for the λ-calculus using the language of total recursive functions, operating upon the Gödel numbers of λ-terms. This is simply a matter of programming. The resulting program is fairly long but much easier to understand than its Turing machine counterpart, namely the Universal Turing Machine.

Exercise 10 (Composition of partial functions.) Show that $\underline{n}\, \mathbf{I} = \mathbf{I}$ for every Church numeral $\underline{n}$. Use this fact to show that the term H defined by

$$H \equiv \lambda x. G\, x\, \mathbf{I}\, (F(G\, x))$$

λ-defines the composition $f \circ g$ of the functions f and g, provided F λ-defines f and G λ-defines g. What can be said about $H\, M$ if $G\, M$ is undefined? Hint: recall the definition of *solvable* from Exercise 7.

5.5 The Second Fixed Point Theorem

We shall formalize the λ-calculus in itself in order to prove some undecidability results. The Gödel numbering is easily coded and decoded by arithmetic operations. In particular, there exist computable functions Ap and Num, operating on natural numbers, such that

$$\text{Ap}(\#M, \#N) = \#(MN)$$

$$\begin{array}{rcl} & = & 7^{\#M} 11^{\#N} \\ \mathrm{Num}(n) & = & \#(\underline{n}) \\ & = & \#\lambda x_2\, x_1.\, \underbrace{x_2(\cdots(x_2}_{n}\ x_1)\cdots) \end{array}$$

These can be λ-defined by terms $\mathbf{AP}$ and $\mathbf{NUM}$, operating on Church numerals. Let us write $\ulcorner M \urcorner$ for $\underline{(\#M)}$, which is the Church numeral for the Gödel number of M. This itself is a λ-term. Using this notation, $\mathbf{AP}$ and $\mathbf{NUM}$ satisfy

$$\begin{array}{rcll} \mathbf{AP}\ulcorner M\urcorner\ulcorner N\urcorner & = & \ulcorner MN\urcorner & \\ \mathbf{NUM}\ \underline{n} & = & \ulcorner \underline{n}\urcorner & \quad \text{(where } \underline{n} \text{ is any Church numeral)} \end{array}$$

Putting $\ulcorner M\urcorner$ for $\underline{n}$ in the latter equation yields a crucial property:

$$\mathbf{NUM}\ulcorner M\urcorner = \ulcorner\ulcorner M\urcorner\urcorner.$$

Theorem 10 (Second Fixed Point) *If F is any λ-term then there exists a term X such that*

$$F\ulcorner X\urcorner = X.$$

Proof Make the following definitions:

$$\begin{array}{rcl} W & \equiv & \lambda x.F(\mathbf{AP}\,x(\mathbf{NUM}\,x)) \\ X & \equiv & W\ulcorner W\urcorner \end{array}$$

Then, by a β-reduction and by the laws for $\mathbf{AP}$ and $\mathbf{NUM}$, we get

$$\begin{array}{rcl} X & \to & F(\mathbf{AP}\ulcorner W\urcorner(\mathbf{NUM}\ulcorner W\urcorner)) \\ & \twoheadrightarrow & F(\mathbf{AP}\ulcorner W\urcorner\ulcorner\ulcorner W\urcorner\urcorner) \\ & \twoheadrightarrow & F(\ulcorner W\ulcorner W\urcorner\urcorner) \\ & \equiv & F(\ulcorner X\urcorner) \end{array}$$

Therefore $X = F(\ulcorner X\urcorner)$. □

Exercise 11 How do we know that the steps in the proof are actually reductions, rather than mere equalities?

What was the First Fixed Point Theorem? If F is any λ-term then there exists a term X such that $FX = X$. Proof: put $X \equiv \mathbf{Y}\,F$. Note the similarity between $\mathbf{Y}\,F$ and the X constructed above. We use fixed point combinators to λ-define recursive functions; we shall use the Second Fixed Point Theorem to prove undecidability of the halting problem.

5.6 Undecidability Results for the λ-Calculus

We can show that the halting problem, as expressed in the λ-calculus, is undecidable.

Theorem 11 *There is no λ-term* $\mathtt{halts}$ *such that*

$$\mathtt{halts}\ulcorner M\urcorner = \begin{cases} \mathtt{true} & \textit{if } M \textit{ has a normal form} \\ \mathtt{false} & \textit{if } M \textit{ has no normal form} \end{cases}$$

Proof Assume the contrary, namely that $\mathtt{halts}$ exists. Put

$$D \equiv \lambda x.\,\mathbf{if}\,(\,\mathtt{halts}\,x)\Omega\,\underline{0}\,;$$

by the Second Fixed Point Theorem, there exists X such that

$$X = D\ulcorner X\urcorner = \mathbf{if}\,(\,\mathtt{halts}\ulcorner X\urcorner)\Omega\,\underline{0}\,.$$

There are two possible cases; both lead to a contradiction:

- If X has a normal form then $X = \mathbf{if}\ \mathtt{true}\,\Omega\,\underline{0} = \Omega$, which has no normal form!
- If X has no normal form then $X = \mathbf{if}\ \mathtt{false}\,\Omega\,\underline{0} = \underline{0}$, which does have a normal form!

$\square$

The proof is a typical diagonalisation argument. The theorem is strong: although $\mathtt{halts}\ulcorner M\urcorner$ can do anything it likes with the code of M, analysing the term's structure, it cannot determine whether M has a normal form. Assuming Church's Thesis — the computable functions are precisely the λ-definable ones — the theorem states that the halting problem for the normalization of λ-terms is computationally undecidable.

Much stronger results are provable. Dana Scott has shown that

- if $\mathcal{A}$ is *any* non-trivial set of λ-terms (which means that $\mathcal{A}$ is neither empty nor the set of all λ-terms), and
- if $\mathcal{A}$ is closed under equality (which means that $M \in \mathcal{A}$ and $M = N$ imply $N \in \mathcal{A}$)

then the test for membership in A is undecidable. The halting problem follows as a special case, taking

$$\mathcal{A} = \{M \mid M = N \text{ and } N \text{ is in normal form}\}$$

See Barendregt [1, page 143], for more information.

6 ISWIM: The λ-calculus as a Programming Language

Peter Landin was one of the first computer scientists to take notice of the λ-calculus and relate it to programming languages. He observed that Algol 60's scope rules and call-by-name rule had counterparts in the λ-calculus. In his paper [9], he outlined a skeletal programming languages based on the λ-calculus. The title referred to the 700 languages said to be already in existence; in principle, they could all share the same λ-calculus skeleton, differing only in their data types and operations. Landin's language, ISWIM (If you See What I Mean), dominated the early literature on functional programming, and was the model for ML.

Lisp also takes inspiration from the λ-calculus, and appeared many years before ISWIM. But Lisp made several fatal mistakes: dynamic variable scoping, an imperative orientation, and no higher-order functions. Although ISWIM allows imperative features, Lisp is essentially an imperative language, because all variables may be updated.

ISWIM was designed to be extended with application-specific data and operations. It consisted of the λ-calculus plus a few additional constructs, and could be translated back into the pure λ-calculus. Landin called the extra constructs *syntactic sugar* because they made the λ-calculus more palatable.

6.1 Overview of ISWIM

ISWIM started with the λ-calculus:

x	variable
$(\lambda x.M)$	abstraction
(MN)	application

It also allowed local declarations:

$\textit{let}\ x = M\ \mathbf{in}\ N$	simple declaration
$\textit{let}\ f\, x_1 \cdots x_k = M\ \mathbf{in}\ N$	function declaration
$\textit{letrec}\ f\, x_1 \cdots x_k = M\ \mathbf{in}\ N$	recursive declaration

Local declarations could be post-hoc:

$$
\begin{array}{l}
N\ \mathbf{where}\ x = M \\
N\ \mathbf{where}\ f\, x_1 \cdots x_k = M \\
N\ \mathbf{whererec}\ f\, x_1 \cdots x_k = M
\end{array}
$$

The meanings of local declarations should be obvious. They can be translated into the pure λ-calculus:

$$
\begin{array}{ll}
\textit{let}\ x = M\ \mathbf{in}\ N & \equiv (\lambda x.N)M \\
\textit{let}\ f\, x_1 \cdots x_k = M\ \mathbf{in}\ N & \equiv (\lambda f.N)(\lambda x_1 \cdots x_k.M) \\
\textit{letrec}\ f\, x_1 \cdots x_k = M\ \mathbf{in}\ N & \equiv (\lambda f.N)(\,\mathbf{Y}\ (\lambda f x_1 \cdots x_k.M))
\end{array}
$$

Programmers were not expected to encode data using Church numerals and the like. ISWIM provided primitive data structures: integers, booleans and ordered pairs. There being no type system, lists could be constructed by repeated pairing, as in Lisp. The constants included

$0\ 1\ -1\ 2\ -2\ \ldots$	integers
$+\ -\ \times\ /$	arithmetic operators
$=\ \neq\ <\ >\ \leq\ \geq$	relational operators
true false	booleans
and or not	boolean connectives
if E **then** M **else** N	conditional

6.2 Call-by-value in ISWIM

The *call-by-value* rule, rather than *call-by-name*, was usually adopted. This was (and still is) easier to implement; we shall shortly see how this was done, using the SECD machine. Call-by-value is indispensable in the presence of imperative operations.

Call-by-value gives more intuitive and predictable behaviour generally. Classical mathematics is based on strict functions; an expression is undefined unless all its parts are defined. Under call-by-name we can define a function f such that if $f(x) = 0$ for all x, with even $f(1/0) = 0$. Ordinary mathematics cannot cope with such functions; putting them on a rigorous basis requires complex theories.

Under call-by-value, *if-then-else* must be taken as a special form of expression. Treating if as a function makes *fact* run forever:

$$\textit{letrec}\ \mathit{fact}(n) = \mathbf{if}\ (n = 0)\ 1\ (n \times \mathit{fact}(n - 1))$$

The arguments to if are always evaluated, including the recursive call; when $n = 0$ it tries to compute $\mathit{fact}(-1)$. Therefore, we take conditional expressions as primitive, with evaluation rules that return M or N *unevaluated*:

$$\begin{aligned} \mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N &\to M \\ \mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N &\to N \end{aligned}$$

Our call-by-value rule never reduces anything enclosed by a λ. So we can translate the conditional expression to the application of an if-function:

$$\mathbf{if}\ E\ \mathbf{then}\ M\ \mathbf{else}\ N \equiv \mathbf{if}\ E\ (\lambda u.M)\ (\lambda u.N)\ 0$$

Choosing some variable u not free in M or N, enclosing those expressions in λ delays their evaluation; finally, the selected one is applied to 0.

6.3 Pairs, Pattern-Matching and Mutual Recursion

ISWIM includes ordered pairs:

(M, N)	pair constructor
$\mathtt{fst}\ \ \mathtt{snd}$	projection functions

For pattern-matching, let $\lambda(p_1, p_2).E$ abbreviate

$$\lambda z.(\lambda p_1\, p_2.E)(\,\mathbf{fst}\, z)(\,\mathbf{snd}\, z)$$

where p_1 and p_2 may themselves be patterns. Thus, we may write

let $(x, y) = M$ **in** E	taking apart M's value
let $f(x, y) = E$ **in** N	defining f on pairs

The translation iterates to handle things like

$$\textit{let}\ (w, (x, (y, z))) = M\ \textbf{in}\ E.$$

We may introduce n-tuples, writing $(x_1, \ldots, x_{n-1}, x_n)$ for the nested pairs

$$(x_1, \ldots, (x_{n-1}, x_n) \ldots).$$

The mutually recursive function declaration

$$\begin{array}{ll} \textit{letrec} & f_1\, \vec{x}_1 = M_1 \\ \textit{and} & f_2\, \vec{x}_2 = M_2 \\ \vdots & \\ \textit{and} & f_k\, \vec{x}_k = M_k \\ \textit{in}\ N & \end{array}$$

can be translated to an expression involving pattern-matching:

$$(\lambda(f_1, \ldots, f_k).N)(\,\mathbf{Y}\,(\lambda(f_1, \ldots, f_k).(\lambda\vec{x}_1.M_1, \lambda\vec{x}_2.M_2, \ldots, \lambda\vec{x}_k.M_k)))$$

We can easily handle the general case of k mutually recursive functions, each with any number of arguments. Observe the power of syntactic sugar!

6.4 From ISWIM to ML

Practically all programming language features, including go to statements and pointer variables, can be formally specified in the λ-calculus, using the techniques of *denotational semantics*. ISWIM is much simpler than that; it is programming directly in the λ-calculus. To allow imperative programming, we can even define sequential execution, letting $M; N$ abbreviate $(\lambda x.N)M$; the call-by-value rule will evaluate M before N.

However, imperative operations must be adopted as primitive; they cannot be defined by simple translation into the λ-calculus.

ISWIM gives us all the basic features of a programming language — variable scope rules, function declarations, and local declarations. (The *let* declaration is particularly convenient; many languages still make us write assignments for this purpose!) To get a real programming language, much more needs to be added, but the languages so obtained will have a common structure.

ISWIM was far ahead of its time and never found mainstream acceptance. Its influence on ML is obvious. Standard ML has changed the syntax of declarations, added polymorphic types, exceptions, fancier pattern-matching and modules — but much of the syntax is still defined by translation. A French dialect of ML, called CAML, retains much of the traditional ISWIM syntax [4].

6.5 The SECD Machine

Landin invented the SECD machine, an interpreter for the λ-calculus, in order to execute ISWIM programs [3, 5, 8]. A variant of the machine executes instructions compiled from λ-terms. With a few optimisations, it can be used to implement real functional languages, such as ML. SECD machines can be realized as byte-code interpreters, their instructions can be translated to native code, and they can be implemented directly on silicon. The SECD machine yields strict evaluation, call-by-value. A lazy version is much slower than graph reduction of combinators, which we shall consider later.

It is tempting to say that a *value* is any fully evaluated λ-term, namely a term in normal form. This is a poor notion of value in functional programming, for two reasons:

1. Functions themselves should be values, but many functions have no normal form. Recursive functions, coded as $\mathbf{Y}\,F$, satisfy $\mathbf{Y}\,F = F(\mathbf{Y}\,F) = F(F(\mathbf{Y}\,F)) = \cdots$. Although they have no normal form, they may well yield normal forms as results when they are applied to arguments.

2. Evaluating the body of a λ-abstraction, namely the M in $\lambda x.M$, serve little purpose; we are seldom interested in the internal structure of a function. Only when it is applied to some argument N do we demand the result and evaluate $M[N/x]$.

Re (2), we clearly cannot use encodings like $\lambda x\,y.x$ for $\mathtt{true}$ and $\lambda f\,x.x$ for $\underline{0}$, since our evaluation rule will not reduce function bodies. We must take the integers, booleans, pairs, etc., as primitive constants. Their usual functions $(+, -, \times, \ldots)$ must also be primitive constants.

6.6 Environments and Closures

Consider the reduction sequence

$$(\lambda xy.x + y)\,3\,5 \to (\lambda y.3 + y)\,5 \to 3 + 5 \to 8.$$

The β-reduction eliminates the free occurrence of x in $\lambda y.x + y$ by substitution for x. Substitution is too slow to be effective for parameter passing; instead, the SECD machine records $x = 3$ in an *environment*.

With curried functions, $(\lambda xy.x + y)\,3$ is a legitimate value. The SECD machine represents it by a *closure*, packaging the λ-abstraction with its current environment:

$$\begin{array}{lccccccc}
\textbf{Clo}(& y & , & x+y & , & x=3 &) \\
 & \uparrow & & \uparrow & & \uparrow & \\
 & \text{bound variable} & & \text{function body} & & \text{environment} &
\end{array}$$

When the SECD machine applies this function value to the argument 5, it restores the environment to $x = 3$, adds the binding $y = 5$, and evaluates $x + y$ in this augmented environment.

A closure is so-called because it "closes up" the function body over its free variables. This operation is costly; most programming languages forbid using functions as values. Until recently, most versions of Lisp let a function's free variables pick up any values they happened to have in the environment of the call (not that of the function's definition!); with this approach, evaluating

$$\begin{array}{l}
\textit{let } g(y) = x + y \textbf{ in} \\
\textit{let } f(x) = g(1) \;\; \textbf{in} \\
f(17)
\end{array}$$

would return 18, using 17 as the value of x in g! This is *dynamic binding*, as opposed to the usual *static binding*. Dynamic binding is confusing because the scope of x in $f(x)$ can extend far beyond the body of f — it includes all code reachable from f (including g in this case).

Common Lisp, now the dominant version, corrects this long-standing Lisp deficiency by adopting static binding as standard. It also allows dynamic binding, though.

6.7 The SECD State

The SECD machine has a state consisting of four components S, E, C, D:

1. The *Stack* is a list of values, typically operands or function arguments; it also returns the result of a function call.

2. The *Environment* has the form $x_1 = a_1; \cdots ; x_n = a_n$, expressing that the variables $x_1, \ldots, x_n$ have the values $a_1, \ldots, a_n$, respectively.

3. The *Control* is a list of commands. For the interpretive SECD machine, a command is a λ-term or the word **app**; the compiled SECD machine has many commands.

4. The *Dump* is empty $(-)$ or is another machine state of the form (S, E, C, D). A typical state looks like

$$(S_1, E_1, C_1, (S_2, E_2, C_2, \ldots (S_n, E_n, C_n, -) \ldots))$$

   It is essentially a list of triples (S_1, E_1, C_1), (S_2, E_2, C_2), ..., (S_n, E_n, C_n) and serves as the function call stack.

6.8 State Transitions

Let us write SECD machine states as boxes:

$$\begin{array}{|c|}\hline \text{Stack} \\ \hline \text{Environment} \\ \hline \text{Control} \\ \hline \text{Dump} \\ \hline \end{array}$$

To evaluate the λ-term M, the machine begins execution an the *initial state* where M is the Control:

$$\begin{array}{r|c|}\cline{2-2} S & - \\ \cline{2-2} E & - \\ \cline{2-2} C & M \\ \cline{2-2} D & - \\ \cline{2-2} \end{array}$$

If the Control is non-empty, then its first command triggers a state transition. There are cases for constants, variables, abstractions, applications, and the $\mathbf{app}$ command.

A constant is pushed on to the Stack:

$$\begin{array}{|c|}\hline S \\ \hline E \\ \hline k; C \\ \hline D \\ \hline \end{array} \longmapsto \begin{array}{|c|}\hline k; S \\ \hline E \\ \hline C \\ \hline D \\ \hline \end{array}$$

The value of a variable is taken from the Environment and pushed on to the Stack. If the variable is x and E contains $x = a$ then a is pushed:

$$\begin{array}{|c|}\hline S \\ \hline E \\ \hline x; C \\ \hline D \\ \hline \end{array} \longmapsto \begin{array}{|c|}\hline a; S \\ \hline E \\ \hline C \\ \hline D \\ \hline \end{array}$$

A λ-abstraction is converted to a closure, then pushed on to the Stack. The closure contains the current Environment:

$$\begin{array}{|c|}\hline S \\ \hline E \\ \hline \lambda x.M; C \\ \hline D \\ \hline \end{array} \longmapsto \begin{array}{|c|}\hline \mathbf{Clo}(x, M, E); S \\ \hline E \\ \hline C \\ \hline D \\ \hline \end{array}$$

A function application is replaced by code to evaluate the argument and the function, with an explicit **app** instruction:

$$\begin{array}{|c|}\hline S\\\hline E\\\hline MN;C\\\hline D\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline S\\\hline E\\\hline N;M;\mathbf{app};C\\\hline D\\\hline\end{array}$$

The $\mathbf{app}$ command calls the function on top of the Stack, with the next Stack element as its argument. A primitive function, like $+$ or $\times$, delivers its result immediately:

$$\begin{array}{|c|}\hline f;a;S\\\hline E\\\hline \mathbf{app};C\\\hline D\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline f(a);S\\\hline E\\\hline C\\\hline D\\\hline\end{array}$$

The closure $\mathbf{Clo}(x, M, E')$ is called by creating a new state to evaluate M in the Environment E', extended with a binding for the argument. The old state is saved in the Dump:

$$\begin{array}{|c|}\hline \mathbf{Clo}(x,M,E');a;S\\\hline E\\\hline \mathbf{app};C\\\hline D\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline -\\\hline x=a;E'\\\hline M\\\hline (S,E,C,D)\\\hline\end{array}$$

The function call terminates in a state where the Control is empty but the Dump is not. To return from the function, the machine restores the state (S, E, C, D) from the Dump, then pushes a on to the Stack. This is the following state transition:

$$\begin{array}{|c|}\hline a\\\hline E'\\\hline -\\\hline (S,E,C,D)\\\hline\end{array} \longmapsto \begin{array}{|c|}\hline a;S\\\hline E\\\hline C\\\hline D\\\hline\end{array}$$

The result of the evaluation, say a, is obtained from a *final state* where the Control and Dump are empty, and a is the sole value on the Stack:

$$\begin{array}{c|c|}\cline{2-2} S & a\\\cline{2-2} E & -\\\cline{2-2} C & -\\\cline{2-2} D & -\\\cline{2-2}\end{array}$$

6.9 A Sample Evaluation

To demonstrate how the SECD machine works, let us evaluate the expression *twice* $\mathbf{sqr}$ 3, where *twice* is $\lambda f\,x.f(f\,x)$ and $\mathbf{sqr}$ is a built-in squaring function. (Note that *twice* is just the Church numeral $\underline{2}$). We start from the initial state:

$$\begin{array}{|c|}\hline - \\\hline - \\\hline \mathit{twice}\ \mathbf{sqr}\ 3 \\\hline - \\\hline \end{array} \overset{\text{applic}}{\longmapsto} \begin{array}{|c|}\hline - \\\hline - \\\hline 3;\ \mathit{twice}\ \mathbf{sqr}\,;\mathbf{app} \\\hline - \\\hline \end{array} \overset{\text{const}}{\longmapsto} \begin{array}{|c|}\hline 3 \\\hline - \\\hline \mathit{twice}\ \mathbf{sqr}\,;\mathbf{app} \\\hline - \\\hline \end{array} \overset{\text{applic}}{\longmapsto}$$

$$\begin{array}{|c|}\hline 3 \\\hline - \\\hline \mathbf{sqr}\,;\ \mathit{twice}\,;\mathbf{app};\mathbf{app} \\\hline - \\\hline \end{array} \overset{\text{const}}{\longmapsto} \begin{array}{|c|}\hline \mathbf{sqr}\,;3 \\\hline - \\\hline \mathit{twice}\,;\mathbf{app};\mathbf{app} \\\hline - \\\hline \end{array} \overset{\text{abstr}}{\longmapsto}$$

$$\begin{array}{|c|}\hline \mathbf{Clo}(f,\lambda x.f(f\,x),-);\ \mathbf{sqr}\,;3 \\\hline - \\\hline \mathbf{app};\mathbf{app} \\\hline - \\\hline \end{array} \overset{\text{app}}{\longmapsto} \begin{array}{|c|}\hline - \\\hline f=\ \mathbf{sqr} \\\hline \lambda x.f(f\,x) \\\hline (3,-,\mathbf{app},-) \\\hline \end{array} \overset{\text{abstr}}{\longmapsto}$$

$$\begin{array}{|c|}\hline \mathbf{Clo}(x,f(f\,x),f=\ \mathbf{sqr}\,) \\\hline f=\ \mathbf{sqr} \\\hline - \\\hline (3,-,\mathbf{app},-) \\\hline \end{array} \overset{\text{return}}{\longmapsto} \begin{array}{|c|}\hline \mathbf{Clo}(x,f(f\,x),f=\ \mathbf{sqr}\,);3 \\\hline - \\\hline \mathbf{app} \\\hline - \\\hline \end{array} \overset{\text{app}}{\longmapsto}$$

$$\begin{array}{|c|}\hline - \\\hline x=3;f=\ \mathbf{sqr} \\\hline f(f\,x) \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{applic}}{\longmapsto} \begin{array}{|c|}\hline - \\\hline x=3;f=\ \mathbf{sqr} \\\hline f\,x;f;\mathbf{app} \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{applic}}{\longmapsto}$$

$$\begin{array}{|c|}\hline - \\\hline x=3;f=\ \mathbf{sqr} \\\hline x;f;\mathbf{app};f;\mathbf{app} \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{var}}{\longmapsto} \begin{array}{|c|}\hline 3 \\\hline x=3;f=\ \mathbf{sqr} \\\hline f;\mathbf{app};f;\mathbf{app} \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{var}}{\longmapsto} \begin{array}{|c|}\hline \mathbf{sqr}\,;3 \\\hline x=3;f=\ \mathbf{sqr} \\\hline \mathbf{app};f;\mathbf{app} \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{apply}}{\longmapsto}$$

$$\begin{array}{|c|}\hline 9 \\\hline x=3;f=\ \mathbf{sqr} \\\hline f;\mathbf{app} \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{var}}{\longmapsto} \begin{array}{|c|}\hline \mathbf{sqr}\,;9 \\\hline x=3;f=\ \mathbf{sqr} \\\hline \mathbf{app} \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{app}}{\longmapsto} \begin{array}{|c|}\hline 81 \\\hline x=3;f=\ \mathbf{sqr} \\\hline - \\\hline (-,-,-,-) \\\hline \end{array} \overset{\text{return}}{\longmapsto} \begin{array}{|c|}\hline 81 \\\hline - \\\hline - \\\hline - \\\hline \end{array}$$

The machine terminates in a final state, giving a value of 81.

6.10 The Compiled SECD Machine

It takes 17 steps to evaluate $((\lambda x\, y.x{+}y)\, 3)\, 5$! Much faster execution is obtained by first compiling the λ-term. Write $[\![M]\!]$ for the list of commands produced by compiling M; there are cases for each of the four kinds of λ-term.

Constants are compiled to the const command, which will (during later execution of the code) push a constant onto the Stack:

$$[\![k]\!] = \mathbf{const}(k)$$

Variables are compiled to the var command, which will push the variable's value, from the Environment, onto the Stack:

$$[\![x]\!] = \mathbf{var}(x)$$

Abstractions are compiled to the closure command, which will push a closure onto the Stack. The closure will include the current Environment and will hold M as a list of commands, from compilation:

$$[\![\lambda x.M]\!] = \mathbf{closure}(x, [\![M]\!])$$

Applications are compiled to the app command at compile time. Under the interpreted SECD machine, this work occurred at run time:

$$[\![MN]\!] = [\![N]\!]; [\![M]\!]; \mathbf{app}$$

We could add further instructions, say for *conditionals*. Let $\mathtt{test}(C_1, C_2)$ be replaced by C_1 or C_2, depending upon whether the value on top of the Stack is $\mathtt{true}$ or $\mathtt{false}$:

$$[\![\,\textbf{if}\ \ E\ \textbf{then}\ M\ \textbf{else}\ N]\!] = [\![E]\!]; \mathtt{test}([\![M]\!], [\![N]\!])$$

To allow built-in 2-place functions such as $+$ and $\times$ could be done in several ways. Those functions could be made to operate upon ordered pairs, constructed using a **pair** instruction. More efficient is to introduce arithmetic instructions such as $\mathtt{add}$ and $\mathtt{mult}$, which pop both their operands from the Stack. Now $((\lambda x\, y.x + y)\, 3)\, 5$ compiles to

$$\mathbf{const}(5); \mathbf{const}(3); \mathbf{closure}(x, C_0); \mathbf{app}; \mathbf{app}$$

and generates two further lists of commands:

$$\begin{aligned} C_0 &= \mathbf{closure}(y, C_1) \\ C_1 &= \mathbf{var}(y); \mathbf{var}(x);\ \mathtt{add} \end{aligned}$$

Many further optimisations can be made, leading to an execution model quite close to conventional hardware. Variable names could be removed from the Environment, and bound variables referred to by depth rather than by name. Special instructions **enter** and **exit** could efficiently handle functions that are called immediately (say, those created by the declaration $\mathbf{let}\, x = N \,\mathbf{in}\, M$), creating no closure:

$$[\![(\lambda x.M)N]\!] = [\![N]\!]; \mathbf{enter}; [\![M]\!]; \mathbf{exit}$$

Tail recursive (sometimes called *iterative*) function calls could be compiled to the **tailapp** command, which would cause the following state transition:

$$\begin{array}{|c|}\hline \mathbf{Clo}(x, C, E'); a \\ \hline E \\ \hline \mathbf{tailapp} \\ \hline D \\ \hline \end{array} \longmapsto \begin{array}{|c|}\hline - \\ \hline x = a; E' \\ \hline C \\ \hline D \\ \hline \end{array}$$

The useless state $(-, E, -, D)$ is never stored on the dump, and the function return after **tailapp** is never executed — the machine jumps directly to C!

6.11 Recursive Functions

The usual fixed point combinator, $\mathbf{Y}$, fails under the SECD machine; it always loops. A modified fixed point combinator, including extra λ's to delay evaluation, does work:

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

But it is hopelessly slow! Recursive functions are best implemented by creating a closure with a pointer back to itself.

Suppose that $f(x) = M$ is a recursive function definition. The value of f is represented by $\mathbf{Y}\,(\lambda f\, x.M)$. The SECD machine should interpret $\mathbf{Y}\,(\lambda f x.M)$ in a special manner, applying the closure for $\lambda f\, x.M$ to a dummy value, $\perp$. If the current Environment is E then this yields the closure

$$\mathbf{Clo}(x,\ M,\ f = \perp; E)$$

Then the machine modifies the closure, replacing the $\perp$ by a pointer looping back to the closure itself:

$$\mathbf{Clo}(x,\ M,\ f = \cdot; E)$$

When the closure is applied, recursive calls to f in M will re-apply the same closure. The cyclic environment supports recursion efficiently.

The technique is called “tying the knot” and works only for function definitions. It does not work for recursive definitions of data structures, such as the infinite list $[0, 0, 0, \ldots]$, defined as $\mathbf{Y}\,(\lambda l.\,\mathbf{cons}\ 0\, l)$. Therefore strict languages like ML allow only functions to be recursive.

7 Lazy Evaluation via Combinators

The SECD machine employs call-by-value. It can be modified for call-by-need (lazy evaluation), as follows. When a function is called, its argument is stored *unevaluated* in a closure containing the current environment. Thus, the call MN is treated something like $M(\lambda u.N)$, where u does not appear in N. This closure is called a *suspension*. When a strict, built-in function is called, such as $+$, its argument is evaluated in the usual way.

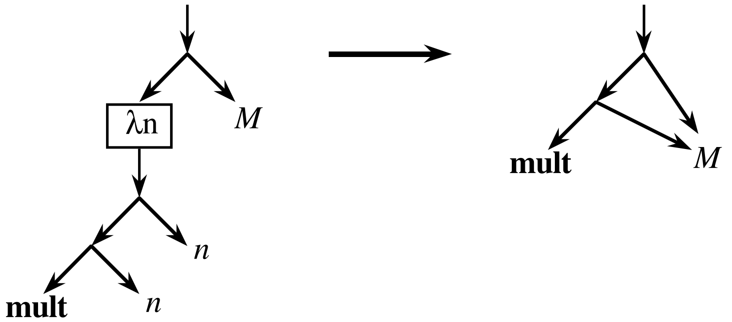
It is essential that no argument be evaluated more than once, no matter how many times it appears in the function's body:

$$\begin{array}{l} \textit{let}\ sqr\, n = n \times n\ \textbf{in}\ N \\ sqr(sqr(sqr\, 2)) \end{array}$$

If this expression were evaluated by repeatedly duplicating the argument of sqr, the waste would be intolerable. Therefore, the lazy SECD machine updates the environment with the value of the argument, after it is evaluated for the first time. But the cost of creating suspensions makes this machine ten times slower than the strict SECD machine, according to David Turner, and compilation can give little improvement.

7.1 Graph Reduction in the λ-Calculus

Another idea is to work directly with λ-terms, using sharing and updating to ensure that no argument is evaluated more than once. For instance, the evaluation of $(\lambda n.n \times n)M$ might be represented by the *graph reduction*



The difficulty here is that λ-abstractions may themselves be shared. We may not modify the body of the abstraction, replacing the bound variable by the actual argument. Instead, we must copy the body — including parts containing no occurrence of the bound variable — when performing the substitution.

Both the lazy SECD machine and graph reduction of λ-terms suffer because of the treatment of bound variables. Combinators have the same expressive power as the λ-calculus, but no bound variables. Graph reduction in combinators does not require copying. David Turner found an efficient method of translating λ-terms into combinators, for evaluation by graph reduction [10]. Offshoots of his methods have been widely adopted for implementing lazy functional languages.

7.2 Introduction to Combinators

In the simplest version, there are only two combinators, $\mathbf{K}$ and $\mathbf{S}$. Combinators are essentially constants. It is possible to define $\mathbf{K}$ and $\mathbf{S}$ in the λ-calculus, but combinatory logic (CL) exists as a theory in its own right.[2]

The terms of combinatory logic, written P, Q, R, ..., are built from $\mathbf{K}$ and $\mathbf{S}$ using application. They may contain free variables, but no bound variables. A typical CL term is $\mathbf{K}\, x(\mathbf{S}\ \mathbf{K}\, x)(\mathbf{K}\ \mathbf{S}\ \mathbf{K}\, y)\, \mathbf{S}$. Although CL is not particularly readable, it is powerful enough to code all the computable functions!

The combinators obey the following reductions:

$$\begin{array}{lll} \mathbf{K}\, P\, Q & \to_w & P \\ \mathbf{S}\, P\, Q\, R & \to_w & P\, R(Q\, R) \end{array}$$

Thus, the combinators could have been defined in the λ-calculus by

$$\begin{array}{rcl} \mathbf{K} & \equiv & \lambda x\, y.x \\ \mathbf{S} & \equiv & \lambda x\, y\, z.x\, z(y\, z) \end{array}$$

But note that $\mathbf{S}\ \mathbf{K}$ does not reduce — because $\mathbf{S}$ requires three arguments — while the corresponding λ-term does. For this reason, combinator reduction is known as *weak reduction* (hence the "w" in $\to_w$).

Here is an example of weak reduction:

$$\mathbf{S}\ \mathbf{K}\ \mathbf{K}\, P \to_w \mathbf{K}\, P(\mathbf{K}\, P) \to_w P$$

Thus $\mathbf{S}\ \mathbf{K}\ \mathbf{K}\, P \twoheadrightarrow_w P$ for all combinator terms P; let us define the identity combinator by $\mathbf{I} \equiv \mathbf{S}\ \mathbf{K}\ \mathbf{K}$.

Many of the concepts of the λ-calculus carry over to combinators. A combinator term P is in *normal form* if it admits no weak reductions. Combinators satisfy a version of the Church-Rosser Theorem: if $P = Q$ (by any number of reductions, forwards or backwards) then there exists a term Z such that $P \twoheadrightarrow_w Z$ and $Q \twoheadrightarrow_w Z$.

7.3 Abstraction on Combinators

Any λ-term may be transformed into a roughly equivalent combinatory term. (The meaning of "roughly" is examined below.) The key is the transformation of a combinatory term P into another combinator term, written as $\lambda^* x.P$ since it behaves like a λ-abstraction.[3]

Definition 12 The operation $\lambda^* x$, where x is a variable, is defined recursively as follows:

$$\begin{array}{lll} \lambda^* x.x & \equiv & \mathbf{I} \\ \lambda^* x.P & \equiv & \mathbf{K}\, P \qquad\qquad (x \text{ not free in } P) \\ \lambda^* x.P\, Q & \equiv & \mathbf{S}\, (\lambda^* x.P)(\lambda^* x.Q) \end{array}$$

[2] It is called combinatory logic for historical reasons; we shall not treat it as a logic.

[3] Some authors write $[x]P$ for $\lambda^* x.P$.

Finally, $\lambda^* x_1 \ldots x_n.P$ abbreviates $\lambda^* x_1.(\ldots \lambda^* x_n.P \ldots)$.

Note that $\lambda^* x$ is not part of the syntax of combinatory logic, but stands for the term constructed as according to the definition above. Here is an example of combinatory abstraction:

$$\begin{aligned}
\lambda^* x\, y.y\, x &\equiv \lambda^* x.(\lambda^* y.y\, x) \\
&\equiv \lambda^* x.\, \mathbf{S}\,(\lambda^* y.y)(\lambda^* y.x) \\
&\equiv \lambda^* x.(\,\mathbf{S}\;\mathbf{I}\,)(\,\mathbf{K}\,x) \\
&\equiv \mathbf{S}\,(\lambda^* x.\,\mathbf{S}\;\mathbf{I}\,)(\lambda^* x.\,\mathbf{K}\,x) \\
&\equiv \mathbf{S}\,(\,\mathbf{K}\,(\,\mathbf{S}\;\mathbf{I}\,))(\,\mathbf{S}\,(\lambda^* x.\,\mathbf{K}\,)(\lambda^* x.x)) \\
&\equiv \mathbf{S}\,(\,\mathbf{K}\,(\,\mathbf{S}\;\mathbf{I}\,))(\,\mathbf{S}\,(\,\mathbf{K}\;\mathbf{K}\,)\,\mathbf{I}\,)
\end{aligned}$$

Each λ^* can *double* the number of applications in a term; in general, growth is exponential. Turner discovered a better abstraction method, discussed in the next section. First, let us show that combinatory abstraction behaves like its λ-calculus cousin. Let FV be defined for combinatory terms in an analogous manner to Definition 3.

Theorem 13 *For every combinatory term P we have*

$$\begin{aligned}
\mathrm{FV}(\lambda^* x.P) &= \mathrm{FV}(P) - \{x\} \\
(\lambda^* x.P)x &\twoheadrightarrow_w P
\end{aligned}$$

Proof We prove both properties independently, by structural induction on P. There are three cases.

If P is the variable x, then $\lambda^* x.x \equiv \mathbf{I}$. Clearly

$$\begin{aligned}
&\mathrm{FV}(\lambda^* x.x) = \mathrm{FV}(\,\mathbf{I}\,) = \emptyset = \mathrm{FV}(x) - \{x\} \\
&(\lambda^* x.x)x \equiv \mathbf{I}\,x \twoheadrightarrow_w x
\end{aligned}$$

If P is any term not containing x, then $\lambda^* x.P \equiv \mathbf{K}\,P$ and

$$\begin{aligned}
&\mathrm{FV}(\lambda^* x.P) = \mathrm{FV}(\,\mathbf{K}\,P) = \mathrm{FV}(P) \\
&(\lambda^* x.P)x \equiv \mathbf{K}\,Px \rightarrow_w P
\end{aligned}$$

If $P \equiv Q\,R$, and x is free in Q or R, then $\lambda^* x.P \equiv \mathbf{S}\,(\lambda^* x.Q)(\lambda^* x.R)$. This case is the inductive step and we may assume, as induction hypotheses, that the theorem holds for Q and R:

$$\begin{aligned}
\mathrm{FV}(\lambda^* x.Q) &= \mathrm{FV}(Q) - \{x\} \\
\mathrm{FV}(\lambda^* x.R) &= \mathrm{FV}(R) - \{x\}
\end{aligned}$$

$$\begin{aligned}
(\lambda^* x.Q)x &\twoheadrightarrow_w Q \\
(\lambda^* x.R)x &\twoheadrightarrow_w R
\end{aligned}$$

We now consider the set of free variables:

$$\begin{aligned}
\mathrm{FV}(\lambda^* x.Q\,R) &= \mathrm{FV}(\,\mathbf{S}\,(\lambda^* x.Q)(\lambda^* x.R)) \\
&= (\mathrm{FV}(Q) - \{x\}) \bigcup (\mathrm{FV}(R) - \{x\}) \\
&= \mathrm{FV}(Q\,R) - \{x\}
\end{aligned}$$

Finally, we consider application:

$$\begin{aligned}
(\lambda^* x.P)x &\equiv \mathbf{S}\,(\lambda^* x.Q)(\lambda^* x.R)x \\
&\to_w (\lambda^* x.Q)x((\lambda^* x.R)x) \\
&\twoheadrightarrow_w Q((\lambda^* x.R)x) \\
&\twoheadrightarrow_w Q\,R
\end{aligned}$$

□

Using $(\lambda^* x.P)x \twoheadrightarrow_w P$, we may derive an analogue of β-reduction for combinatory logic. We also get a strong analogue of α-conversion — changes in the abstraction variable are absolutely insignificant, yielding identical terms.

Theorem 14 *For all combinatory terms P and Q,*

$$\begin{aligned}
(\lambda^* x.P)Q &\twoheadrightarrow_w P[Q/x] \\
\lambda^* x.P &\equiv \lambda^* y.P[y/x] \qquad \textit{if } y \notin \mathrm{FV}(P)
\end{aligned}$$

Proof Both statements are routine structural inductions; the first can also be derived from the previous theorem by a general substitution theorem [1]. □

7.4 The Relation Between λ-Terms and Combinators

The mapping $(\ )_{CL}$ converts a λ-term into a combinator term. It simply applies λ^* recursively to all the abstractions in the λ-term; note that the innermost abstractions are performed first! The inverse mapping, $(\ )_\lambda$, converts a combinator term into a λ-term.

Definition 15 The mappings $(\ )_{CL}$ and $(\ )_\lambda$ are defined recursively as follows:

$$\begin{aligned}
(x)_{CL} &\equiv x \\
(M\,N)_{CL} &\equiv (M)_{CL}(N)_{CL} \\
(\lambda x.M)_{CL} &\equiv \lambda^* x.(M)_{CL}
\end{aligned}$$

$$\begin{aligned}
(x)_\lambda &\equiv x \\
(\,\mathbf{K}\,)_\lambda &\equiv \lambda x\,y.x \\
(\,\mathbf{S}\,)_\lambda &\equiv \lambda x\,y\,z.x\,z(y\,z) \\
(P\,Q)_\lambda &\equiv (P)_\lambda(Q)_\lambda
\end{aligned}$$

Different versions of combinatory abstraction yield different versions of $(\ )_{CL}$; the present one causes exponential blow-up in term size, but it is easy to reason about. Let us abbreviate $(M)_{CL}$ as M_{CL} and $(P)_\lambda$ as P_λ. It is easy to check that $(\ )_{CL}$ and $(\ )_\lambda$ do not add or delete free variables:

$$\mathrm{FV}(M) = \mathrm{FV}(M_{CL}) \qquad \mathrm{FV}(P) = \mathrm{FV}(P_\lambda)$$

Equality is far more problematical. The mappings do give a tidy correspondence between the λ-calculus and combinatory logic, provided we assume the principle of *extensionality*. This asserts that two functions are equal if they return equal results for every argument value. In combinatory logic, extensionality takes the form of a new rule for proving equality:

$$\frac{Px = Qx}{P = Q} \qquad (x \text{ not free in } P \text{ or } Q)$$

In the λ-calculus, extensionality can be expressed by a similar rule or by introducing η-reduction:

$$\lambda x.Mx \to_\eta M \qquad (x \text{ not free in } M)$$

Assuming extensionality, the mappings preserve equality [1]:

$$\begin{array}{rcll} (M_{CL})_\lambda & = & M & \text{in the } \lambda\text{-calculus} \\ (P_\lambda)_{CL} & = & P & \text{in combinatory logic} \\ M = N & \Longleftrightarrow & M_{CL} = N_{CL} & \\ P = Q & \Longleftrightarrow & P_\lambda = Q_\lambda & \end{array}$$

Normal forms and reductions are not preserved. For instance, $\mathbf{S}\ \mathbf{K}$ is a normal form of combinatory logic; no weak reductions apply to it. But the corresponding λ-term is not in normal form:

$$(\mathbf{S}\ \mathbf{K})_\lambda \equiv (\lambda x\, y\, z.x\, z(y\, z))(\lambda x\, y.x) \twoheadrightarrow \lambda y\, z.z$$

There are even combinatory terms in normal form whose corresponding λ-term has no normal form! Even where both terms follow similar reduction sequences, reductions in combinatory logic have much finer granularity than those in the λ-calculus; consider how many steps are required to simulate a β-reduction in combinatory logic.

Normal forms are the outputs of functional programs; surely, they ought to be preserved. Reduction is the process of generating the outputs. Normally we should not worry about this, but lazy evaluation has to deal with infinite outputs that cannot be fully evaluated. Thus, the rate and granularity of reduction is important. Despite the imperfect correspondence between λ-terms and combinators, compilers based upon combinatory logic appear to work. Perhaps the things not preserved are insignificant for computational purposes. More research needs to be done in the operational behaviour of functional programs.

8 Compiling Methods Using Combinators

Combinator abstraction gives us a theoretical basis for removing variables from λ-terms, and will allow efficient graph reduction. But first, we require a mapping from λ-terms to combinators that generates more compact results. Recall that λ^* causes exponential blowup:

$$\lambda^* x\, y.y\, x \equiv \mathbf{S}\,(\,\mathbf{K}\,(\,\mathbf{S}\ \mathbf{I}\,))(\,\mathbf{S}\,(\,\mathbf{K}\ \mathbf{K}\,)\,\mathbf{I}\,)$$

The improved version of combinatory abstraction relies on two new combinators, $\mathbf{B}$ and $\mathbf{C}$, to handle special cases of $\mathbf{S}$:

$$\begin{array}{lcl} \mathbf{B}\, P\, Q\, R & \to_w & P(Q\, R) \\ \mathbf{C}\, P\, Q\, R & \to_w & P\, R\, Q \end{array}$$

Note that $\mathbf{B}\, P\, Q\, R$ yields the function composition of P and Q. Let us call the new abstraction mapping λ^T, after David Turner, its inventor:

$$\begin{array}{lcll} \lambda^T x.x & \equiv & \mathbf{I} & \\ \lambda^T x.P & \equiv & \mathbf{K}\, P & (x \text{ not free in } P) \\ \lambda^T x.P\, x & \equiv & P & (x \text{ not free in } P) \\ \lambda^T x.P\, Q & \equiv & \mathbf{B}\, P(\lambda^T x.Q) & (x \text{ not free in } P) \\ \lambda^T x.P\, Q & \equiv & \mathbf{C}\,(\lambda^T x.P)Q & (x \text{ not free in } Q) \\ \lambda^T x.P\, Q & \equiv & \mathbf{S}\,(\lambda^T x.P)(\lambda^T x.Q) & (x \text{ free in } P \text{ and } Q) \end{array}$$

Although λ^T is a bit more complicated than λ^*, it generates much better code (i.e. combinators). The third case, for $P\, x$, takes advantage of extensionality; note its similarity to η-reduction. The next two cases abstract over $P\, Q$ according to whether or not the abstraction variable is actually free in P or Q. Let us do our example again:

$$\begin{array}{rcl} \lambda^T x\, y.y\, x & \equiv & \lambda^T x.(\lambda^T y.y\, x) \\ & \equiv & \lambda^T x.\, \mathbf{C}\,(\lambda^T y.y)x \\ & \equiv & \lambda^T x.\, \mathbf{C}\ \mathbf{I}\, x \\ & \equiv & \mathbf{C}\ \mathbf{I} \end{array}$$

The size of the generated code has decreased by a factor of four! Here is another example, from Paper 6 of the 1993 Examination. Let us translate the λ-encoding of the ordered pair operator:

$$\begin{array}{rcl} \lambda^T x.\lambda^T y.\lambda^T f.f\, x\, y & \equiv & \lambda^T x.\lambda^T y.\, \mathbf{C}\,(\lambda^T f.f\, x)\, y \\ & \equiv & \lambda^T x.\lambda^T y.\, \mathbf{C}\,(\,\mathbf{C}\,(\lambda^T f.f)\, x)\, y \\ & \equiv & \lambda^T x.\lambda^T y.\, \mathbf{C}\,(\,\mathbf{C}\ \mathbf{I}\ x)\, y \\ & \equiv & \lambda^T x.\, \mathbf{C}\,(\,\mathbf{C}\ \mathbf{I}\ x) \\ & \equiv & \mathbf{B}\ \mathbf{C}\,(\lambda^T x.\, \mathbf{C}\ \mathbf{I}\ x) \\ & \equiv & \mathbf{B}\ \mathbf{C}(\,\mathbf{C}\ \mathbf{I}\,). \end{array}$$

Unfortunately, λ^T can still cause a quadratic blowup in code size; additional primitive combinators should be introduced (See Field and Harrison [5, page 286]. Furthermore, all the constants of the functional language — numbers, arithmetic operators, … — must be taken as primitive combinators.

Introducing more and more primitive combinators makes the code smaller and faster. This leads to the method of *super combinators*, where the set of primitive combinators is extracted from the program itself.

Exercise 12 Show $\mathbf{B}\,P\,\mathbf{I} = P$ using extensionality.

Exercise 13 Verify that $\mathbf{C}\ \mathbf{I}$ behaves like the λ-term $\lambda x\,y.y\,x$ when applied to two arguments.

Exercise 14 What would $\lambda^T x\,y.y\,x$ yield if we did not apply the third case in the definition of λ^T?

8.1 Combinator Terms as Graphs

Consider the ISWIM program

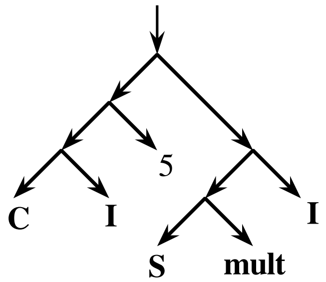
$$\textit{let}\ sqr(n) = n \times n\ \textbf{in}\ sqr(5)$$

Let us translate it to combinators:

$$\begin{aligned}(\lambda^T f.f\,5)(\lambda^T n.\,\mathbf{mult}\,n\,n) &\equiv \mathbf{C}\ \mathbf{I}\,5\,(\,\mathbf{S}\,(\lambda^T n.\,\mathbf{mult}\,n)(\lambda^T n.n))\\ &\equiv \mathbf{C}\ \mathbf{I}\,5\,(\,\mathbf{S}\ \mathbf{mult}\ \mathbf{I}\,)\end{aligned}$$

This is a *closed* term — it contains no free variables (and no bound variables, of course). Therefore it can be evaluated by reducing it to normal form.

Graph reduction works on the combinator term's graph structure. This resembles a binary tree with branching at each application. The graph structure for $\mathbf{C}\ \mathbf{I}\,5\,(\,\mathbf{S}\ \mathbf{mult}\ \mathbf{I}\,)$ is as follows:



Repeated arguments cause sharing in the graph, ensuring that they are never evaluated more than once.

8.2 The Primitive Graph Transformations

Graph reduction deals with terms that contain no variables. Each term, and its subterms, denote constant values. Therefore we may transform the graphs destructively — operands are never copied. The graph is *replaced* by its normal form!

The primitive combinators reduce as shown in Figure 1. The sharing in the reduction for $\mathbf{S}$ is crucial, for it avoids copying R.

We also require graph reduction rules for the built-in functions, such as $\mathbf{mult}$. Because $\mathbf{mult}$ is a strict function, the graph for $\mathbf{mult}\,P\,Q$ can only be reduced after P and Q have been reduced to numeric constants m and n. Then $\mathbf{mult}\,m\,n$ is replaced by the constant whose value is $m \times n$. Graph reduction proceeds by walking down the graph's leftmost branch, seeking something to reduce. If the leftmost symbol is a combinator like $\mathbf{I}$, $\mathbf{K}$, $\mathbf{S}$, $\mathbf{B}$, or $\mathbf{C}$, with the requisite number of operands, then it applies the corresponding transformation. If the leftmost symbol is a strict combinator like $\mathbf{mult}$, then it recursively traverses the operands, attempting to reduce them to numbers.

Figure 2 presents the graph reduction sequence for the ISWIM program

$$\textit{let}\ sqr(n) = n \times n\ \textbf{in}\ sqr(5).$$

The corresponding term reductions are as follows:

$$\begin{aligned}
\mathbf{C}\,\mathbf{I}\,5\,(\mathbf{S}\ \mathbf{mult}\ \mathbf{I}) &\to \mathbf{I}\,(\mathbf{S}\ \mathbf{mult}\ \mathbf{I})\,5\\
&\to \mathbf{S}\ \mathbf{mult}\ \mathbf{I}\,5\\
&\to \mathbf{mult}\,5\,(\mathbf{I}\,5)\\
&\to \mathbf{mult}\,5\,5\\
&\to 25
\end{aligned}$$

Clearly visible in the graphs, but not in the terms, is that the two copies of 5 are shared. If, instead of 5, the argument of sqr had been a large combinatory term P compiled from the program, then P would have been evaluated only once. Graph reduction can also discard terms by the rule $\mathbf{K}\,P\,Q \to_w P$; here Q is never evaluated.

8.3 Booleans and Pairing

The λ-calculus encodings of ordered pairs, Church numerals and so forth work with combinators, but are impractical to use for compiling a functional language. New combinators and new reductions are introduced instead.

With lazy evaluation, if-then-else can be treated like a function, with the two reductions

$$\begin{aligned}
\mathbf{if}\ \mathbf{true}\,P\,Q &\to_w P\\
\mathbf{if}\ \mathbf{false}\,P\,Q &\to_w Q.
\end{aligned}$$

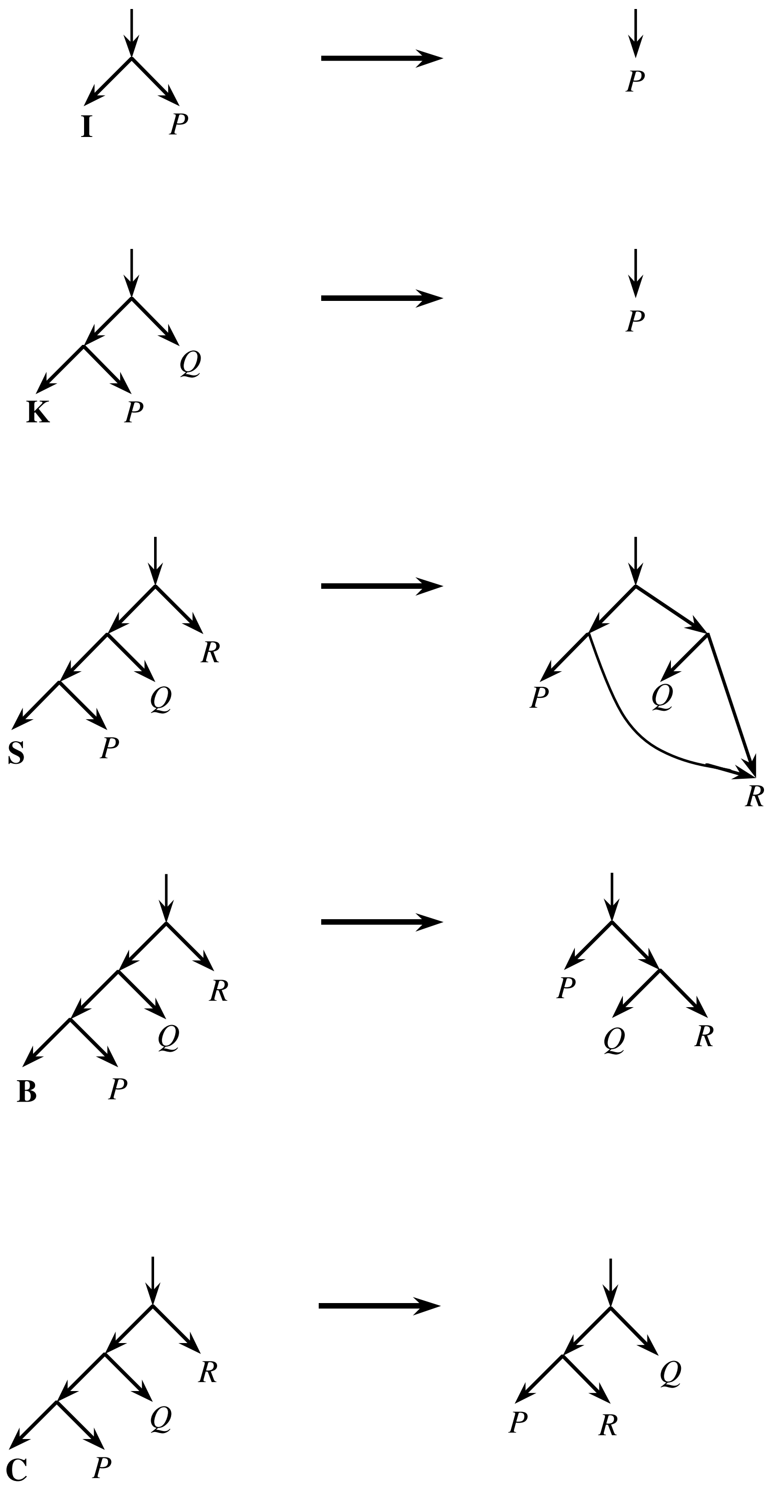


Figure 1: Graph reduction for combinators

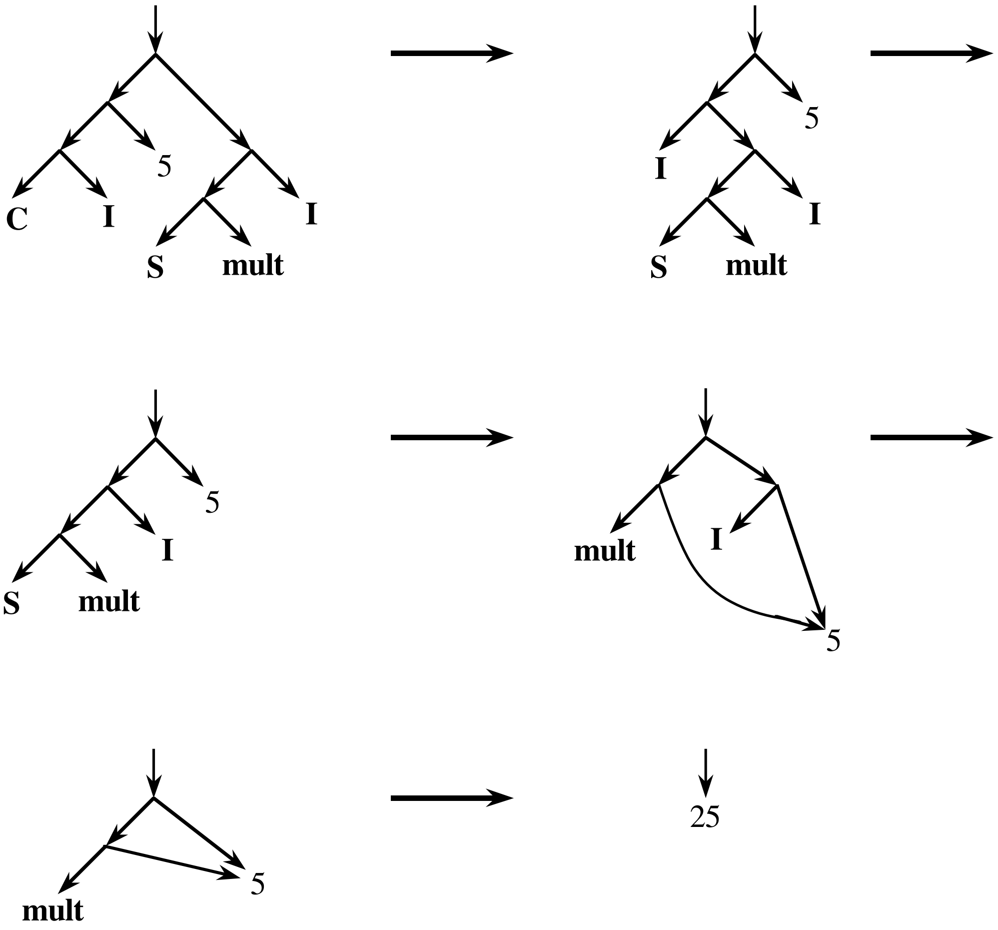


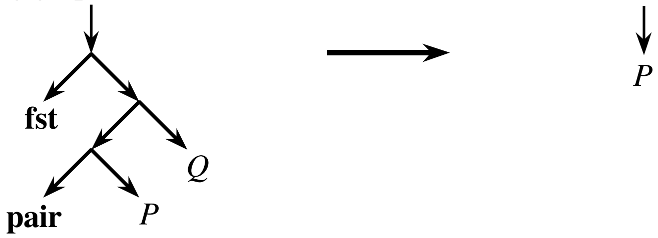
Figure 2: A graph reduction sequence

These reductions discard P or Q if it is not required; there is no need for tricks to delay their evaluation. The first reduction operates on graphs as shown.

Pairing is also lazy, as it is in the λ-calculus; we introduce the reductions

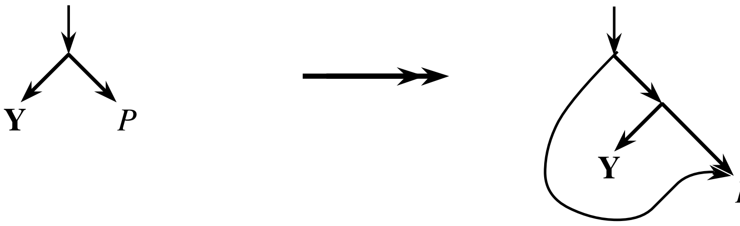
$$\begin{aligned}\mathbf{fst}\,(\,\mathbf{pair}\,P\,Q) &\to_w P\\ \mathbf{snd}\,(\,\mathbf{pair}\,P\,Q) &\to_w Q.\end{aligned}$$

The corresponding graph reductions should be obvious:

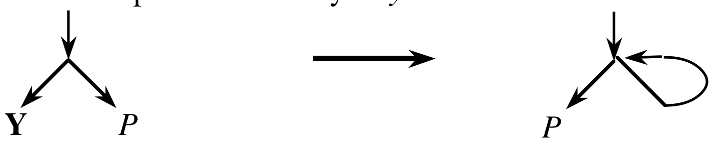


8.4 Recursion: Cyclic Graphs

Translating $\mathbf{Y}$ into combinator form will work, yielding a mult-step reduction resembling[4]



This is grossly inefficient; $\mathbf{Y}$ must repeat its work at every recursive invocation! Instead, take $\mathbf{Y}$ as a primitive combinator satisfying $\mathbf{Y}\,P \to_w P(\mathbf{Y}\,P)$ and adopt a graph reduction rule that replaces the $\mathbf{Y}$ by a *cycle*:



Since P is never copied, reductions that occur in it yield permanent simplifications — they are not repeated when the function is entered recursively.

To illustrate this, consider the ISWIM program

$$\mathit{letrec}\ \mathit{from}(n) = \ \mathbf{pair}\,n(\mathit{from}(1+n))\ \mathbf{in}\ \mathit{from}(1).$$

[4]The picture is an over-simplification; recall that we do not have $\mathbf{Y}\,P \twoheadrightarrow P(\mathbf{Y}\,P)$!

The result should be the infinite list $(1,(2,(3,\ldots)))$. We translate *from* into combinators, starting with

$$\mathbf{Y}\;(\lambda^T f\,n.\,\mathbf{pair}\,n(f(\,\mathbf{add}\,1\,n))$$

and obtain (verify this)

$$\mathbf{Y}\,(\,\mathbf{B}\,(\,\mathbf{S}\;\mathbf{pair}\,)(\,\mathbf{C}\;\mathbf{B}\,(\,\mathbf{add}\,1)))$$

Figures 3 and 4 give the graph reductions. A cyclic node, labelled θ, quickly appears. Its rather tortuous transformations generate a recursive occurrence of *from* deeper in the graph. The series of reductions presumes that the environment is demanding evaluation of the result; in lazy evaluation, nothing happens until it is forced to happen.

Graph reduction will leave the term $\mathbf{add}\,1\,1$ unevaluated until something demands its value; the result of $from(1)$ is really $(1,(1+1,(1+1+1,\ldots)))$. Graph reduction works a bit like macro expansion. Non-recursive function calls get expanded once and for all the first time they are encountered; thus, programmers are free to define lots of simple functions in order to aid readability. Similarly, constant expressions are evaluated once and for all when they are first encountered. Although this behaviour avoids wasteful recomputation, it can cause the graph to grow and grow, consuming all the store — a *space leak*. The displayed graph reduction illustrates how this could happen.

Exercise 15 Translate $\mathbf{Y}$ to combinators and do some steps of the reduction of $\mathbf{Y}\;P$.

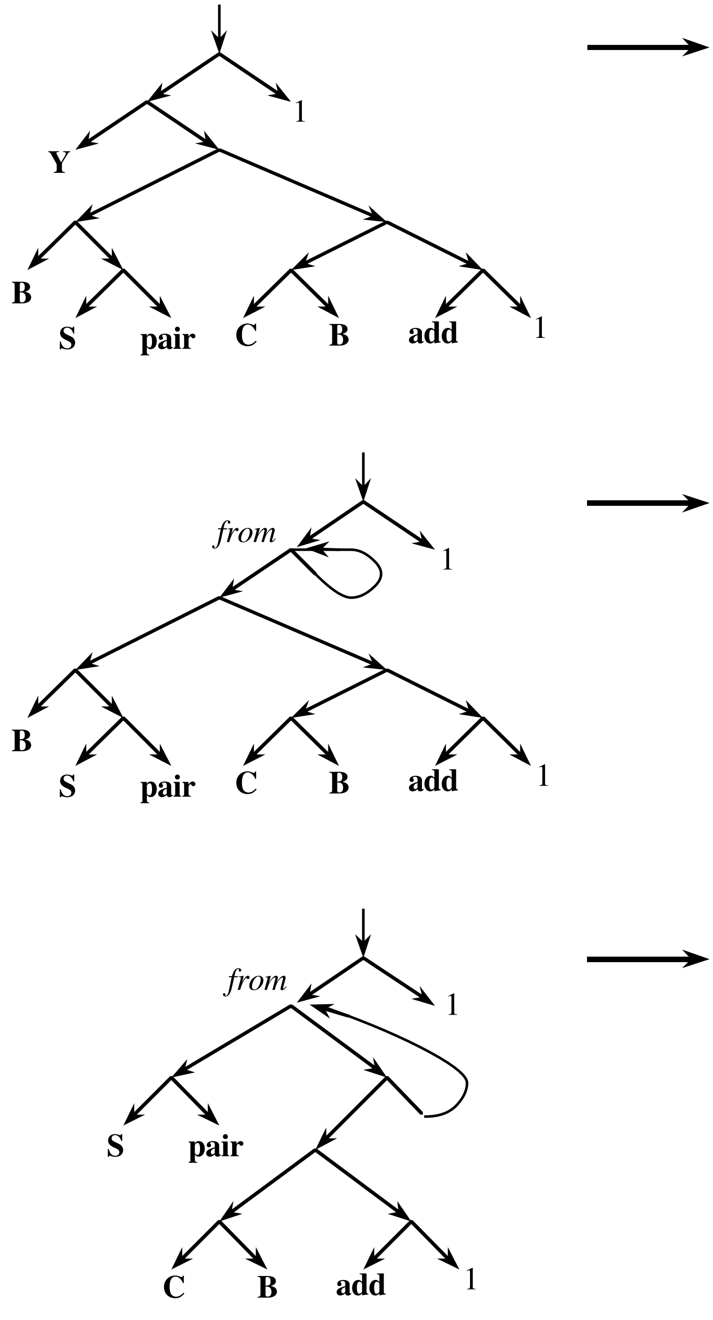


Figure 3: Reductions involving recursion

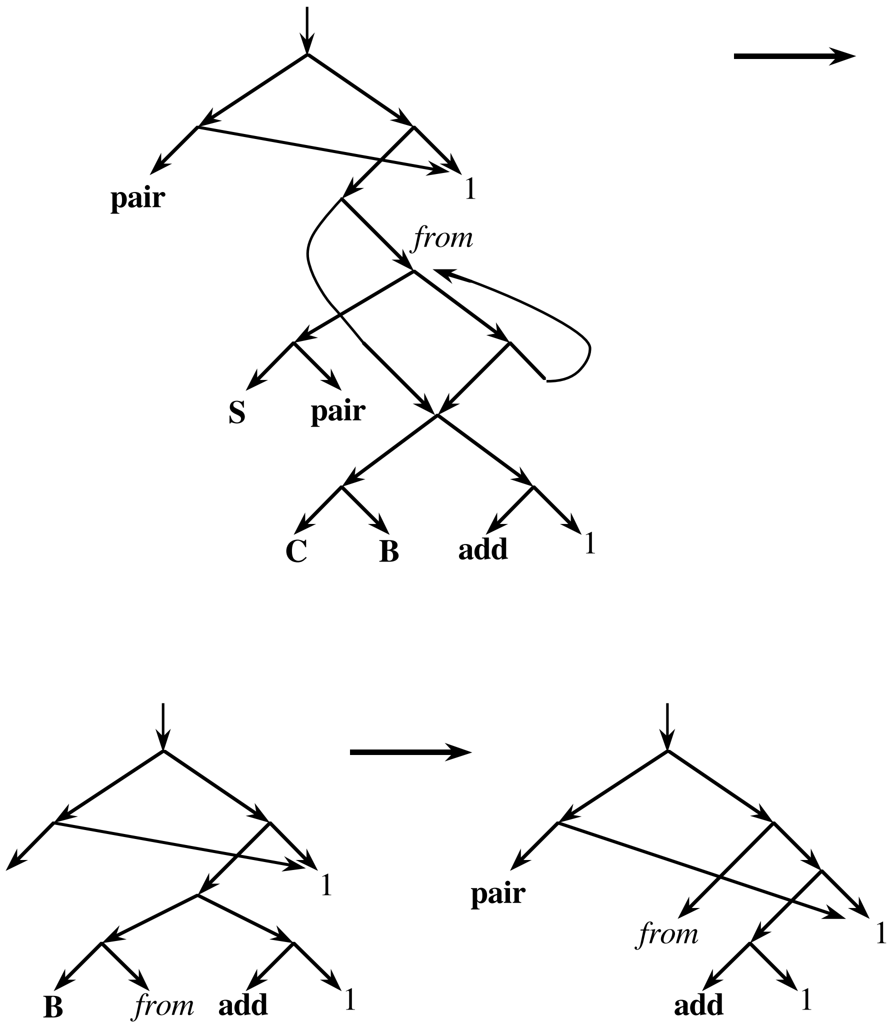


Figure 4: Reductions involving recursion (continued)

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[3] W. H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[4] G. Cousineau and G. Huet. The CAML primer. Technical report, INRIA, Rocquencourt, France, 1990.

[5] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[6] Michael J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall, 1988.

[7] J. Roger Hindley and Jonathon P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge, 1986.

[8] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, January 1964.

[9] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[10] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.