) allow it

function
nment *a*.
in inter-
ning lan-
e, which

's for the
cribes an
finitional
be inter-
972] pre-
higher-
ler func-
dered in

has been
ve auto-

Programming Languages;
Concepts & Constructs,
Ravi Sethi, 2nd Edition
Addison-Wesley 1996
Out-of-print

14 Static Types and the Lambda Calculus

Church was struck by certain similarities between his new concept and that used in Whitehead and Russell [1925] for the class of all x's such that $f(x)$; to wit, $\hat{x}f(x)$. Because the new concept differed quite appreciably from class membership, Church moved the caret down from over the x to the line just to the left of x; specifically, $\wedge xf(x)$. Later, for reasons of typography, an appendage was added to the caret to produce a lambda; the result was $\lambda xf(x)$.

–*Rosser [1984], from an after-dinner talk on the history of the lambda calculus in which he presented "a still shorter and more perspicuous" proof of the Church-Rosser theorem.*

Static or compile-time type checking anticipates run-time behavior, so it is reasonable to study types at the end of this book, after the dynamic semantics of expressions is understood. Only then can we hope to make sense out of claims like "strong typing prevents run-time errors." It is easier to study error prevention if we know what an error is.

Claims about error prevention have to be worded carefully because static types prevent only certain kinds of run-time errors. At compile time, we can detect an attempt to add procedures instead of numbers or an attempt to use a pointer instead of a record, but we cannot always detect an attempt to divide by zero or an attempt to use an out-of-bounds array index.

The notion of the dynamic state of a computation, central to imperative programming, has little to do with static types, which are a property of the program text. Type checking of an assignment

$$i := i + 1$$

deals only with the types of the left and right sides and not with the values of variables. Types can therefore be studied by concentrating on expressions and functional languages. Statements can be checked by giving them a special type **void**. For example, the type-checking rule for conditional statements

$$S \ ::= \ \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2$$

can be expressed as follows:

> The **if-then-else** construct is built up of an expression E of type **bool** for boolean, and statements S_1 and S_2 of type **void**; the result S has type **void**.

This rule treats the **if-then-else** constructor as if it were an operator in an expression.

Functional languages can themselves be reduced to smaller core languages that are more convenient for the study of types.

The Lambda Calculus

the lambda calculus is a vehicle for studying languages

The small syntax of the lambda calculus makes it a convenient vehicle for studying types in programming languages. The pure lambda calculus has just three constructs: variables, function application, and function creation. Nevertheless, it has had a profound influence on the design and analysis of programming languages. Its surprising richness comes from the freedom to create and apply functions, especially higher-order functions of functions.

The lambda calculus gets its name from the Greek letter lambda, λ. The notation $\lambda x.\ M$ is used for a function with parameter x and body M. Thus, $\lambda x.\ x * x$ is a function that maps 5 to $5 * 5$. Functions are written next to their arguments, so $f\ a$ is the application of function f to argument a, as in $\sin \theta$ or $\log n$. In

$$(\lambda x.\ x * x)\ 5$$

function $\lambda x.\ x{*}x$ is applied to 5. Formulas like $(\lambda x.\ x{*}x)\ 5$ are called terms.

Church [1941] introduced the pure lambda calculus in the 1930s to study computation with functions. He was interested in the general properties of functions, independently of any particular problem area. The integer 5 and the multiplication operator $*$ belong to arithmetic and are not part of the pure calculus.

A grammar for *terms* in the *pure lambda calculus* is:

$$M \ ::= \ x \mid (\, M_1\ M_2 \,) \mid (\, \lambda\, x.\, M \,)$$

beta-e
provides a
o

We use letters f, x, y, z for variables and M, N, P, Q for terms. A term is either a *variable* x, an *application* $(M\ N)$ of function M to N, or an *abstraction* $(\lambda x.\ M)$. A constant c can represent values like integers and operations on data structures like lists. That is, c can stand for basic constants like **true** and **nil** as well as constant functions like + and *head*.

The lambda calculi are therefore a family of languages for computation with functions. Members of the family are obtained by choosing a set of constants. In informal usage, "the lambda calculus" refers to any member of this family.

The progression in this chapter is as follows:

- The pure lambda calculus is untyped. Functions can be applied freely; it even makes sense to write $(x\ x)$, where x is applied to itself. In formulating a notion of computation for the pure calculus, we look at scope, parameter passing, and evaluation strategies.
- A functional programming language is essentially a lambda calculus with appropriate constants. This view will be supported by relating a fragment of ML to a lambda calculus.
- The typed lambda calculus associates a type with each term.
- Finally, we consider a lambda calculus with polymorphic types that has been used to study types in ML.

14.1 EQUALITY OF PURE LAMBDA TERMS

beta-equality provides a notion of value

This chapter opened with an informal description of the pure lambda calculus: x is a variable, $(M\ N)$ represents the application of function M to N, and the abstraction $(\lambda x.\ M)$ represents a function with parameter x and body M. Now it is time to be more precise about the roles of abstraction and application.

This section develops an equality relation on terms, called *beta-equality* for historical reasons. We write $M =_\beta N$ if M and N are beta-equal. Informally, if $M =_\beta N$, then M and N must have the "same value."

Beta-equality deals with the result of applying an abstraction $(\lambda x.\ M)$ to an argument N. In other words, beta-equality deals with the notions of function call and parameter passing in programming languages. An abstraction corresponds to a function definition, and an application to a function call. Suppose that function *square* is defined by

fun $square(x)\ =\ x * x;$

The function call $square(5)$ is evaluated by substituting 5 for x in the body $x*x$. In the terminology of this section, $square(5)\ =_\beta\ 5 * 5$.

Syntactic Conventions

The following abbreviations make terms more readable:

- Parentheses may be dropped from $(M\ N)$ and $(\lambda x.\ M)$. In the absence of parentheses, function application groups from left to right. Thus, $x\ y\ z$ abbreviates $((x\ y)\ z)$, and the parentheses in $x\ (y\ z)$ are necessary to ensure that x is applied to $(y\ z)$. Function application has higher precedence than abstraction, so $\lambda x.\ x\ z$ abbreviates $(\lambda x.\ (x\ z))$.
- A sequence of consecutive abstractions, as in $\lambda x.\ \lambda y.\ \lambda z.\ M$, can be written with a single lambda, as in $\lambda xyz.\ M$. Thus, $\lambda xy.\ x$ abbreviates $\lambda x.\ \lambda y.\ x$.

The following terms will be used within the examples in this chapter:

$$I = \lambda x.\ x$$
$$K = \lambda xy.\ x$$
$$S = \lambda xyz.\ (x\ z)\ (y\ z)$$

Here, S could have been written with fewer parentheses as $\lambda xyz.\ x\ z\ (y\ z)$. Its full form is

$$S \;=\; (\lambda x.\ (\lambda y.\ (\lambda z.\ ((x\ z)\ (y\ z)))))$$

A pure lambda term without free variables is called a *closed term*, or *combinator*.

Free and Bound Variables

care is needed with free and bound variables

Abstractions of the form $\lambda x.\ M$ are also referred to as bindings because they constrain the role of x in $\lambda x.\ M$. Variable x is said to be *bound* in $\lambda x.\ M$. The set *free*(M) of *free variables* of M, the variables that appear unbound in M, is given by the following syntax-directed rules:

$$\mathit{free}(x) = \{x\}$$
$$\mathit{free}(M\ N) = \mathit{free}(M) \cup \mathit{free}(N)$$
$$\mathit{free}(\lambda x.\ M) = \mathit{free}(M) - \{x\}$$

In words, variable x is free in the term x. A variable is free in $M\ N$ if it is either free in M or free in N. With the exception of x, all other free variables of M are free in $\lambda x.\ M$.

Free variables have been a trouble spot in both programming languages and the lambda calculus, so we take a closer look at them. For example, z is a free variable of the following term because it is free in the subterm $\lambda y.\ z$:

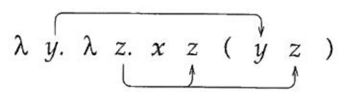
a nam occurs i variable i bound

$(\lambda y.\ z)\ (\lambda z.\ z)$

We now introduce a way of distinguishing between this first occurrence of z and the other ones in the subterm $(\lambda z.\ z)$.

The occurrence of x to the right of the λ in $\lambda x.\ M$ is called a *binding occurrence* or simply a *binding* of x. All occurrences of x in $\lambda x.\ M$ are *bound* within the *scope* of this binding. All unbound occurrences of a variable in a term are *free*. Each occurrence of a variable is either free or bound; it cannot be both.

The only occurrence of x in $\lambda x.\ y$ is bound within its own scope. The lines in the following diagram go from a binding to a bound occurrence of y, and from a binding to bound occurrences of z.



In this diagram, the occurrence of x is free because it is not within the scope of any binding within the term.

Substitution

a name clash occurs if a free variable in N is bound in M

The result of applying an abstraction $(\lambda x.\ M)$ to an argument N will be formalized by "substituting" N for x in M. Informally, N replaces all free occurrences of x in M. A definition of substitution is rather tricky, as evidenced by a long history of inadequate definitions. The following definition first tackles the easy case, which suffices for most examples. A more precise syntax-directed definition appears in Section 14.2.

The *substitution* of a term N for a variable x in M is written as $\{N/x\}\ M$ and is defined as follows:

1. Suppose that the free variables of N have no bound occurrences in M. Then, the term $\{N/x\}\ M$ is formed by replacing all free occurrences of x in M by N.
2. Otherwise, suppose that variable y is free in N and bound in M. Consistently replace the binding and corresponding bound occurrences of y in M by some fresh variable z.[1] Repeat the renaming of bound variables in M until case 1 applies, then proceed as in case 1.

Example 14.1 In each of the following cases, M has no bound occurrences, so N replaces all occurrences of x in M to form $\{N/x\}\ M$:

[1] The syntax of λ-terms can be made independent of the spellings of variables by using positional indexes, as in de Bruijn [1972]. Positional indexes eliminate the need for renaming.

$$\begin{aligned}
\{u/x\}\ x &= u \\
\{u/x\}\ (x\ x) &= (u\ u) \\
\{u/x\}\ (x\ y) &= (u\ y) \\
\{u/x\}\ (x\ u) &= (u\ u) \\
\{(\lambda x.\ x)/x\}\ x &= (\lambda x.\ x)
\end{aligned}$$

In the following cases, M has no free occurrences of x, so $\{N/x\}\ M$ is M itself:

$$\begin{aligned}
\{u/x\}\ y &= y \\
\{u/x\}\ (y\ z) &= (y\ z) \\
\{u/x\}\ (\lambda y.\ y) &= (\lambda y.\ y) \\
\{u/x\}\ (\lambda x.\ x) &= (\lambda x.\ x) \\
\{(\lambda x.\ x)/x\}\ y &= y
\end{aligned}$$

In the following cases, free variable u in N has bound occurrences in M, so $\{N/x\}\ M$ is formed by first renaming the bound occurrences of u in M:

$$\begin{aligned}
\{u/x\}\ (\lambda u.\ x) &= \{u/x\}\ (\lambda z.\ x) = (\lambda z.\ u) \\
\{u/x\}\ (\lambda u.\ u) &= \{u/x\}\ (\lambda z.\ z) = (\lambda z.\ z)
\end{aligned}$$

□

Beta-Equality

beta-equality is a congruence: equals can be replaced by equals

The key axiom of beta-equality is as follows:

$$(\lambda x.\ M)\ N \ =_\beta\ \{N/x\}\ M \qquad (\beta \text{ axiom})$$

Thus, $(\lambda x.\ x)\ u =_\beta u$ and $(\lambda x.\ y)\ u =_\beta y$.

The following axiom allows bound variables to be systematically renamed:

$$(\lambda x.\ M)\ =_\beta\ \lambda z.\ \{z/x\}\ M \quad \text{provided that } z \text{ is not free in } M \qquad (\alpha \text{ axiom})$$

Thus, $\lambda x.\ x =_\beta \lambda y.\ y$ and $\lambda xy.\ x =_\beta \lambda uv.\ u$.

The remaining rules for beta-equality formalize general properties of equalities (see Fig. 14.1). Each of the following must be true with any notion of equality on terms:

Idempotence. A term M equals itself.

Commutativity. If M equals N, then, conversely, N must equal M.

Transitivity. If M equals N and N equals P, then M equals P.[2]

[2] An *equivalence* relation on a set S is any binary relation that has the idempotence, commutativity, and transitivity properties.

$$(\lambda x.\, M) \;=_\beta\; \lambda z.\, \{z/x\}\, M \quad \text{provided that } z \text{ is not free in } M \qquad (\alpha \text{ axiom})$$

$$(\lambda x.\, M)\, N \;=_\beta\; \{N/x\}\, M \qquad (\beta \text{ axiom})$$

$$M =_\beta M \qquad (\text{idempotence axiom})$$

$$\frac{M =_\beta N}{N =_\beta M} \qquad (\text{commutativity rule})$$

$$\frac{M =_\beta N \qquad N =_\beta P}{M =_\beta P} \qquad (\text{transitivity rule})$$

$$\frac{M =_\beta M' \qquad N =_\beta N'}{M\, N =_\beta M'\, N'} \qquad (\text{congruence rule})$$

$$\frac{M =_\beta M'}{\lambda x.\, M =_\beta \lambda x.\, M'} \qquad (\text{congruence rule})$$

Figure 14.1 Axioms and rules for beta-equality.

The replacement of equals for equals is formalized by the two *congruence* rules in Fig. 14.1. The first rule can be read as follows:

If $M =_\beta M'$ and $N =_\beta N'$, then $M\, N =_\beta M'\, N'$.

Furthermore,

If $M =_\beta M'$, then $\lambda x.\, M =_\beta \lambda x.\, M'$.

Example 14.2 The axioms and rules for beta-equality will be applied to show that

$$SII \;=_\beta\; \lambda z.\, z\, z$$

Application groups from left to right, so SII is written for $(SI)\,I$. S is reserved for $\lambda xyz.\, xz\,(yz)$ and I is reserved for $\lambda x.\, x$, so SII is

$$(\lambda xyz.\, xz(yz))\;(\lambda x.\, x)\;(\lambda x.\, x)$$

This example concentrates on the α and β axioms; subterms to which these axioms apply will be highlighted by underlining them. We begin by using the α axiom to rename bound variables, for clarity.

$$(\lambda xyz.\ xz(yz))\ \underline{(\lambda x.\ x)}\ (\lambda x.\ x) \quad =_\beta \quad (\lambda xyz.\ xz(yz))\ (\lambda u.\ u)\ (\lambda x.\ x)$$

The second copy of $\lambda x.\ x$ is now renamed into $\lambda v.\ v$:

$$(\lambda xyz.\ xz(yz))\ (\lambda u.\ u)\ \underline{(\lambda x.\ x)} \quad =_\beta \quad (\lambda xyz.\ xz(yz))\ (\lambda u.\ u)\ (\lambda v.\ v)$$

The resulting term on the right side of this equality has only one binding for each variable.

The first change in the structure of the term is due to the β axiom:

$$\underline{(\lambda xyz.\ xz(yz))\ (\lambda u.\ u)}\ (\lambda v.\ v) \quad =_\beta \quad (\lambda yz.\ (\lambda u.\ u)\ z\ (yz))\ (\lambda v.\ v)$$

The right side is formed by substituting $(\lambda u.\ u)$ for x in $(\lambda yz.\ xz(yz))$. Three more applications of the β axiom are needed to complete the proof of $SII =_\beta \lambda z.\ zz$:

$$\begin{aligned} SII &=_\beta (\lambda yz.\ \underline{(\lambda u.\ u)\ z}\ (yz))\ (\lambda v.\ v) \\ &=_\beta \underline{(\lambda yz.\ z\ (yz))\ (\lambda v.\ v)} \\ &=_\beta \lambda z.\ z\ (\ \underline{(\lambda v.\ v)\ z}\ ) \\ &=_\beta \lambda z.\ zz \end{aligned}$$

□

14.2 SUBSTITUTION REVISITED

The description of substitution on page 551 can be summarized as follows. If the free variables of N have no bound occurrences in M, then $\{N/x\}\ M$ is formed by replacing all free occurrences of x in M by N; otherwise, bound variables in M are renamed until this rule applies. This section contains a syntax-directed definition of substitution.

The next example motivates the renaming of bound variables during substitution.

Example 14.3 Consider the term $\lambda xy.\ minus\ x\ y$. Formally, *minus* is just a variable; intuitively, *minus* $x\ y$ stands for the subtraction $x - y$. This example studies the term

$$(\lambda uv.\ minus\ u\ v)\ v\ u$$

Since bound variables can be renamed, we can rewrite this term as

$$(\lambda xy.\ \mathit{minus}\ x\ y)\ v\ u$$

Two applications of the β axiom from Fig. 14.1 yield

$$\begin{aligned}(\lambda xy.\ \mathit{minus}\ x\ y)\ v\ u\ &=_\beta\ (\lambda y.\ \mathit{minus}\ v\ y)\ u\\ &=_\beta\ \mathit{minus}\ v\ u\end{aligned}$$

The original term therefore satisfies the equality

$$(\lambda uv.\ \mathit{minus}\ u\ v)\ v\ u\ =_\beta\ \mathit{minus}\ v\ u$$

The naive approach of implementing $\{N/x\}\,M$ by putting N in place of the free occurrences of x in M incorrectly suggests the following equality:

$$\{v/u\}(\lambda v.\ \mathit{minus}\ u\ v)\ \ ?=\ \ \lambda v.\ \mathit{minus}\ v\ v$$

The correct result is obtained if the bound variable v is renamed:

$$\{v/u\}(\lambda z.\ \mathit{minus}\ u\ z)\ =\ \lambda z.\ \mathit{minus}\ v\ z$$ □

The *substitution* of N for x in M, written $\{N/x\}M$, is defined by the syntax-directed rules in Fig. 14.2. We use P and Q to refer to subterms of M.

In words, the substitution of N for x in x yields N. If y is a variable different from x, then y is left unchanged by the substitution of N for x in y.

The substitution of N for x distributes across an application $(P\ Q)$; that is, we substitute N for x in both P and Q.

The tricky case occurs when N is substituted for x in an abstraction:

$$\begin{array}{rcll}
\{N/x\}\ x & = & N & \\
\{N/x\}\ y & = & y & y \neq x\\
\{N/x\}\ (P\ Q) & = & \{N/x\}\ P\ \{N/x\}\ Q & \\
\{N/x\}\ (\lambda x.\ P) & = & \lambda x.\ P & \\
\{N/x\}\ (\lambda y.\ P) & = & \lambda y.\ \{N/x\}\ P & y \neq x, y \notin \mathit{free}(N)\\
\{N/x\}(\lambda y.\ P) & = & \lambda z.\ \{N/x\}\{z/y\}P & y \neq x, z \notin \mathit{free}(N), z \notin \mathit{free}(P)
\end{array}$$

Figure 14.2 Rules for substitution.

1. Since x is not free in $\lambda x.\ P$, the term $\lambda x.\ P$ itself is the result of substituting N for the free occurrences of x in it.
2. Consider the substitution of N for x in $\lambda y.\ P$, with y different from x. If y is not free in N, then the result is $\lambda y.\ \{N/x\}P$.
3. Finally, suppose that y is free in N. Bound variables can be renamed, so we rename y in $\lambda y.\ P$ by a fresh variable z. Of course, z must be a variable that is not free in N and not free in P. The renaming of y in $\lambda y.\ P$ yields $\lambda z.\ \{z/y\}P$. The substitution of N for x in $\lambda z.\ \{z/y\}P$ yields $\lambda z.\ \{N/x\}\{z/y\}\ P$.

Example 14.4 The reader is urged to verify the following equalities:

$$\begin{aligned}
\{u/x\}\ (\lambda u.\ x) &= (\lambda z.\ u)\\
\{u/x\}\ (\lambda u.\ u) &= (\lambda z.\ z)\\
\{u/x\}\ (\lambda y.\ x) &= (\lambda y.\ u)\\
\{u/x\}\ (\lambda y.\ u) &= (\lambda y.\ u)
\end{aligned}$$

The first equality deals with the substitution of u for x in $(\lambda u.\ x)$. Blind substitution of u for x leads to the wrong answer $(\lambda u.\ u)$. □

14.3 COMPUTATION WITH PURE LAMBDA TERMS

reductions can be applied in any order

Computation in the lambda calculus is symbolic. A term is "reduced" into as simple a form as possible. Among the two beta-equal terms

$$(\lambda x.\ M)\ N \ =_\beta\ \{N/x\}\ M$$

the right side $\{N/x\}\ M$ is considered to be simpler than $(\lambda x.\ M)\ N$. Among

$$(\lambda xy.\ x)\ u\ v \ =_\beta\ (\lambda y.\ u)\ v \ =_\beta\ u$$

u is simpler than $(\lambda y.\ u)\ v$, which in turn is simpler than $(\lambda xy.\ x)\ u\ v$.

These observations motivate a rewriting rule called β-*reduction.* An additional rule, called α-*conversion,* renames bound variables.

$$\begin{array}{llll}
(\lambda x.\ M)\ N & \underset{\beta}{\Rightarrow} & \{N/x\}\ M & & (\beta\text{-reduction})\\
\lambda x.\ M & \underset{\alpha}{\Rightarrow} & \lambda y.\ \{y/x\}\ M & y \text{ not free in } M & (\alpha\text{-conversion})
\end{array}$$

Now $(\lambda xy.\ x)\ u \underset{\beta}{\Rightarrow} (\lambda y.\ u)$ and $(\lambda y.\ u)\ v \underset{\beta}{\Rightarrow} u$.

This section examines β-reduction. A fundamental result of the lambda calculus implies that the result of a computation is independent of the order in which β-reductions are applied.

Reductions

We write $P \underset{\beta}{\Rightarrow} Q$ if a subterm of P is β-reduced to create Q. A subterm of the form $(\lambda x.\ M)\ N$ is called a *redex*, for "reduction expression." Thus, if $P \underset{\beta}{\Rightarrow} Q$ then P has a redex $(\lambda x.\ M)\ N$ that is replaced by $\{N/x\}\ M$ to create Q. Similarly, we write $P \underset{\alpha}{\Rightarrow} Q$ if α-conversion of a subterm of P yields Q.

a term is in normal form if it cannot be reduced

A *reduction* is any sequence of β-reductions and α-conversions. A term that cannot be β-reduced is said to be in β-*normal form*, or simply in *normal form*. The term $\lambda z.\ zz$ is in normal form because none of its subterms is a redex of the form $(\lambda x.\ M)\ N$.

The following example considers alternative reductions that start with SII and end with the normal form $\lambda z.\ zz$.

Example 14.5 In Fig. 14.3, redexes are underlined and arrows represent β-reductions. Some of the lines are dashed for clarity.

The starting term at the top of the figure is SII. Again, S is $\lambda xyz.\ xz\ (yz)$ and I is $\lambda x.\ x$, so the starting term is

$$(\lambda xyz.\ xz\ (yz))\ \ (\lambda x.\ x)\ \ (\lambda x.\ x)$$

Upon β-reduction of the only redex in this term, we get

$$(\lambda yz.\ (\lambda x.\ x)\ z\ (yz))\ (\lambda x.\ x)$$

This term has two redexes. The entire term is a redex, and so is the subterm $(\lambda x.\ x)\ z$. The following reduction begins by reducing the inner redex:

$$\begin{tikzcd}
 & SII = \underline{(\lambda xyz.\ xz(yz))\ (\lambda x.\ x)}(\lambda x.\ x) \arrow[d] & & \\
 & \underline{(\lambda yz.\ \underline{(\lambda x.\ x)\ z}(yz))\ (\lambda x.\ x)} \arrow[dl] \arrow[dr, dashed] & & \\
\underline{(\lambda yz.\ z\ (yz))\ (\lambda x.\ x)} \arrow[dr] & & \lambda z.\underline{(\lambda x.\ x)\ z}\ \underline{((\lambda x.\ x)\ z)} \arrow[dl] \arrow[dr, dashed] & \\
 & \lambda z.\ z\ (\underline{(\lambda x.\ x)\ z}) \arrow[dr] & & \lambda z.\underline{(\lambda x.\ x)\ z}z \arrow[dl] \\
 & & \lambda z.\ z\ z &
\end{tikzcd}$$

Figure 14.3 Alternative reductions from SII to $\lambda z.\ zz$.

$$\begin{aligned}(\lambda yz.\ \underline{(\lambda x.\ x)\ z}\ (yz))\ (\lambda x.\ x) &\underset{\beta}{\Rightarrow} \underline{(\lambda yz.\ z\ (yz))\ (\lambda x.\ x)}\\ &\underset{\beta}{\Rightarrow} \lambda z.\ z\ (\ \underline{(\lambda x.\ x)\ z}\ )\\ &\underset{\beta}{\Rightarrow} \lambda z.\ z\ z\end{aligned}$$

Each path in Fig. 14.3 represents a reduction from *SII* to $\lambda z.\ zz$. □

Nonterminating Reductions

It is possible for a reduction to continue forever, without reaching a normal form. Reductions starting with

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

do not terminate. For clarity, let us α-convert the first $(\lambda x.\ xx)$ into $(\lambda y.\ yy)$. Then

$$(\lambda y.\ yy)(\lambda x.\ xx) \underset{\beta}{\Rightarrow} (\lambda x.\ xx)(\lambda x.\ xx)$$

and we are back where we started.

The first few steps of a more "useful" nonterminating computation appear in Fig. 14.4. The computation begins with *Yf*, where *Y* is a special term such that *Yf* reduces to $f(Yf)$. *Y* is an example of a "fixed-point combinator."

A combinator is a pure lambda term without free variables. A combinator *M* is called a *fixed-point combinator* if $Mf =_\beta f\ (Mf)$. The significance of fixed-point combinators is explored in Section 14.4, where fixed-point combinators will be used to set up recursions.

The Church-Rosser Theorem

The result "normal forms are unique, if they exist," applies to reductions that terminate in normal forms. A stronger result, called the *Church-Rosser theorem*, applies to all reductions, even nonterminating ones. One form of this theorem is illustrated in Fig. 14.5. For all starting terms *M*, suppose that one sequence

$$\begin{aligned}Yf &= (\lambda f.\ (\lambda x.\ f(xx))\ (\lambda x.\ f(xx)))\ f\\ &\underset{\beta}{\Rightarrow} (\lambda x.\ f(xx))\ (\lambda x.\ f(xx))\\ &\underset{\beta}{\Rightarrow} f\ ((\lambda x.\ f(xx))\ (\lambda x.\ f(xx)))\\ &= f\ (Yf)\end{aligned}$$

Figure 14.4 The term *Yf* β-reduces to $f\ (Yf)$.

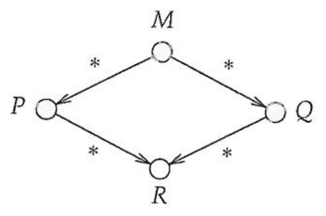


Figure 14.5 If M reduces to P and to Q, then both can reach some common R.

of reductions takes M to P and that another sequence takes M to Q. Then we can always find some common term R, such that P can reduce to R and Q can also reduce to R. The filled circles next to M, P, and Q emphasize that the result holds for all such M, P, and Q. The open circle at R emphasizes that only some terms R are reachable from both P and Q.

The following statement of the Church-Rosser theorem uses the notation $\stackrel{*}{\Rightarrow}$ for a sequence of zero or more α-conversions and β-reductions. We write $P \Rightarrow Q$ if $P \underset{\alpha}{\Rightarrow} Q$ or if $P \underset{\beta}{\Rightarrow} Q$. Thus, $P \stackrel{*}{\Rightarrow} Q$ means that for some terms $P_0, P_1, \ldots, P_k$, where $k \geq 0$,

$$P = P_0 \Rightarrow P_1 \Rightarrow \cdots \Rightarrow P_k = Q$$

Note that $P \stackrel{*}{\Rightarrow} P$ holds; this case corresponds to $k = 0$.

Church-Rosser Theorem. For all pure λ-terms M, P, and Q, if $M \stackrel{*}{\Rightarrow} P$ and $M \stackrel{*}{\Rightarrow} Q$, then there must exist a term R such that $P \stackrel{*}{\Rightarrow} R$ and $Q \stackrel{*}{\Rightarrow} R$. □

The Church-Rosser theorem says that the result of a computation does not depend on the order in which reductions are applied. All possible reduction sequences progress toward the same end result. The end result is a normal form, if one exists.

The Church-Rosser theorem extends to any two beta-equal terms: If $P =_\beta Q$, then there must exist a term R such that $P \stackrel{*}{\Rightarrow} R$ and $Q \stackrel{*}{\Rightarrow} R$.

Computation Rules

Function applications $M\ N$ in programming languages are often implemented as follows: evaluate both M and N, then pass the value of the argument N to the function obtained from M. With this approach, functions are said to be called by value. A similar computation rule can be defined for β-reductions in the lambda calculus.

A *reduction strategy* for the lambda calculus is a rule for choosing redexes; formally, a reduction strategy maps each term P that is not in normal form into a term Q such that $P \underset{\beta}{\Rightarrow} Q$.

leftmost-outermost reaches a normal form if there is one

The *call-by-value reduction strategy* chooses the leftmost-innermost redex in a term. By contrast, the *call-by-name reduction strategy* chooses the leftmost-outermost redex. Here, inner and outer refer to nesting of terms. For example, the entire term is the outermost redex in

$$\underline{(\lambda yz.\ (\lambda x.\ x)\ z\ (yz))\ (\lambda x.\ x)}$$

The innermost redex is the subterm $(\lambda x.\ x)\ z$:

$$(\lambda yz.\ \underline{(\lambda x.\ x)\ z}\ (yz))\ (\lambda x.\ x)$$

The call-by-name strategy is also referred to as *normal-order reduction*; it is guaranteed to reach a normal form, if one exists. Call-by-value, on the other hand, can get stuck, forever evaluating an argument that will never be used. An example can be constructed using $K = \lambda xy.\ x$:

$$(\lambda xy.\ x)\ z\ N \underset{\beta}{\Rightarrow} (\lambda y.\ z)\ N \underset{\beta}{\Rightarrow} z \qquad \text{(call-by-name)}$$

Call-by-value, however, will reduce the innermost redex in the subterm N rather than the entire term $(\lambda y.\ z)\ N$. If reductions starting from N do not terminate, then call-by-value will fail to reach the normal form z. Such an N is $(\lambda x.\ xx)(\lambda x.\ xx)$, which reduces to itself:

$$\begin{aligned}(\lambda y.\ z)\ ((\lambda x.\ xx)(\lambda x.\ xx)) &\underset{\beta}{\Rightarrow} (\lambda y.\ z)\ ((\lambda x.\ xx)(\lambda x.\ xx))\\ &\underset{\beta}{\Rightarrow} (\lambda y.\ z)\ ((\lambda x.\ xx)(\lambda x.\ xx))\\ &\underset{\beta}{\Rightarrow} \cdots \qquad \text{(call-by-value)}\end{aligned}$$

Despite the possibility of an avoidable runaway evaluation, functional languages have used call-by-value because it can be implemented efficiently and it reaches the normal form sufficiently often.

Example 14.6 Call-by-value can reach a normal form faster than call-by-name, where faster means using fewer β-reductions. The term in this example has the form $(\lambda x.\ xx)\ N$. Since the body xx of $\lambda x.\ xx$ has two copies of x, call-by-value will win by first reducing N to a normal form.

The call-by-value reduction takes three steps:

add cons get an lambda

$$\begin{aligned}(\lambda x.\ xx)\ (\ \underline{(\lambda y.\ y)\ (\lambda z.\ z)}\ ) &\underset{\beta}{\Rightarrow} (\lambda x.\ xx)\ (\lambda z.\ z)\\ &\underset{\beta}{\Rightarrow} (\lambda z.\ z)\ (\lambda z.\ z)\\ &\underset{\beta}{\Rightarrow} (\lambda z.\ z)\end{aligned}$$

The call-by-name reduction takes four steps:

$$\begin{aligned}\underline{(\lambda x.\ xx)\ ((\lambda y.\ y)\ (\lambda z.\ z))} &\underset{\beta}{\Rightarrow} ((\lambda y.\ y)\ (\lambda z.\ z))\ ((\lambda y.\ y)\ (\lambda z.\ z))\\ &\underset{\beta}{\Rightarrow} (\lambda z.\ z)\ ((\lambda y.\ y)\ (\lambda z.\ z))\\ &\underset{\beta}{\Rightarrow} (\lambda y.\ y)\ (\lambda z.\ z)\\ &\underset{\beta}{\Rightarrow} (\lambda z.\ z)\end{aligned}$$

□

14.4 PROGRAMMING CONSTRUCTS AS LAMBDA-TERMS

Constants and a little syntactic sugar will be added to the pure lambda calculus in this section to build a tiny functional programming language, called ML0 . The purpose is not to build a real language but to support the claim that a functional language is essentially a lambda calculus. Certain properties of programming languages can therefore be studied in terms of the lambda calculus.

An Applied Lambda Calculus

add constants to get an applied lambda calculus

Terms in an *applied lambda calculus* have the following syntax:

$$M \ ::= \ c \ | \ x \ | \ (\,M_1\ M_2\,) \ | \ (\,\lambda x.\, M\,)$$

Constants, represented by c, correspond to the built-in constants and operators in a programming language. Each applied lambda calculus has its own set of constants. The constants in this section are

true, **false**
if
0, **iszero**, **pred**, **succ**
fix

The definitions of free and bound variables, substitution, α-conversion, and β-reduction carry over from the pure calculus to an applied calculus. As usual, parentheses can be dropped, so

if x y **false**

is a way of writing