6 The Lambda Calculus

The lambda abstraction and its copy rule come from a system invented in the 1930s by Church, called the *lambda calculus*. Church developed the lambda calculus to study the foundations of mathematics and logic. At one time Church hoped that the lambda calculus would serve as a set-theoretic-like foundation for mathematics, but this goal proved unachievable. In the 1960s, Strachey, Landin, and others observed that the lambda calculus worked well as a notation for stating the semantic properties of computer programming languages. This application has proved fruitful, and the lambda calculus stands today as a fundamental tool for programming language design and analysis.

6.1 The Untyped Lambda Calculus

The lambda calculus is pleasant because it is so simple. Its syntax is in Figure 6.1. The first construct in the syntax rule is called a *lambda abstraction*, the second is an *application* (or *combination*), and the third is an *identifier* (or *variable*).

In the lambda abstraction $\lambda I_0.E$, the I_0 is called a *binding identifier*. The *scope* of λI_0 is E, less those lambda abstractions within E whose binding identifiers are also I_0. Occurrences of I_0 in E within λI_0's scope are said to be *bound*; an unbound identifier is *free*. The free identifiers in an expression, E, are denoted by $FV(E)$:

$$FV(\lambda I.E) = FV(E) - \{I\}$$
$$FV(E_1\ E_2) = FV(E_1) \cup FV(E_2)$$
$$FV(I) = \{I\}$$

An expression, E, is *closed* if $FV(E) = \emptyset$.

The free identifiers in an expression can be affected by substitution. We write $[E_1/I]E_2$ to denote the substitution of E_1 for all free occurrences of I in E_2. Substitution is defined as usual:

E ∈ Expression
I ∈ Identifier

$$E ::= (\lambda I.\, E) \mid (E_1\ E_2) \mid I$$

Figure 6.1

$[E/I](\lambda I.E_1) = \lambda I.E_1$

$[E/I](\lambda J.E_1) = \lambda J.\, [E/I]E_1$, if $I \neq J$ and $J \notin FV(E)$

$[E/I](\lambda J.E_1) = \lambda K.\, [E/I][K/J]E_1$, if $I \neq J$, $J \in FV(E)$, and K is fresh

$[E/I](E_1\ E_2) = [E/I]E_1\ [E/I]E_2$

$[E/I]I = E$

$[E/I]J = J$, if $J \neq I$

Here are a few examples of lambda calculus expressions:

$(\lambda X.\, X)$

$(\lambda X.\, (X\ (Y\ X)))$

$((\lambda X.\, (X\ X))(\lambda Y.\, Z))$

$(\lambda X.\, (\lambda Y.\, (X\ (\lambda X.\, (\lambda Y.\, Y)))))$

The piles of brackets prove tiresome, so we drop the outermost set and abbreviate nested combinations $((E_1\ E_2)\ E_3)$ to $(E_1\ E_2\ E_3)$. We abbreviate $(\lambda I.(E))$ to $(\lambda I.\, E)$ as well. Here are the abbreviated forms of the previous expressions:

$\lambda X.\, X$

$\lambda X.\, X\ (Y\ X)$

$(\lambda X.\, X\ X)\,(\lambda Y.\, Z)$

$\lambda X.\, \lambda Y.\, X\ (\lambda X.\, \lambda Y.\, Y)$

With this abbreviation, the scope of a binding identifier, I, in $\lambda I. \cdots$ extends from the period that follows I to the first unmatched right parenthesis (or the end of the phrase, whichever comes first), less any lambda abstractions with the same binding identifier.

So far, it is not clear what the expressions *mean*. Don't worry—they don't mean anything, yet! For the moment, the lambda calculus is just a notation of identifiers.

These rewriting rules manipulate lambda expressions:

α-rule: $\lambda\mathrm{I.\,E} \;\Rightarrow\; \lambda\mathrm{J}.[\mathrm{J}/\mathrm{I}]\mathrm{E}$, if $\mathrm{J} \notin FV(\mathrm{E})$

β-rule: $(\lambda\mathrm{I.\,E_1})\,\mathrm{E_2} \;\Rightarrow\; [\mathrm{E_2}/\mathrm{I}]\mathrm{E_1}$

η-rule: $\lambda\mathrm{I.\,E\,I} \;\Rightarrow\; \mathrm{E}$, if $\mathrm{I} \notin FV(\mathrm{E})$

The α-rule suggests that the choice of binding identifier is unimportant; the β-rule says that binding of an argument to a binding identifier is just substitution; and the η-rule implies that all lambda calculus expressions represent functions. The α-, β-, and η-rules make good sense for a logic of functions, and a good intuition to have is that the lambda calculus is a language of purely functions.

We do not use the α- and η-rules for computation; for the moment, that leaves just the β-rule. Here is a sample computation, where the substitutions are displayed:

$$\begin{aligned}
&(\lambda Y.\,(\lambda X.\,(Y\,X))\,Y)\,(\lambda Z.\,X)\\
&\Rightarrow\; [\lambda Z.X/Y](\lambda X.\,(Y\,X))\,Y\\
&=\; (\lambda A.\,[\lambda Z.X/Y][A/X](Y\,X))\,([\lambda Z.X/Y]Y)\\
&=\; (\lambda A.\,[\lambda Z.X/Y](Y\,A))\,(\lambda Z.X)\\
&=\; (\lambda A.\,(\lambda Z.\,X)A)\,(\lambda Z.\,X)\\
&\Rightarrow\; [\lambda Z.X/A]((\lambda Z.\,X)\,A)\\
&=\; (\lambda Z.\,X)\,(\lambda Z.\,X) \;\Rightarrow\; [\lambda Z.X/Z]X \;=\; X
\end{aligned}$$

Say that an expression, E, contains the subexpression $(\lambda\mathrm{I.\,E_1})\mathrm{E_2}$; the subexpression is called a (β-)*redex*; the expression $[\mathrm{E_2}/\mathrm{I}]\mathrm{E_1}$ is its *contractum*; and the action of replacing the contractum for the redex in E, giving E', is called a *contraction* or a *reduction step*. A reduction step is written $\mathrm{E}\Rightarrow\mathrm{E}'$. A *reduction sequence* is a series of reduction steps $\mathrm{E_1}\Rightarrow\mathrm{E_2}\Rightarrow\cdots\Rightarrow\mathrm{E}_i\Rightarrow\cdots$ that has finite or infinite length. If the sequence has finite length, starting at $\mathrm{E_1}$ and ending at E_n, $n \geq 0$, we write $\mathrm{E_1}\Rightarrow^*\mathrm{E}_n$. (Note that, for every E, $\mathrm{E}\Rightarrow^*\mathrm{E}$.) A lambda expression is in (β-)*normal form* if it contains no (β-)redexes.* An expression *has* normal form if it can be rewritten to an expression *in* normal form. A fun and time-consuming game is to rewrite lambda expressions into their normal forms. But beware—not all expressions have normal forms. Here is a famous example that does not:

$$(\lambda X.\,X\,X)\,(\lambda X.\,X\,X) \;\Rightarrow\; (\lambda X.\,X\,X)\,(\lambda X.\,X\,X) \;\Rightarrow\; (\lambda X.\,X\,X)\,(\lambda X.\,X\,X) \;\Rightarrow\; \cdots$$

Here is another example that you should try: $(\lambda X.\,X\,(X\,X))\,(\lambda X.\,X\,(X\,X))$; and yet another: $\lambda F.\,(\lambda X.\,F\,(X\,X))(\lambda X.\,F\,(X\,X))$. None of these has a normal form. It is no coincidence that all of them use *self-application*—the application of an expression to

* If both the β- and η-rules were used in reduction sequences, we would say an expression is in $\beta\eta$-normal form if it had no $\beta\eta$-redexes.

itself. It is through self-application that repetitive computation can be simulated in the lambda calculus. Indeed, the third of the previous three examples is famous because it can encode recursive function definitions. Call it **Y**; a definition **rec-define** $f = \cdots f \cdots$ is encoded **define** $f = \mathbf{Y}(\lambda f. \cdots f \cdots)$.

Arithmetic can be simulated in the lambda calculus. Although it is not our intention to use the lambda calculus in this way, it is interesting that we can define the following simulations of the natural numbers, called the ‘‘Church numerals’’:

$\overline{0}$: $\lambda S.\ \lambda Z.\ Z$
$\overline{1}$: $\lambda S.\ \lambda Z.\ S\ Z$
$\overline{2}$: $\lambda S.\ \lambda Z.\ S\,(S\ Z)$
$\cdots$
$\overline{i}$: $\lambda S.\ \lambda Z.\ S\,(S\,(\cdots(S\ Z)\cdots))$, S repeated i times

Now we can encode numeric operations, such as the successor function, $succ = \lambda N.\ \lambda S.\ \lambda Z.\ S\,(N\ S\ Z)$, and addition, $add = \lambda M.\lambda N.\ M\ succ\ N$, and indeed, all of the general recursive functions, in the sense that, if $\overline{n}$ is the Church numeral for number n, then for every general recursive function $f: \mathbb{N} \rightarrow \mathbb{N}$, we can construct a lambda expression, F, such that, for all $n \in \mathbb{N}$, $(F\ \overline{n}) \Rightarrow^* \overline{f(n)}$ (when $f\,(n)$ is defined, otherwise $(F\ \overline{n})$ has no normal form). This means the lambda calculus is as powerful a computation system as any known. A consequence of this result is that it is impossible to mechanically decide whether or not an arbitrary expression has a normal form. (If we could decide this question, then we could solve the famous, undecidable ‘‘halting problem.’’)

The β-rule has several pleasing properties. The most important are these three:

6.1 Theorem (The Confluence Property)

For any lambda expression E, if $E \Rightarrow^* E_1$ and $E \Rightarrow^* E_2$, then there exists a lambda expression, E_3, such that $E_1 \Rightarrow^* E_3$ and $E_2 \Rightarrow^* E_3$ (modulo application of the α-rule to E_3).

This theorem implies that the order in which we reduce redexes is unimportant; all reduction sequences can be made to meet at a common expression (up to renaming of binding identifiers). But the confluence property does not say that all reduction sequences *must* meet, only that they can be made to meet; for example, let Δ be $(\lambda X.\ X\ X)$. For $E_0 = (\lambda Y.Z)(\Delta\,\Delta)$, we have $E_0 \Rightarrow Z$ and $E_0 \Rightarrow E_0$. By the confluence property, there must be some E_3 such that $Z \Rightarrow^* E_3$ and $E_0 \Rightarrow^* E_3$. Since Z is in normal form, E_3 must be Z, and indeed, $E_0 \Rightarrow^* Z$. But notice that the reduction sequence $E_0 \Rightarrow E_0 \Rightarrow \cdots \Rightarrow E_0 \Rightarrow \cdots$ never reaches Z, although the confluence property guarantees that a reduction sequence $E_0 \Rightarrow^* E_0$ can be extended to $E_0 \Rightarrow^* E_0 \Rightarrow^* Z$. Thus, the strategy for choosing redexes is significant. A key consequence of confluence is

6.2 Corollary (The Uniqueness of Normal Forms Property)

If E can be rewritten to some E' in normal form, then E' is unique (modulo application of the α-rule).

The proof of this property is easy: If expression E would, in fact, reduce to distinct normal forms E_1 and E_2, the confluence property tells us there is some E_3 such that $E_1 \Rightarrow^* E_3$ and $E_2 \Rightarrow^* E_3$. But this is impossible, since E_1 and E_2 cannot be further reduced.

Uniqueness of normal forms is crucial, because it gives a programmer a naive but useful semantics for the lambda calculus: The ‘‘meaning’’ of an expression is the normal form to which it rewrites.

The confluence property is sometimes called the *Church-Rosser property*, after the persons who first proved it. A proof of the confluence property is described in Section 6.8.

Finally, there is a particular rewriting strategy that always discovers a normal form, if one exists. Say that the *leftmost-outermost redex* in an expression is the redex whose λ-symbol lies to the leftmost in the expression. (The redex is ‘‘leftmost-outermost’’ in the sense that, when the expression is drawn as a tree, the redex is outermost—it is not embedded in another redex—and it is the leftmost of all the outermost redexes.) The leftmost-outermost rewriting strategy reduces the leftmost-outermost redex at each stage until no more redexes exist.

6.3 Theorem (The Standardization Property)

If an expression has a normal form, it will be found by the leftmost-outermost rewriting strategy.

Here is an example of a reduction by the leftmost-outermost strategy:

$$(\lambda Y.\ Y\ Y)\ ((\lambda X.X)(\lambda Z.Z))$$
$$\Rightarrow\ ((\lambda X.X)(\lambda Z.Z))\ ((\lambda X.X)(\lambda Z.Z))$$
$$\Rightarrow\ (\lambda Z.Z)\ ((\lambda X.X)(\lambda Z.Z))\ \Rightarrow\ (\lambda X.X)(\lambda Z.Z)\ \Rightarrow\ (\lambda Z.Z)$$

The leftmost-outermost strategy is a ‘‘lazy evaluation’’ strategy—an argument is not reduced until it finally appears as the leftmost-outermost redex in an expression. Other reduction strategies might arrive at a normal form in fewer steps. The previous reduction is done quicker if we reduce the rightmost-innermost redex each time:

$(\lambda Y.\ Y\ Y)\,((\lambda X.X)(\lambda Z.Z))$
$\Rightarrow\ (\lambda Y.\ Y\ Y)\,(\lambda Z.Z)$
$\Rightarrow\ (\lambda Z.Z)\,(\lambda Z.Z)\ \Rightarrow\ (\lambda Z.Z)$

The rightmost-innermost strategy roughly corresponds to ‘‘eager evaluation.’’ But in some cases the strategy will not discover a normal form, whereas the leftmost-outermost strategy will, for example, $(\lambda Y.Z)((\lambda X.\ X\ X)(\lambda X.\ X\ X))$.

6.2 Call-by-Name and Call-by-Value Reduction

The β-rule in the previous section is sometimes titled the *call-by-name* β-rule because it places no restriction on the argument, E_2, of the redex $(\lambda \mathrm{I}.\mathrm{E}_1)\mathrm{E}_2$. There is an alternative, the *call-by-value* β-rule (or β-*val* rule), which does restrict the form of E_2.

Say that identifiers, I, and lambda abstractions, λI.E, are *Values*. The call-by-value β-rule reads as follows:

β-*val*: $(\lambda \mathrm{I}.\mathrm{E}_1)\mathrm{E}_2\ \Rightarrow\ [\mathrm{E}_2/\mathrm{I}]\mathrm{E}_1$, if E_2 is a Value

This roughly corresponds to call-by-value evaluation of actual parameters in programming languages. Here is a computation undertaken with the β-*val* rule:

$(\lambda W.A)((\lambda Y.\lambda Z.\ Z\ Z)((\lambda X.\ B)C)\,(\lambda X.\ X\ X))$
$\Rightarrow\ (\lambda W.A)((\lambda Y.\lambda Z.\ Z\ Z)C\ (\lambda X.\ X\ X))$
$\Rightarrow\ (\lambda W.A)((\lambda Z.\ Z\ Z)(\lambda X.\ X\ X))$
$\Rightarrow\ (\lambda W.A)((\lambda X.\ X\ X)(\lambda X.\ X\ X))$
$\Rightarrow\ (\lambda W.A)((\lambda X.\ X\ X)(\lambda X.\ X\ X))\ \Rightarrow\ \cdots$

At each stage there is only one redex, and the computation is nonterminating, since the argument to λW will never reduce to a Value. Contrast the above reduction sequence to one generated by the usual β-rule, which terminates in one step. Perhaps this suggests that the β-*val* rule is not as ‘‘useful’’ as the β-rule, but Section 6.7 will show that the β-*val* rule is often the appropriate one for specifying the operational semantics of programming languages.

A useful insight is that the β-*val* rule can be coded as *two* rules:

β-val_1: $(\lambda \mathrm{I}.\mathrm{E}_1)\mathrm{J}\ \Rightarrow\ [\mathrm{J}/\mathrm{I}]\mathrm{E}_1$
β-val_2: $(\lambda \mathrm{I}.\mathrm{E}_1)(\lambda \mathrm{J}.\mathrm{E}_2)\ \Rightarrow\ [(\lambda \mathrm{J}.\mathrm{E}_2)/\mathrm{I}]\mathrm{E}_1$

Here, the notion of ‘‘Value’’ is encoded within the patterns of the two rules. Another variant of the β-*val* rule is just β-val_2: Only lambda abstractions are Values. This variant works well with a lambda calculus of closed expressions. Yet another variant arises when additional phrases, such as arithmetic expressions, are added to the lambda calculus. In the case of arithmetic, the numerals 0, 1, 2, and so on, are Values, and we have:

β-*val*$_3$: $(\lambda \mathrm{I}.\mathrm{E}_1)n \;\Rightarrow\; [n/\mathrm{I}]\mathrm{E}_1$, if n is a numeral

When the β-*val* rule is used, the purpose of a computation is to reduce an expression to a Value. But Values differ from normal forms: An expression—even one without a normal form—can reduce to more than one Value. An example is $(\lambda X.\ \lambda Y.\ (\lambda Z.\ Z\ Z\ Z)$ $(\lambda Z.\ Z\ Z\ Z))A$. Nonetheless, for typed programming languages, the notion of Value is useful because the inputs and outputs for programs are typically phrases of type *int*, *bool*, and the like. For these types, Values and normal forms coincide. Section 6.6 gives details.

6.3 An Induction Principle

Proofs of properties of programming languages are usually undertaken by structural induction. But structural induction often fails to work when properties about substitution must be proved. Let $\mathrm{E}_1 \equiv \mathrm{E}_2$ mean that E_1 and E_2 are identical, modulo use of the α-rule, and consider this example:

6.4 Proposition

For all identifiers, I, *and expressions,* E_1 *and* E_2, *if* $\mathrm{I} \notin FV(\mathrm{E}_1)$, *then* $[\mathrm{E}_2/\mathrm{I}]\mathrm{E}_1 \equiv \mathrm{E}_1$.

Attempted proof: We use induction on the structure of E_1. There are three cases:

(i) $\mathrm{E}_1 = \mathrm{J}$: Assume $\mathrm{I} \notin FV(\mathrm{J})$. This implies $\mathrm{I} \neq \mathrm{J}$, so $[\mathrm{E}_2/\mathrm{I}]\mathrm{J} = \mathrm{J}$.

(ii) $\mathrm{E}_1 = (\mathrm{E}_{11}\ \mathrm{E}_{12})$: Assume $\mathrm{I} \notin FV(\mathrm{E}_{11}\ \mathrm{E}_{12})$; then $\mathrm{I} \notin FV(\mathrm{E}_{11})$ and $\mathrm{I} \notin FV(\mathrm{E}_{12})$. Since $[\mathrm{E}_2/\mathrm{I}](\mathrm{E}_{11}\ \mathrm{E}_{12}) = ([\mathrm{E}_2/\mathrm{I}]\mathrm{E}_{11}\ [\mathrm{E}_2/\mathrm{I}]\mathrm{E}_{12})$, the result follows immediately from the inductive hypotheses for E_{11} and E_{12}.

(iii) $\mathrm{E}_1 = \lambda \mathrm{J}.\mathrm{E}_{11}$: Assume $\mathrm{I} \notin FV(\lambda \mathrm{J}.\mathrm{E}_{11})$. If $\mathrm{I} = \mathrm{J}$, the result is immediate. If $\mathrm{I} \neq \mathrm{J}$ and $\mathrm{J} \notin FV(\mathrm{E}_2)$, then $[\mathrm{E}_2/\mathrm{I}](\lambda \mathrm{J}.\mathrm{E}_{11}) = (\lambda \mathrm{J}.[\mathrm{E}_2/\mathrm{I}]\mathrm{E}_{11})$. We have that $\mathrm{I} \notin FV(\mathrm{E}_{11})$, so the result follows from the inductive hypothesis for E_{11}. Finally, consider when $\mathrm{I} \neq \mathrm{J}$ and $\mathrm{J} \in FV(\mathrm{E}_2)$. Then, $[\mathrm{E}_2/\mathrm{I}](\lambda \mathrm{J}.\mathrm{E}_{11}) = (\lambda \mathrm{K}.[\mathrm{E}_2/\mathrm{I}][\mathrm{K}/\mathrm{J}]\mathrm{E}_{11})$. Now, we are stuck, because the inductive hypothesis applies only to E_{11} and *not* to $[\mathrm{K}/\mathrm{J}]\mathrm{E}_{11}$, which may be a different expression!

The technical problem with substitution is frustrating, since the renaming of J to K in case (iii) is cosmetic and does not affect the structure of E_{11} at all. The problem has motivated some researchers to replace explicit substitution by an implicit form (where substitution always, automatically, correctly occurs) or even to abandon identifiers and substitution altogether and replace identifiers by numerical offsets, similar to the relative addresses for variables that a compiler calculates. The exercises introduce these alternatives. Here, we stick with substitution and sidestep the problem in the traditional way. We define the *rank* of a lambda calculus expression as follows:

$$\begin{aligned} &rank(\mathrm{I}) = 0 \\ &rank(\mathrm{E}_1\ \mathrm{E}_2) = max\{rank(\mathrm{E}_1),\ rank(\mathrm{E}_2)\} + 1 \\ &rank(\lambda \mathrm{I}.\ \mathrm{E}) = rank(\mathrm{E}) + 1 \end{aligned}$$

The definition of *rank* admits a useful induction principle:

6.5 Theorem (Induction on Rank)

To prove that a property, P, holds for all lambda calculus expressions, it suffices to prove, for an arbitrary expression, E_0, *with* $rank(\mathrm{E}_0) = j$, *that*
if (for all expressions, E, *such that* $rank(\mathrm{E}) < j$, $P(\mathrm{E})$ *holds),*
then $P(\mathrm{E}_0)$ *holds.*

The soundness of induction on rank can be proved by means of mathematical induction; this is left as an exercise. The utility of induction on rank is enhanced by the following lemma.

6.6 Lemma

For all identifiers I *and* J *and expressions* E, $rank(\mathrm{E}) = rank([\mathrm{J}/\mathrm{I}]\mathrm{E})$.

Proof: The proof is by induction on the rank of E. For arbitrary $j \geq 0$, we may assume, for all expressions E' such that $rank(\mathrm{E}') < j$, that for all I and J, $rank(\mathrm{E}') = rank([\mathrm{J}/\mathrm{I}]\mathrm{E}')$. We must show, when $rank(\mathrm{E}) = j$, that for all I and J, $rank(\mathrm{E}) = rank([\mathrm{J}/\mathrm{I}]\mathrm{E})$. There are three cases:

(i) E=K, an identifier: Then $rank([\mathrm{J}/\mathrm{I}]\mathrm{K}) = 0 = rank(\mathrm{K})$, whether or not I equals K.

(ii) $\mathrm{E} = (\mathrm{E}_1\ \mathrm{E}_2)$: $[\mathrm{J}/\mathrm{I}](\mathrm{E}_1\ \mathrm{E}_2) = ([\mathrm{J}/\mathrm{I}]\mathrm{E}_1\ [\mathrm{J}/\mathrm{I}]\mathrm{E}_2)$. By the definition of *rank*, $rank(\mathrm{E}_1) < rank(\mathrm{E})$ and $rank(\mathrm{E}_2) < rank(\mathrm{E})$, and by the inductive hypothesis, $rank([\mathrm{J}/\mathrm{I}]\mathrm{E}_1) = rank(\mathrm{E}_1)$ and $rank([\mathrm{J}/\mathrm{I}]\mathrm{E}_2) = rank(\mathrm{E}_2)$; this implies the result.

(iii) $\mathrm{E}=(\lambda\mathrm{K}.\ \mathrm{E}_1)$: the only interesting case to consider of $[\mathrm{J}/\mathrm{I}](\lambda\mathrm{K}.\ \mathrm{E}_1)$ is when $\mathrm{I} \neq \mathrm{K}$ but $\mathrm{J}=\mathrm{K}$. The resulting expression is $(lam \mathrm{L}.\ [\mathrm{J}/\mathrm{I}][\mathrm{L}/\mathrm{J}]\mathrm{E}_1)$. Since $rank(\mathrm{E}_1) < rank(\mathrm{E})$, we have that $rank([\mathrm{L}/\mathrm{J}]\mathrm{E}_1) = rank(\mathrm{E}_1)$, by the inductive hypothesis. But this implies $rank([\mathrm{L}/\mathrm{J}]\mathrm{E}_1) < rank(\mathrm{E})$, and by applying the inductive hypothesis again, we have $rank([\mathrm{J}/\mathrm{I}][\mathrm{L}/\mathrm{J}]\mathrm{E}_1) = rank([\mathrm{L}/\mathrm{J}]\mathrm{E}_1) = rank(\mathrm{E}_1)$. This gives the result. □

With the aid of Lemma 6.6, we can view induction by rank as a form of structural induction that is unaffected by cosmetic substitutions. We return to this point momentarily, but first we perform the proof of Proposition 6.4.

Proof of Proposition 6.4: The proof is by induction on the rank of E_1. For arbitrary $j \geq 0$, we assume, for all expressions E' such that $rank(E') < j$, that $I \notin FV(E')$ implies $[E_2/I]E' \equiv E'$. We must show the same result for E_1, where $rank(E_1) = j$. Now, E_1 must have one of three forms:

(i) $E_1 = J$: The reasoning in the ''attempted proof'' applies.
(ii) $E_1 = (E_{11}\, E_{12})$: The reasoning in the ''attempted proof'' applies, since $rank(E_{11}) < rank(E_1)$ and $rank(E_{12}) < rank(E_1)$.
(iii) $E_1 = (\lambda J.E_{11})$: If $I = J$, the reasoning in the ''attempted proof'' applies. If $I \neq J$ and $J \notin FV(E_2)$, then the ''attempted proof'' reasoning applies, since $rank(E_{11}) < rank(E_1)$. Finally, if $I \neq J$ and $J \in FV(E_2)$, then $[E_2/I](\lambda J.E_{11}) = (\lambda K.\ [E_2/I][K/J]E_{11})$. By Lemma 6.6, $rank([K/J]E_{11}) = rank(E_{11}) < rank(E_1)$. Since $I \notin FV(E_{11})$, it is easy to prove that $I \notin FV([K/J]E_{11})$. Therefore, the inductive hypothesis gives us $[E_2/I][K/J]E_{11} \equiv [K/J]E_{11}$. Since $(\lambda J.E_{11}) \equiv (\lambda K.[K/J]E_{11})$, this gives the result. $\square$

This proof looks almost exactly like a proof by structural induction where the only exception is when a cosmetic renaming of identifiers is necessary, so from here on, we write proofs that use induction on rank as if they are structural induction proofs. Thanks to Lemma 6.6, we obtain an additional inductive hypothesis for cosmetic substitutions like $[K/J]E_{11}$. Induction by rank will be essential to proofs of several standard results for the lambda calculus.

6.4 The Simply Typed Lambda Calculus

An important extension to the lambda calculus is a typing system. The *simply typed lambda calculus* is presented in Figure 6.2. Since the system is similar to the one in Chapter 5, we do not consider examples.

When we write type expressions of the form $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$, we usually drop the rightmost brackets and write $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. The β-rule applies to the simply typed lambda calculus:

$$\beta\text{-rule: } ((\lambda I{:}\tau.\, E_1)\, E_2) \Rightarrow [E_2/I]E_1$$

When the β-*val* rule is used, it reads as before, namely,

$$\beta\text{-}val \text{ rule: } ((\lambda I{:}\tau.\, E_1)\, E_2) \Rightarrow [E_2/I]E_1, \text{ if } E_1 \text{ is a Value}$$

where the notion of ''Value'' is defined recursively on the types in the language:

$E \in$ Expression
$I \in$ Identifier
$\tau \in$ Data-Type
$\iota \in$ Primitive-Data-Type (for example, *int*)

$$E ::= (\lambda I{:}\tau.\, E) \mid (E_1\ E_2) \mid I$$
$$\tau ::= \iota \mid \tau_1 \rightarrow \tau_2$$

$$\frac{\pi \uplus \{I{:}\tau_1\} \vdash E{:}\,\tau_2}{(\lambda\, I{:}\,\tau_1.\, E){:}\,\tau_1 \rightarrow \tau_2} \qquad \frac{\pi \vdash E_1{:}\,\tau_1 \rightarrow \tau_2 \qquad \pi \vdash E_2{:}\,\tau_1}{\pi \vdash (E_1\ E_2){:}\,\tau_2} \qquad \pi \vdash I{:}\,\tau, \quad \text{if } (I{:}\tau) \in \pi$$

Figure 6.2

$Value(\iota) =$ a set of constants, for example, $Value(int) = \{0, 1, 2, \ldots\}$
$Value(\tau_1 \rightarrow \tau_2) = \{(\lambda I{:}\tau_1.E) \mid$ there exists π such that $\pi \vdash (\lambda I{:}\tau_1.E){:}\,\tau_1 \rightarrow \tau_2$ holds$\}$

The simply typed lambda calculus has the confluence, uniqueness of normal forms, and unicity of typing properties. Another notable property is that substitution preserves typing:

6.7 Lemma

For all π, τ_1, τ_2, I, E_1, *and* E_2, *if* $\pi \vdash E_2{:}\,\tau_2$ *and* $\pi \uplus \{I{:}\tau_2\} \vdash E_1{:}\,\tau_1$ *hold, then so does* $\pi \vdash [E_2/I]E_1{:}\,\tau_1$.

Proof: The proof is by induction on the rank of E_1, which has one of three forms:

(i) $E_1 = I$: If $I = J$, then $[E_2/I]I = E_2$, and since $\pi \vdash E_2{:}\tau_2$, the result holds. If $I \neq J$, then $[E_2/I]J = J$. We know that $\pi \uplus \{I{:}\tau_2\} \vdash J{:}\tau_1$, but this implies $(J{:}\tau_1) \in \pi$, by the typing rule for identifiers, implying $\pi \vdash J{:}\tau_1$.

(ii) $E_1 = (E_{11}\ E_{12})$: The typing rule for combinations implies that $\pi \uplus \{I{:}\tau_2\} \vdash E_{11}{:}\,\tau_{12} \rightarrow \tau_1$ and $\pi \uplus \{I{:}\tau_2\} \vdash E_{12}{:}\,\tau_{12}$ both hold, for some type τ_{12}. The inductive hypothesis for E_{11} yields $\pi \vdash [E_2/I]E_{11}{:}\,\tau_{12} \rightarrow \tau_1$ and for E_{12} yields $\pi \vdash [E_2/I]E_{12}{:}\,\tau_{12}$. By the typing rule for combinations, we get the result.

(iii) $E_1 = (\lambda J{:}\tau_{11}.\, E_{11})$ for $\tau_1 = \tau_{11} \rightarrow \tau_{12}$: There are three subcases:

(a) If $I = J$, we must show $\pi \vdash (\lambda I{:}\tau_{11}.\, E_{11}){:}\,\tau_{11} \rightarrow \tau_{12}$. By hypothesis, we have $\pi \uplus \{I{:}\tau_2\} \vdash (\lambda I{:}\tau_{11}.\, E_{11}){:}\,\tau_{11} \rightarrow \tau_{12}$, and by the typing rule for abstractions we know that $\pi \uplus \{I{:}\tau_1\} \uplus \{I{:}\tau_{11}\} \vdash E_{11}{:}\,\tau_{12}$ holds. But $\pi \uplus \{I{:}\tau_1\} \uplus \{I{:}\tau_{11}\} = \pi \uplus \{I{:}\tau_{11}\}$, so $\pi \uplus \{I{:}\tau_{11}\} \vdash E_{11}{:}\,\tau_{12}$ holds also, and this implies the result.

(b) If $I \neq J$ and J is not free in E_2, then we must show $\pi \vdash (\lambda J{:}\tau_{11}.\, [E_2/I]E_1)$: $\tau_{11} \rightarrow \tau_{12}$. By reasoning similar to that in case (a), we know that $\pi \uplus \{I{:}\tau_1\} \uplus \{J{:}\tau_{11}\} \vdash E_{11}{:}\tau_{12}$ holds. Since $I \neq J$, we also have $\pi \uplus \{J{:}\tau_{11}\} \uplus \{I{:}\tau_1\} \vdash E_{11}{:}\,\tau_{12}$. By the inductive hypothesis on E_{11} we get $\pi \uplus \{J{:}\,\tau_{11}\} \vdash [E_2/I]{:}\,\tau_{12}$, which implies the result.

(c) If $I \neq J$, and J is free in E_2, we must show $\pi \vdash (\lambda K{:}\,\tau_{11}.\, [E_2/I]\,[K/J]E_{11})$: $\tau_{11} \rightarrow \tau_{12}$, where K is fresh. By hypothesis, we have $\pi \uplus \{I{:}\,\tau_1\} \uplus \{J{:}\,\tau_{11}\} \vdash E_{11}{:}\,\tau_{12}$. Since K is fresh, we can easily prove $\pi \uplus \{I{:}\tau_1\} \uplus \{J{:}\tau_{11}\} \uplus \{K{:}\tau_{11}\} \vdash E_{11}{:}\,\tau_{12}$. (The proof is left as an exercise.) Since I, J, and K are all distinct, $\pi \uplus \{K{:}\tau_{11}\} \uplus \{I{:}\tau_1\} \uplus \{J{:}\tau_{11}\} \vdash E_{11}{:}\,\tau_{12}$ holds also. The rule for identifiers gives us $\pi \uplus \{K{:}\tau_{11}\} \uplus \{I{:}\tau_1\} \vdash K{:}\,\tau_{11}$, so we apply the inductive hypothesis to E_{11}. This gives us $\pi \uplus \{K{:}\,\tau_{11}\} \uplus \{I{:}\tau_1\} \vdash [K/J]E_{11}{:}\,\tau_{12}$. Next, we can apply the inductive hypothesis a second time, this time to $[K/J]E_{11}$, and obtain $\pi \uplus \{K{:}\tau_{11}\} \vdash [E_1/I]\,[K/J]E_{11}{:}\,\tau_{12}$. The typing rule for abstractions gives the result. □

A consequence of the above lemma is that the β-rule does not change the typing of an expression. As a result, we obtain this important theorem:

6.8 Theorem (The Subject Reduction Property)

If $\pi \vdash E_1{:}\,\tau$ *holds and* $E_1 \Rightarrow^* E_2$, *then* $\pi \vdash E_2{:}\,\tau$ *holds as well.*

Proof: It suffices to show the result for $E_1 \Rightarrow E_2$, a single reduction step. The proof is by induction on the structure of E_1:

(i) $E_1 = I$: This case is impossible.

(ii) $E_1 = \lambda I{:}\tau_1.\, E_{11}$: Clearly, $E_{11} \Rightarrow E_{11}'$, and the inductive hypothesis for E_{11} implies the result.

(iii) $E_1 = (E_{11}\, E_{12})$: If the reduction step is wholly within E_{11}, that is, $E_{11} \Rightarrow E_{11}'$, then the inductive hypothesis for E_{11} implies the result. A similar argument holds if $E_{12} \Rightarrow E_{12}'$. The only other possibility is that E_1 is $(\lambda I{:}\tau.\, E')E_{12}$ and $E_1 \Rightarrow [E_{12}/I]E'$. Then, the result follows from Lemma 6.7. □

The Subject Reduction Property shows that the β-rule does not change the typing structure of an expression, hence type (re)checking is unnecessary during a reduction sequence.

The next result about the simply typed lambda calculus is startling and crucial:

6.9 Theorem (The Strong Normalization Property)

If $\pi \vdash E{:}\,\tau$ *holds, then every reduction sequence starting from* E *has finite length.*

The proof is nontrivial and will not be shown here; a presentation can be found in Hindley and Seldin 1986 or Thompson 1991.

Two immediate consequences of the strong normalization property are: (i) every well-typed term has a normal form; (ii) any rewriting strategy will find it. The reason for strong normalization is that the typing system prevents the coding of self-application, the method for simulating repetition. A term $(X\ X)$ would have to be typed in a way that the first occurrence of X has type $\tau_1 \rightarrow \tau_2$ and the second occurrence of X has type τ_1. But this is impossible with the rules for a well-typed expression.

Another major benefit of the typing system is that it allows us to reintroduce a denotational semantics and easily prove two important soundness results. These are covered in the next section.

6.5 Denotational Semantics and Soundness

The usual semantics for the simply typed lambda calculus is the lazy evaluation semantics from Chapter 5:

$$[\![\pi \vdash \lambda \mathrm{I}{:}\tau_1.\,\mathrm{E}{:}\,\tau_1{\rightarrow}\tau_2]\!]e \;= f, \quad \text{where } f\,u \;= [\![\pi \uplus \{\mathrm{I}{:}\tau_1\} \vdash \mathrm{E}{:}\,\tau_2]\!](e \uplus \{\mathrm{I}{=}u\})$$
$$\text{and } e \uplus \{\mathrm{I}{=}u\} = \{\mathrm{I}{=}u\} \cup (e - \{(\mathrm{I}{=}v) \mid (\mathrm{I}{=}v) \in e\})$$
$$[\![\pi \vdash \mathrm{E}_1\,\mathrm{E}_2{:}\,\tau_2]\!]e = ([\![\pi \vdash \mathrm{E}_1{:}\,\tau_1{\rightarrow}\tau_2]\!]e)\,([\![\pi \vdash \mathrm{E}_2{:}\,\tau_1]\!]e)$$
$$[\![\pi \vdash \mathrm{I}{:}\,\tau]\!]e = v, \quad \text{where } (\mathrm{I}{=}v) \in e$$

As usual, let $[\![\iota]\!]$ be the set of values named by ι (for example, $[\![int]\!] = Int$), and let $[\![\tau_1 \rightarrow \tau_2]\!] = [\![\tau_1]\!] \rightarrow [\![\tau_2]\!]$, that is, the set of functions from arguments in $[\![\tau_1]\!]$ to answers in $[\![\tau_2]\!]$.

The first important result is soundness of typing. As before, an environment, e, is *consistent* with a type assignment, π, if $(\mathrm{I}{:}\tau) \in \pi$ exactly when $(\mathrm{I}{=}v) \in e$ and $v \in [\![\tau]\!]$. Let Env_π be the set of those environments that are consistent with π.

6.10 Theorem

If $e \in Env_\pi$, *then* $[\![\pi \vdash \mathrm{E}{:}\,\tau]\!]e \in [\![\tau]\!]$.

Proof: The proof is an induction on the structure of E. The three cases are:

(i) An identifier, I: We have $\pi \vdash \mathrm{I}{:}\theta$ and $(\mathrm{I}{:}\theta) \in \pi$. The result follows because e is consistent with π.

(ii) An application $(\mathrm{E}_1\ \mathrm{E}_2)$: By the typing rule for applications and the inductive hypothesis, we have $[\![\pi \vdash \mathrm{E}_1{:}\,\tau_1 \rightarrow \tau_2]\!]e \in [\![\tau_1 \rightarrow \tau_2]\!]$ and $[\![\pi \vdash \mathrm{E}_2{:}\,\tau_1]\!]e \in [\![\tau_1]\!]$. Thus, $([\![\pi \vdash \mathrm{E}_1{:}\,\tau_1{\rightarrow}\tau_2]\!]e)([\![\pi \vdash \mathrm{E}_2{:}\,\tau_1]\!]e) \in [\![\theta_2]\!]$.

(iii) An abstraction $\lambda \mathrm{I}{:}\tau_1.\,\mathrm{E}$: By the typing rule for abstractions, we have

$\pi \uplus \{\mathrm{I}{:}\tau_1\} \vdash \mathrm{E}{:}\tau_2$. Assume that $v \in [\![\tau_1]\!]$, for an arbitrary v. Then, $e \uplus \{\mathrm{I}{=}v\}$ is consistent with $\pi \uplus \{\mathrm{I}{:}\tau_1\}$. By the inductive hypothesis, we have $[\![\pi \vdash \mathrm{E}{:}\ \tau_2]\!]\ (e \uplus \{\mathrm{I}{=}v\}) \in [\![\tau_2]\!]$. Since v is arbitrary, the function $f(v) = [\![\pi \vdash \mathrm{E}{:}\ \tau_2]\!](e \uplus \{\mathrm{I}{=}v\})$ is in the set $[\![\tau_1 \rightarrow \tau_2]\!]$. □

The typing theorem gives a ''road map'' for the semantics of the typed lambda calculus, telling us what form of value a well-typed expression represents. If we attempt to use the above denotational semantics for the untyped lambda calculus (delete all typing information), we encounter a significant problem: An expression like $\lambda X.\, X$ apparently represents an identity function that can take all other functions—including itself, $(\lambda X.\, X)\,(\lambda X.\, X)$—as arguments. This implies that the set theoretic representation of the identity function is a set of argument, answer pairs of the form: $Id = \{\ldots,\ (Id, Id),\ \ldots\}$. Such a set is outlawed by the foundation axiom of set theory, so the well definedness of the semantics for untyped lambda calculus falls into doubt. There is a way to repair the problem, due to Scott, and see a denotational semantics text for details.

The next important result, one that has been promised for several chapters, is the proof of soundness of the β-rule. The result is a corollary of the following:

6.11 Theorem (The ''Substitution Lemma'')

For all $e \in Env_\pi$, $[\![\pi \vdash [\mathrm{E}_2/\mathrm{I}]\mathrm{E}_1{:}\ \tau_1]\!]e = [\![\pi \uplus \{\mathrm{I}{:}\tau_2\} \vdash \mathrm{E}_1{:}\ \tau_1]\!](e \uplus \{\mathrm{I} = [\![\pi \vdash \mathrm{E}_2{:}\ \tau_2]\!]e\})$.

Proof: The proof is by induction on the rank of E_1.

(i) $\mathrm{E}_1 = \mathrm{J}$: If $\mathrm{J} \neq \mathrm{I}$, the result is immediate. If $\mathrm{J} = \mathrm{I}$, then $[\![\pi \vdash [\mathrm{E}_2/\mathrm{I}]\mathrm{I}{:}\ \tau_1]\!]e = [\![\pi \vdash \mathrm{E}_2{:}\ \tau_1]\!]e = [\![\pi \uplus \{\mathrm{I}{:}\tau_1\} \vdash \mathrm{I}{:}\ \tau_1]\!](e \uplus \{\mathrm{I} = [\![\pi \vdash \mathrm{E}_2{:}\ \tau_1]\!]e\})$.

(ii) $\mathrm{E}_1 = (\mathrm{E}_{11}\ \mathrm{E}_{12})$: Since $[\mathrm{E}_2/\mathrm{I}](\mathrm{E}_{11}\ \mathrm{E}_{12}) = ([\mathrm{E}_2/\mathrm{I}]\mathrm{E}_{11}\ [\mathrm{E}_2/\mathrm{I}]\mathrm{E}_{12})$, the result follows from the inductive hypotheses for E_{11} and E_{12}.

(iii) $\mathrm{E}_1 = (\lambda \mathrm{J}{:}\tau_{11}.\, \mathrm{E}_{12})$: If $\mathrm{J} = \mathrm{I}$, the result is immediate. If $\mathrm{J} \neq \mathrm{I}$ and J is not free in E_2, then the result follows from the inductive hypothesis for E_{12}. Otherwise, $[\![\pi \vdash [\mathrm{E}_2/\mathrm{I}](\lambda \mathrm{J}{:}\tau_{11}.\, \mathrm{E}_{12}){:}\ \tau_{11} \rightarrow \tau_{12}]\!]e = [\![\pi \vdash \lambda \mathrm{K}{:}\tau_{11}.\, [\mathrm{E}_2/\mathrm{I}][\mathrm{K}/\mathrm{J}]\mathrm{E}_{12}{:}\ \tau_{11} \rightarrow \tau_{12}]\!]e = f$, such that $f(v) = [\![\pi_1 \vdash [\mathrm{E}_2/\mathrm{I}][\mathrm{K}/\mathrm{J}]\mathrm{E}_{12}{:}\ \tau_{12}]\!]e_1$, where $\pi_1 = \pi \uplus \{\mathrm{K}{:}\tau_{11}\}$ and $e_1 = e \uplus \{\mathrm{K}{=}v\}$. Now, $[\mathrm{K}/\mathrm{J}]\mathrm{E}_{12}$ has the same rank as E_{12}, so the inductive hypothesis says that the previous value equals $[\![\pi_1 \uplus \{\mathrm{I}{:}\tau_2\} \vdash [\mathrm{K}/\mathrm{J}]\mathrm{E}_{12}{:}\ \tau_{12}]\!](e_1 \uplus \{\mathrm{I}{=}[\![\pi \vdash \mathrm{E}_2{:}\ \tau_2]\!]e_1\}) = [\![\pi_1 \uplus \{\mathrm{I}{:}\tau_2\} \vdash [\mathrm{K}/\mathrm{J}]\mathrm{E}_{12}{:}\ \tau_{12}]\!](e_1 \uplus \{\mathrm{I}{=}[\![\pi \vdash \mathrm{E}_2{:}\ \tau_2]\!]e\})$, since K is not in E_2. Let $\pi_2 = \pi_1 \uplus \{\mathrm{I}{:}\tau_2\}$ and $e_2 = e_1 \uplus \{\mathrm{I}{=}[\![\pi \vdash \mathrm{E}_2{:}\ \tau_2]\!]e\}$; by the inductive hypothesis on E_{12}, the previous value equals $[\![\pi_2 \uplus \{\mathrm{J}{:}\ \tau_{11}\} \vdash \mathrm{E}_{12}{:}\ \tau_{12}]\!](e_2 \uplus \{\mathrm{J}{=}[\![\pi \uplus \{\mathrm{K}{:}\ \tau_{11},\ \mathrm{I}{:}\tau_2\} \vdash \mathrm{K}{:}\ \tau_{11}]\!]e_2\}) = [\![\pi_2 \uplus \{\mathrm{J}{:}\ \tau_{11}\} \vdash \mathrm{E}_{12}{:}\ \tau_{12}]\!](e_2 \uplus \{\mathrm{J}{=}v\}) = [\![\pi \uplus \{\mathrm{I}{:}\tau_2,\ \mathrm{J}{:}\tau_{11}\} \vdash \mathrm{E}_{12}{:}\tau_{12}]\!](e \uplus \{\mathrm{I} = [\![\pi \vdash \mathrm{E}_2{:}\ \tau_2]\!]e\} \uplus \{\mathrm{J}{=}v\})$, since K does not appear in E_{12}. But this value is just $g(v)$, where $g = [\![\pi \uplus \{\mathrm{I}{:}\tau_2\} \vdash (\lambda \mathrm{J}{:}\tau_{11}.\, \mathrm{E}_{12}){:}\ \tau_{11} \rightarrow \tau_{12}]\!](e \uplus \{r{=}[\![\pi \vdash \mathrm{E}_2{:}\ \tau_2]\!]e\})$. □

It follows immediately that $[\![\pi \vdash ((\lambda \mathrm{I}{:}\tau_1.\, \mathrm{E}_2)\mathrm{E}_1){:}\, \tau_2]\!]e \;= [\![\pi \vdash [\mathrm{E}_1/\mathrm{I}]\mathrm{E}_2{:}\, \tau_2]\!]e$, and then a proof like that of Theorem 6.8 yields

6.12 Theorem

For $e \in Env_\pi$, $\pi \vdash \mathrm{E}_1{:}\,\tau$ *and* $\mathrm{E}_1 \Rightarrow^* \mathrm{E}_2$ *imply* $[\![\pi \vdash \mathrm{E}_1{:}\tau]\!]e = [\![\pi \vdash \mathrm{E}_2{:}\,\tau]\!]e$.

6.6 Lambda Calculus with Constants and Operators

To obtain facilities for repetition, arithmetic, and the like, we extend the simply typed lambda calculus by a set of constants and operators. For arithmetic, we might add:

$$\mathrm{E} \;::=\; \cdots \mid 0 \mid 1 \mid 2 \mid \cdots \mid (\mathit{plus}\ \mathrm{E}_1\ \mathrm{E}_2) \mid (\mathit{minus}\ \mathrm{E}_1\ \mathrm{E}_2)$$

along with the following typing rules:

$$\pi \vdash n{:}\, \mathit{int},\ \text{for all}\ n \geq 0 \qquad \frac{\pi \vdash \mathrm{E}_1{:}\,\mathit{int} \qquad \pi \vdash \mathrm{E}_2{:}\,\mathit{int}}{\pi \vdash (\mathit{plus}\ \mathrm{E}_1\ \mathrm{E}_2){:}\,\mathit{int}} \qquad \frac{\pi \vdash \mathrm{E}_1{:}\,\mathit{int} \qquad \pi \vdash \mathrm{E}_2{:}\,\mathit{int}}{\pi \vdash (\mathit{minus}\ \mathrm{E}_1\ \mathrm{E}_2){:}\,\mathit{int}}$$

Symbols like 0 and 1 that cannot be reduced are *constants*, and symbols like *plus* and *minus* that take arguments and can be reduced are *operators*. An expression that adds 1 to 3 and then subtracts 2 is written (*minus* (*plus* 1 3) 2). If we prefer, we can define operators as constants of function type, for example, $\pi \vdash \mathit{plus}{:}\, \mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{int}$ and code addition as ((*plus* 1) 3). The difference is often just a matter of style. In the terminology of Section 6.2, we say that 0, 1, 2, . . . , are the Values for the *int*-typed expressions; that is, *Value*(*int*) = {0, 1, 2, . . . }. This particular set is special, and we call it the set of *numerals.*

Operators have little use without rewriting rules. For example, we want to show that (*minus* (*plus* 1 3) 2) rewrites to (*minus* 4 2), which rewrites to 2. Here are the rewriting rules for *plus* and *minus*:

$(\mathit{plus}\ \ m\ \ n) \;\Rightarrow\; m{+}n$, where m and n are numerals,
and $m{+}n$ is the numeral that denotes the sum of m and n
$(\mathit{minus}\ \ m\ \ n) \;\Rightarrow\; m{-}n$, where m and n are numerals,
and $m{-}n$ is the numeral that denotes the difference of m and n

(For simplicity, we use natural numbers and natural number subtraction: If $n > m$, then $(\mathit{minus}\, m\, n) \Rightarrow 0$.) The two rules are call-by-value rules, like the ones in Section 6.2, and they should be read as abbreviating two families of rewriting rules. For example, the rule for *plus* abbreviates this family:

$(\mathit{plus}\ 0\ 0) \Rightarrow 0 \qquad (\mathit{plus}\ 1\ 1) \Rightarrow 2$
$(\mathit{plus}\ 0\ 1) \Rightarrow 1 \qquad (\mathit{plus}\ 1\ 2) \Rightarrow 3$
$(\mathit{plus}\ 1\ 0) \Rightarrow 1 \qquad \cdots$

Rules like the ones above are called δ-*rules*, and we use the term δ–*redex* for an expression that can be reduced by a δ-rule. General properties of δ-rules are given in Section 6.8; for now, we take on faith that properties like confluence and subject reduction hold for the simply typed lambda calculus extended by δ-rules. We choose the leftmost-outermost redex in an expression by picking that redex whose λ-symbol or operator lies to the leftmost in the expression.

We can also add booleans, where $\mathit{Value}(\mathit{bool}) = \{\mathit{true}, \mathit{false}\}$:

$$\pi \vdash \mathit{true}: \mathit{bool} \qquad \pi \vdash \mathit{false}: \mathit{bool} \qquad \frac{\pi \vdash \mathrm{E}: \mathit{bool}}{\pi \vdash (\mathit{not}\ \mathrm{E}): \mathit{bool}}$$

$$\frac{\pi \vdash \mathrm{E}_1: \mathit{int} \quad \pi \vdash \mathrm{E}_2: \mathit{int}}{\pi \vdash (\mathit{equals}\ \mathrm{E}_1\ \mathrm{E}_2): \mathit{bool}} \qquad \frac{\pi \vdash \mathrm{E}_1: \mathit{bool} \quad \pi \vdash \mathrm{E}_2: \tau \quad \pi \vdash \mathrm{E}_3: \tau}{\pi \vdash (\mathit{if}\ \mathrm{E}_1\ \mathrm{E}_2\ \mathrm{E}_3): \tau}$$

We use these δ-rules:

$(\mathit{equals}\ m\ m) \Rightarrow \mathit{true},$ where m is a numeral
$(\mathit{equals}\ m\ n) \Rightarrow \mathit{false},$ where m and n are different numerals
$(\mathit{not}\ \mathit{true}) \Rightarrow \mathit{false} \qquad (\mathit{not}\ \mathit{false}) \Rightarrow \mathit{true}$
$(\mathit{if}\ \mathit{true}\ \mathrm{E}_1\ \mathrm{E}_2) \Rightarrow \mathrm{E}_1 \qquad (\mathit{if}\ \mathit{false}\ \mathrm{E}_1\ \mathrm{E}_2) \Rightarrow \mathrm{E}_2$

Here is a leftmost-outermost reduction that uses the β-rule and the above δ-rules:

$(\lambda X{:}\mathit{bool}.\ \mathit{if}\ X\ (\mathit{minus}\ 2\ 1)\ (\mathit{plus}\ 3\ 4))\ (\mathit{equals}\ 0\ 1)$
$\Rightarrow (\mathit{if}\ (\mathit{equals}\ 0\ 1)\ (\mathit{minus}\ 2\ 1)\ (\mathit{plus}\ 3\ 4))$
$\Rightarrow (\mathit{if}\ \mathit{false}\ (\mathit{minus}\ 2\ 1)\ (\mathit{plus}\ 3\ 4)) \Rightarrow (\mathit{plus}\ 3\ 4) \Rightarrow 7$

We now add an operator and δ-rule to express repetition:

$$\frac{\pi \vdash \mathrm{E}: \tau \rightarrow \tau}{\pi \vdash (\mathit{fix}\ \mathrm{E}): \tau} \qquad (\mathit{fix}\ \mathrm{E}) \Rightarrow (\mathrm{E}\ (\mathit{fix}\ \mathrm{E}))$$

The δ-rule for *fix* has canceled the strong normalization property, since it is now possible to write expressions that have no normal forms. We do an example; let F name the following expression:

$\lambda \mathit{FAC}{:}\ \mathit{int} \rightarrow \mathit{int}.\lambda N{:}\ \mathit{int}.\ \mathit{if}\ (\mathit{equals}\ N\ 0)\ \ 1\ \ (\mathit{times}\ N\ (\mathit{FAC}\ (\mathit{minus}\ N\ 1)))$

Thus, $(\mathit{fix}\ F)$ is a representation of the factorial function. We put it to work on the argument 2 and do a leftmost-outermost reduction:

$$
\begin{aligned}
& (\mathit{fix}\ F)\ 2 \\
\Rightarrow\ & F\ (\mathit{fix}\ F)\ 2 \\
=\ & (\lambda \mathit{FAC}{:}\mathit{int}{\to}\mathit{int}.\lambda N{:}\mathit{int}.\ \mathit{if}\ (\mathit{equals}\ N\ 0)\ \ 1\ \ (\mathit{times}\ N\ (\mathit{FAC}\ (\mathit{minus}\ N\ 1))))\ (\mathit{fix}\ F)\ 2 \\
\Rightarrow\ & (\lambda N{:}\mathit{int}.\ \mathit{if}\ (\mathit{equals}\ N\ 0)\ \ 1\ \ (\mathit{times}\ N\ ((\mathit{fix}\ F)\ (\mathit{minus}\ N\ 1))))\ 2 \\
\Rightarrow\ & \mathit{if}\ (\mathit{equals}\ 2\ 0)\ \ 1\ \ (\mathit{times}\ 2\ ((\mathit{fix}\ F)\ (\mathit{minus}\ 2\ 1))) \\
\Rightarrow\ & \mathit{if}\ \mathit{false}\ \ 1\ \ (\mathit{times}\ 2\ ((\mathit{fix}\ F)\ (\mathit{minus}\ 2\ 1))) \\
\Rightarrow\ & \mathit{times}\ 2\ ((\mathit{fix}\ F)\ (\mathit{minus}\ 2\ 1)) \\
\Rightarrow\ & \mathit{times}\ 2\ ((F\ \ (\mathit{fix}\ F))\ (\mathit{minus}\ 2\ 1)) \\
=\ & \mathit{times}\ 2 \\
& \quad ((\lambda \mathit{FAC}{:}\mathit{int}{\to}\mathit{int}.\lambda N{:}\mathit{int}.\ \mathit{if}\ (\mathit{equals}\ N\ 0)\ \ 1\ \ (\mathit{times}\ N\ (\mathit{FAC}\ (\mathit{minus}\ N\ 1))))\ (\mathit{fix}\ F) \\
& \qquad (\mathit{minus}\ 2\ 1)) \\
\Rightarrow\ & \mathit{times}\ 2\ ((\lambda N{:}\mathit{int}.\ \mathit{if}\ (\mathit{equals}\ N\ 0)\ \ 1\ \ (\mathit{times}\ N\ ((\mathit{fix}\ F)\ (\mathit{minus}\ N\ 1))))(\mathit{minus}\ 2\ 1)) \\
\Rightarrow\ & \mathit{times}\ 2\ (\mathit{if}\ (\mathit{equals}\ (\mathit{minus}\ 2\ 1)\ 0)\ \ 1\ \ (\mathit{times}\ (\mathit{minus}\ 2\ 1) \\
& \quad ((\mathit{fix}\ F)\ (\mathit{minus}\ (\mathit{minus}\ 2\ 1)\ 1)))) \\
\Rightarrow\ & \mathit{times}\ 2\ (\mathit{if}\ (\mathit{equals}\ 1\ 0)\ \ 1\ \ (\mathit{times}\ (\mathit{minus}\ 2\ 1)\ ((\mathit{fix}\ F)\ (\mathit{minus}\ (\mathit{minus}\ 2\ 1)\ 1)))) \\
\Rightarrow\ & \mathit{times}\ 2\ (\mathit{if}\ \mathit{false}\ \ 1\ \ (\mathit{times}\ (\mathit{minus}\ 2\ 1)\ ((\mathit{fix}\ F)\ (\mathit{minus}\ (\mathit{minus}\ 2\ 1)\ 1)))) \\
\Rightarrow\ & \mathit{times}\ 2\ (\mathit{times}\ (\mathit{minus}\ 2\ 1)\ ((\mathit{fix}\ F)\ (\mathit{minus}\ (\mathit{minus}\ 2\ 1)\ 1))) \\
\Rightarrow\ & \mathit{times}\ 2\ (\mathit{times}\ 1\ ((\mathit{fix}\ F)\ (\mathit{minus}\ (\mathit{minus}\ 2\ 1)\ 1))) \\
\Rightarrow\ & \mathit{times}\ 2\ (\mathit{times}\ 1\ ((F\ (\mathit{fix}\ F))\ (\mathit{minus}\ (\mathit{minus}\ 2\ 1)\ 1)))
\end{aligned}
$$

and so the reduction goes. You are invited to perform the remaining stages of rewriting; the Value uncovered is 2.

Since the example used leftmost-outermost reduction, a number of numerical expressions, like (*minus* 2 1), were copied and reduced several times. We will address this efficiency question momentarily.

The β- and δ-rules for the arithmetic language form an operational semantics, since the rules show how to compute an arithmetic expression to a Value. But the specific strategy of applying the rewriting rules (for example, leftmost-outermost, rightmost-innermost) is unspecified by the rules themselves. The confluence property suggests that the order of reductions should not be important, but efficiency questions and the possibility of nontermination means that it is. In the above example, the rightmost-innermost reduction strategy might be tried to get a more efficient reduction sequence, but the reduction of $(\mathit{fix}\,F)2$ does not terminate. (Try it.)

Ideally, we should use a set of rewriting rules that are insensitive to reduction strategy, but this is rarely achievable. In the above example, if the efficiency of the reduction sequence is an issue, the β-rule can be replaced by the β-*val* rule. Since the numerals are the only *int*-typed phrases that are Values, the β-*val* rule would force $(\lambda N{:}int.\cdots N\cdots)(minus\,2\,1)$ to reduce to $(\lambda N{:}int.\cdots N\cdots)1$ before it reduces to $\cdots 1\cdots$. But the β-*val* rule does not interact well with the previously stated rewriting rule for *fix*. (Try it on the above example.) A variant of *fix* that is sometimes used with the β-*val* rule is

$$fix'\,E \;\Rightarrow\; E\,(\lambda I{:}\tau_1.\, fix'\,E\,I)$$

where E must be typed $(\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)$. Another possiblity is to use the *rec* operator and its rule from Chapter 2:

$$rec\,I{:}\tau.\,E \;\Rightarrow\; [(rec\,I{:}\tau.\,E)/I]E$$

A consequence of the latter rule is, either $rec\,I{:}\tau.\,E$ must be a Value or else the rule β-val_1 must be dropped because identifiers no longer stand for Values.

When the call-by-value rules are used for computation, the strategy for reduction changes from normal form discovery to Value discovery. In particular, reductions should not be performed within a lambda abstraction, since a lambda abstraction is already a Value. For the call-by-value reduction rules in this section, a leftmost-outermost strategy that does not reduce redexes within lambda abstractions always discovers Values, if they exist.

If the call-by-value rules are not desired, there is an implementation technique for the β-rule, known as *call-by-need*, which gives efficiency at least as good as the β-*val* rule. For a call-by-need implementation, a phrase is implemented as a graph, and the implementation of the β-rule upon $(\lambda I.E_1)E_2$ replaces all occurrences of I in E_1 by a pointer to E_2's subgraph. Since all occurrences of I point to the same subgraph, whenever any occurrence of I is computed, the sharing implies that all occurrences are computed. Details are given in Wadsworth 1971 and Peyton-Jones 1987.

6.7 Operational Semantics for a Source Language

Section 1.8 stated that a programming language's denotational semantics can do double duty as an operational semantics. In this section, we justify why this is so and demonstrate that such an operational semantics is a simply typed lambda calculus with constants and operators. The development is pedantic and can be skipped if the reader accepts these claims.

The metalanguage for the operational semantics definition is the simply typed lambda calculus with integers, booleans, locations, and stores. Integers and booleans were presented in the previous section. Locations are the usual: $loc_i : intloc$, for $i \geq 0$, and $Value(intloc) = \{\, loc_i \mid i \geq 0 \,\}$. There are no operations upon locations. The store type has these constructions:

$$\frac{\pi \vdash E_1 : \mathit{int} \quad \pi \vdash E_2 : \mathit{int} \quad \cdots \quad \pi \vdash E_n : \mathit{int}}{\pi \vdash \langle E_1, E_2, \ldots, E_n \rangle : \mathit{store}}, \text{ for } n \geq 0$$

$$\frac{\pi \vdash E_1 : \mathit{intloc} \quad \pi \vdash E_2 : \mathit{store}}{\pi \vdash (\mathit{lookup}\ E_1\ E_2) : \mathit{int}} \qquad \frac{\pi \vdash E_1 : \mathit{intloc} \quad \pi \vdash E_2 : \mathit{int} \quad \pi \vdash E_3 : \mathit{store}}{\pi \vdash (\mathit{update}\ E_1\ E_2\ E_3) : \mathit{store}}$$

and $\mathit{Value}(\mathit{store}) = \{\langle n_1, n_2, \ldots, n_m \rangle \mid m \geq 0 \text{ and for all } 1 \leq i \leq m,\ n_i \in \mathit{Value}(\mathit{int})\}$. As usual, a store like $\langle 3, 1, 7 \rangle$ defines a store that has 3 in loc_1's location, 1 in loc_2's location, and 7 in loc_3's location. The rewriting rules for the store operators are the expected ones:

$$(\mathit{lookup}\ loc_i\ \langle n_1, n_2, \ldots, n_i, \ldots, n_m \rangle) \Rightarrow n_i$$
$$(\mathit{update}\ loc_i\ n\ \langle n_1, n_2, \ldots, n_i, \ldots, n_m \rangle) \Rightarrow \langle n_1, n_2, \ldots, n, \ldots, n_m \rangle$$

The arguments to *lookup* and *update* must be Values for the reductions to occur.

Now, we are ready to define the operational semantics of the source programming language; we use the typing rules in Figure 5.1 plus these two:

$$\frac{E_1 : \tau\mathit{exp} \quad \pi \vdash E_2 : \mathit{store}}{\pi \vdash [\![E_1 : \tau\mathit{exp}]\!]\, E_2 : \tau} \qquad \frac{E_1 : \mathit{comm} \quad \pi \vdash E_2 : \mathit{store}}{\pi \vdash [\![E_1 : \mathit{comm}]\!]\, E_2 : \mathit{store}}$$

The semantics definition is a set of rewriting rules:

$$[\![E_1 := E_2 : \mathit{comm}]\!]s \Rightarrow (\mathit{update}\ E_1\ ([\![E_2 : \mathit{intexp}]\!]s)\ s)$$
$$[\![E_1 ; E_2 : \mathit{comm}]\!]s \Rightarrow [\![E_2 : \mathit{comm}]\!]([\![E_1 : \mathit{comm}]\!]s)$$
$$[\![\textbf{while}\ E_1\ \textbf{do}\ E_2\ \textbf{od} : \mathit{comm}]\!]s$$
$$\Rightarrow \mathit{if}\ ([\![E_1 \mathit{boolexp}]\!]s)\ ([\![\textbf{while}\ E_1\ \textbf{do}\ E_2\ \textbf{od} : \mathit{comm}]\!]([\![E_2 : \mathit{comm}]\!]s))\ s$$
$$\ldots$$
$$[\![E_1 + E_2 : \mathit{intexp}]\!]s \Rightarrow (\mathit{plus}\ ([\![E_1 : \mathit{intexp}]\!]s)\ ([\![E_2 : \mathit{intexp}]\!]s))$$
$$[\![@L : \mathit{intexp}]\!]s \Rightarrow (\mathit{lookup}\ L\ s)$$
$$\ldots$$

where, for all of the above rules, s is a Value

Computations with the rewriting rules proceed like the calculations seen in earlier chapters.

Since the metalanguage is the simply typed lambda calculus, we can reformat the above rules with lambda abstractions and compute with the β-*val* rule:

$$[\![E_1{:=}E_2{:}\mathit{comm}]\!] \;\Rightarrow\; \lambda s{:}\mathit{store}.\,(\mathit{update}\; E_1\; ([\![E_2{:}\mathit{intexp}]\!]s)\; s)$$
$$[\![E_1;E_2{:}\mathit{comm}]\!] \;\Rightarrow\; \lambda s{:}\mathit{store}.\,[\![E_2{:}\mathit{comm}]\!]([\![E_1{:}\mathit{comm}]\!]s)$$
$$\ldots$$

Now, the previous two typing rules are unneeded, because we can make the equivalences $\mathit{comm} \equiv \mathit{store} \rightarrow \mathit{store}$ and $\tau\mathit{exp} \equiv \mathit{store} \rightarrow \tau$.

The next step is to add abstractions, parameters, and blocks to the source programming language. If we follow the lines of Chapter 5, then we add records and lambda abstraction to the source programming language, creating a simply typed lambda calculus of it. We focus upon lambda abstractions here, leaving records as an exercise. Once we add lambda abstractions to the source language, we may add the β-rule for rewriting them. The result is a *substitution semantics*, so named because the semantics of lambda abstraction in the source language is understood by means of syntactic substitution, as defined by the β-rule. In the terminomolgy of Chapter 2, a substitution semantics applies the copy rule to a program with lambda abstractions, reducing the program into one in the core language, and then uses the semantics of the core language to calculate the meaning.

Substitution semantics works fine, but insight can be gained from an *environment semantics*, where an environment argument and its operations are added to the metalanguage. The rewriting rules for the language are altered to include environments:

$$[\![\pi \vdash E_1{:=}E_2{:}\mathit{comm}]\!]$$
$$\Rightarrow\; \lambda e{:}\mathit{env}.\lambda s{:}\mathit{store}.\,(\mathit{update}\; ([\![\pi \vdash E_1{:}\mathit{intloc}]\!]e)\; ([\![\pi \vdash E_2{:}\mathit{intexp}]\!]e\, s)\; s)$$
$$[\![\pi \vdash E_1;E_2{:}\mathit{comm}]\!] \;\Rightarrow\; \lambda e{:}\mathit{env}.\lambda s{:}\mathit{store}.\,[\![\pi \vdash E_2{:}\mathit{comm}]\!]e\,([\![\pi \vdash E_1{:}\mathit{comm}]\!]e\, s)$$
$$\ldots$$
$$[\![\pi \vdash \lambda I{:}\theta_1.\,E{:}\,\theta_1 \rightarrow \theta_2]\!] \;\Rightarrow\; \lambda e{:}\mathit{env}.\lambda u{:}\theta_1.\,[\![\pi \uplus \{I{:}\theta_1\} \vdash E{:}\,\theta_2]\!](\mathit{bind}\; I\; u\; e)$$
$$[\![\pi \vdash E_1 E_2{:}\,\theta_2]\!] \;\Rightarrow\; \lambda e{:}\mathit{env}.\,([\![\pi \vdash E_1{:}\,\theta_1 \rightarrow \theta_2]\!]e)\;([\![\pi \vdash E_2 \theta_1]\!]e)$$
$$[\![\pi \vdash I{:}\,\theta]\!] \;\Rightarrow\; \lambda e{:}\mathit{env}.\,\mathit{find}\; I\; e$$

bind and *find* are operations that insert and look up bindings in the environment; their definitions are left as exercises. A technical point is that the β-rule can be applied to the source language lambda abstractions, or the β-*val* rule can be applied to the metalanguage lambda abstractions and the end result is the same. In earlier chapters, this was called soundness of the copy rules; here, it becomes a confluence result.

6.8 Subtree Replacement Systems

When a lambda calculus is extended by δ-rules, one must verify that confluence, standardization, and other properties are preserved. The general version of a lambda calculus-like system is called a *subtree replacement system* (*SRS*).[*] In this and the next section, we study subtree replacement systems and state general properties that imply confluence and standardization. These properties can be used to verify that an operational semantics for a programming language is worthwhile.

Say that a language, L, is defined by a single syntax rule:

$$L ::= construction_1 \mid \cdots \mid construction_n, \quad n > 0.$$

Languages defined by multiple syntax rules can also be studied, but it is simplest to work with just one rule. Next, let the syntax rule be augmented by a set of *variables* $V = \{X, Y, Z, \ldots\}$

$$L_V ::= construction_1 \mid \cdots \mid construction_n \mid V$$

and let L_V be the language that results. L_V is the language of *polynomials of L*. That is, a polynomial of L is an L-expression with zero or more variables in it. For $lhs, rhs \in L_V$, we say that $lhs \Rightarrow rhs$ is a *rewriting rule* for L if *lhs* and *rhs* are polynomials of L and every variable that appears in *rhs* also appears in *lhs*. The rewriting rule is *linear* if no variable appears in *lhs* more than once.

6.13 Definition

A *(linear) subtree replacement system (SRS)* is a pair, (L, R), where L is a syntax rule and R is a set of linear rewriting rules for L.

An example of an SRS follows:

$$\begin{aligned}&(\,\mathrm{E} ::= true \mid false \mid not\ \mathrm{E} \mid if\ \mathrm{E}_1\ \mathrm{E}_2\ \mathrm{E}_3,\\ &\quad \{not\ true \Rightarrow false,\ not\ false \Rightarrow true,\ if\ true\ X\ Y \Rightarrow X,\ if\ false\ X\ Y \Rightarrow Y\}\,)\end{aligned}$$

For simplicity, we work with systems whose operators are in prefix form.

The systems are called subtree replacement systems because the expressions are trees. We draw the trees so that the operators are at the roots. An example is

```
       if
   /   |   \
not  not  true
|      |
true not
       |
       false
```

[*] Another name is *term rewriting system*, but we wish to emphasize the tree-like structure of the phrases in the system.

which depicts *if (not true) (not (not false)) true*. A polynomial expression is an incomplete tree, where variables mark the places where subtrees need insertion. Rewriting rules are tree transformers: When a tree matches the pattern defined by the left-hand side of a rewriting rule, the variables in the left-hand side mark subtrees in the tree, and the marked subtrees are used to build a new tree with the structure described by the right-hand side of the rule. These notions can be formalized.

A *substitution*, denoted by σ, is a set of variable, polynomial pairs. The application of a substitution, σ, to a polynomial, E, written σE, is the replacement of all occurrences of X in E by polynomials, E$'$, for all $(X, E') \in \sigma$. An example, in terms of the SRS above, is $\sigma = \{(X, \mathit{true}), (Z, (\mathit{not}\ Y))\}$, $E = \mathit{if}\ \mathit{true}\ X\ (\mathit{not}\ Z)$, and $\sigma E = \mathit{if}\ \mathit{true}\ \mathit{true}\ (\mathit{not}\ (\mathit{not}\ Y))$. An expression, E, *matches* a polynomial, p, if there is a substitution, σ, such that $\sigma p = E$. An expression is a *redex* if it matches *lhs* of a rewriting rule $\mathit{lhs} \Rightarrow \mathit{rhs}$, via a substitution σ; the *contractum* is the expression $\sigma \mathit{rhs}$. The replacement of the redex by its contractum is a *contraction* or *reduction.* These notions are the same as the ones used with the lambda calculus, and we continue to use terms such as ‘‘normal form,’’ ‘‘reduction sequence,’’ and so on.

Given a polynomial, p, we say that its *pattern* is p less its variables. For example, the pattern of $\mathit{if}\ \mathit{true}\ X\ Y$ is $\mathit{if}\ \mathit{true}$ [ ] [ ], where the brackets mark the missing variables. The pattern of a polynomial shows the structure necessary for an expression to match the polynomial. If an expression, E, is a redex by means of a rewriting rule, $\mathit{lhs} \Rightarrow \mathit{rhs}$, we say that the *pattern of the redex* is that subpart of E that matches the pattern of *lhs*. For example, for redex *if true false true*, the pattern of the redex is *if true* [ ] [ ].

Recall the *confluence property*: For all E, if $E \Rightarrow^* E_1$ and $E \Rightarrow^* E_2$, then there is some E_3 such that $E_1 \Rightarrow^* E_3$ and $E_2 \Rightarrow^* E_3$. Verifying that an arbitrary linear SRS has confluence is undecidable, so we will state a criterion that is sufficient for confluence. The notion behind the criterion is that rewriting rules should not ‘‘interfere’’ with one another.

Consider an SRS that contains this pair of rules:

$$f\ (g\ X)\ Y \Rightarrow Y$$
$$g\ a \Rightarrow a$$

The expression $f\ (g\ a)\ a$ can be reduced by the first rule to a and by the second rule to $f\ a\ a$. Can a and $f\ a\ a$ be reduced to the same expression? This cannot be done with the two rules here, so the question of confluence depends upon the other rules of the SRS. This situation is dangerous, and the danger arises because the two rules interfere with one another—reducing a redex by the second rule alters a redex by the first rule. In contrast, the rules

$$f\ X\ Y \Rightarrow Y$$
$$g\ a \Rightarrow a$$

do not interfere with each other, in the sense that, if an expression contains a redex by the first rule and a redex by the second rule, the two redexes can be reduced in either order, and the end result is the same. For example, $f\ a\ (g\ a) \Rightarrow g\ a$ and also $f\ a\ (g\ a) \Rightarrow f\ a\ a$, but then $g\ a \Rightarrow a$ and $f\ a\ a \Rightarrow a$. For the expression $f\ (g\ a)\ a$, we see that $f\ (g\ a)\ a \Rightarrow a$ and also $f\ (g\ a)\ a \Rightarrow f\ a\ a \Rightarrow a$. Also, a rewriting rule can interfere with itself; for $f\ (f\ X) \Rightarrow a$, and the expression $f\ (f\ (f\ a))$, we see that the expression can be reduced in two different ways that cannot be reconciled: $f\ (f\ (f\ a)) \Rightarrow f\ a$ and $f\ (f\ (f\ a)) \Rightarrow a$.

A precise definition of the above intuition is: A pair of rules $(lhs_1 \Rightarrow rhs_1,\ lhs_2 \Rightarrow rhs_2)$ *interfere* if there is a substitution σ such that σlhs_1 contains σlhs_2 as a subtree and the pattern of σlhs_2 overlaps the pattern of σlhs_1. For the substitution, σ, to be meaningful, we assume that the variables in the two rules are distinct (else we rename the variables to make them distinct). Also, since every rewriting rule trivially ''interferes'' with itself, we ignore trivial self interference.

In the first example above, the substitution $\sigma = \{(X,a),\ (Y,a)\}$ shows that the rules interfere. No substitution can be found for the second example, because the rules do not interfere. In the final example, after renaming the variable in the second occurrence of the rule, we find that the substitution $\sigma = \{(X,\ f\ a),\ (Y,\ a)\}$ makes the rule pair $(\ f\ (f\ X) \Rightarrow a,\ \ f\ (f\ Y) \Rightarrow a\ )$ interfere.

In practice, it is easy to see when a pair of rules interfere: Build a tree where the patterns of the left-hand sides of the two rules overlap in the tree. In the first example, such a tree has form $f\ (g\ a)$ E, and in the third example the tree has form $f\ (f\ (f$ E)).

6.14 Definition

A linear SRS is *orthogonal* if no pair of its rewriting rules interfere.

Orthogonal systems are common and natural; the δ-rule sets in Section 6.6 are examples. The main result of this section is

6.15 Theorem

Every orthogonal SRS has the confluence property.

A core programming language is typically an orthogonal SRS, and the addition of lambda abstraction preserves orthogonality, so Theorem 6.15 implies that the languages we develop have semantics definitions that make the rewriting rules sound.

The remainder of this section is devoted to a proof of Theorem 6.15, and if you are disinterested in its development, you may proceed to the next section.

A key notion is that of *residual*. Let R_1 and R_2 be redexes in an expression, E, and say that R_1 is reduced, which we write as: $\mathrm{E} \overset{R_1}{\Rightarrow} \mathrm{E}'$. The residuals of R_2 are those copies of R_2, if any, remaining in E′. We formalize this as

6.16 Definition

For redexes R_1 and R_2 in E, and $E \overset{R_1}{\Rightarrow} E'$, *the residuals of* R_2 in E', written $res(R_2)$, are defined as

(i) If R_1 are R_2 are disjoint subtrees in E, then $res(R_2)$ is the single copy of R_2 in E'.

(ii) If R_1 is within R_2 in E, and the patterns of R_1 and R_2 are disjoint, then $res(R_2)$ is the single copy of R_2 in E'. (But note that the subpart of R_2 that contained R_1 is changed.)

(iii) If R_2 is within R_1 in E, and the patterns of R_1 and R_2 are disjoint, then $res(R_2)$ are those copies, if any, of R_2 in E' that were propagated by the rewriting rule.

(iv) If the patterns of R_1 and R_2 overlap, then $res(R_2)$ in E' is empty.

For example, for rules $f\ X\ Y \Rightarrow h\ Y\ Y$ and $g\ X \Rightarrow a$, expressions $E = f\ (g\ a)\ (f\ (g\ b)\ (g\ c))$ and $E' = f\ (g\ a)\ (h\ (g\ c)\ (g\ c))$, and reduction step $E \Rightarrow E'$, the residual of $g\ a$ is its single copy in E', $g\ b$ has no residuals, $g\ c$ has two residuals in E', $f\ (g\ b)\ (g\ c)$ has none (why?), and the residual of E in E' is E' itself.

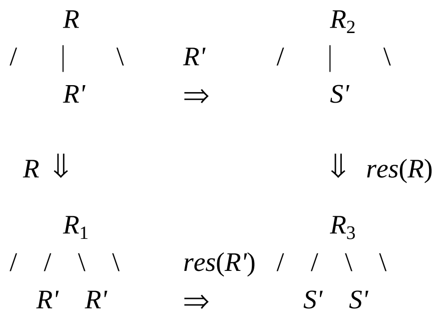
A residual of a redex is itself a redex, and after several reduction steps, we say that the *descendants* of a redex are the residuals of the residuals of . . . the redex, that is, the ‘‘residual’’ definition is extended by transitivity. For example, for expression E above, the descendants of $g\ c$ in E_3, where

$$E = f\ (g\ a)\ (f\ (g\ b)\ (g\ c)) \Rightarrow f\ (g\ a)\ (h\ (g\ c)\ (g\ c))$$
$$\Rightarrow h\ (h\ (g\ c)\ (g\ c))\ (h\ (g\ c)\ (g\ c)) \Rightarrow h\ (h\ a\ (g\ c))\ (h\ (g\ c)\ (g\ c)) = E_3$$

are the three occurrences of $g\ c$ in E_3.

An excellent way to track the descendants of a redex is by ‘‘coloring’’ them: If we wish to monitor the descendants of a redex in a reduction sequence, we color the pattern of the original redex with a red pencil, and at each reduction step, any redexes with red-colored patterns that are copied into the next expression in the sequence are copied in red. (If the reduction step contracts a redex with a red pattern, the pattern of the contractum is not colored red.) In the above example, if we wish to monitor the residuals of $g\ c$ in $E \Rightarrow^* E_3$, we color the ‘‘g’’ part of $g\ c$ in E with red. Each reduction step copies red patterns into red patterns, so it is easy to locate the three residuals in E_3 (and note that the a in E_3 is not colored red).

An important game to play is reduction-of-colored-redexes-only. Given expression E, say that we color the patterns of some of the redexes in E and then generate a reduction sequence where only colored redexes are reduced. Stated formally, a *complete development* of a set of redexes S in an expression E is a reduction sequence that at each stage reduces a descendant of a redex in S until no more descendants exist. An intuitive and useful result is



where S' denotes the contractum of R'

Figure 6.3

6.17 Proposition

Every complete development must terminate.

Proposition 6.17 lets us write $E \Rightarrow^{S} E'$ to denote a complete development of *S*. When *S* is a singleton set, $\{R\}$, we write $E \Rightarrow^{R} E'$, as usual.

An obvious question to ask is: Does the order of reduction in a complete development influence the final result? We will see that the answer is ‘‘no,’’ and the easiest way to get to the answer is by way of the proof of confluence. Here is the central property in our proof of confluence:

6.18 Definition

A subtree replacement system has the *closure property* if, for every redex R that contains another redex R', if $R \Rightarrow^{R} R_1$ and $R \Rightarrow^{R'} R_2$, then there exists some R_3 such that $R_1 \Rightarrow^{res(R')} R_3$ and $R_2 \Rightarrow^{res(R)} R_3$.

Figure 6.3 pictures the closure property. Closure formalizes the idea that rule pairs should not interfere with each other. You are left with the straightforward (but crucial) exercise of showing that every orthogonal SRS has the closure property.

Now, we show how confluence follows from closure. Say that an SRS has the *parallel moves property* if, for redex sets S_1 and S_2, $E \Rightarrow^{S_1} E_1$ and $E \Rightarrow^{S_2} E_2$ imply that there exists some E_3 such that $E_1 \Rightarrow^{res(S_2)} E_3$ and $E_2 \Rightarrow^{res(S_1)} E_3$. That is, the parallel moves property generalizes closure to sets of redexes. Momentarily, we show that closure and the parallel moves property are equivalent, but first we go to the main goal:

$$\begin{tikzcd}
E \arrow[r, Rightarrow] \arrow[d, Rightarrow] & F_1 \arrow[r, Rightarrow] \arrow[d, Rightarrow, "*"] & F_2 & \cdots \arrow[r, Rightarrow] & F_{n-1} \arrow[r, Rightarrow] & F_n \\
 & & \vdots & & & \vdots \\
E_1 \arrow[r, Rightarrow, "*"] \arrow[d, Rightarrow] & E_{11} \arrow[d, Rightarrow, "*"] & \cdots \\
E_2 \arrow[r, Rightarrow, "*"] & E_{21} \\
\vdots & \vdots \\
E_{m-1} \arrow[d, Rightarrow] & \cdots \\
E_m & \cdots
\end{tikzcd}$$

Figure 6.4

6.19 Theorem

If an SRS has the parallel moves property, then it has confluence.

Proof: Our job is to ‘‘tile the plane’’ bounded by E, E_m, and F_n at the corners, where $E \Rightarrow E_1 \Rightarrow E_2 \Rightarrow \cdots \Rightarrow E_m$ and $E \Rightarrow F_1 \Rightarrow F_2 \Rightarrow \cdots \Rightarrow F_n$, $m, n \geq 0$. See Figure 6.4. Each square in the figure will be filled by a ‘‘tile’’ created by the parallel moves property. For example, we could tile the plane column by column; the first two tiles of the first column would be (1) the tile whose edges are $E \Rightarrow E_1$, $E \Rightarrow F_1$, $E_1 \Rightarrow^* E_{11}$, and $F_1 \Rightarrow^* E_{11}$, where E_{11} is the name of the new corner; (2) the tile whose edges are $E_1 \Rightarrow E_2$, $E_1 \Rightarrow^* E_{11}$, $E_2 \Rightarrow^* E_{21}$, and $E_{11} \Rightarrow^* E_{21}$, where E_{21} is the new corner. The rest of the first column uses tiles similar to that of tile (2). Each tile has a format that matches the parallel moves property. The second (and subsequent) columns are covered with tiles of the form: $E_{ij} \Rightarrow^* E_{(i+1)j}$, $E_{ij} \Rightarrow^* E_{i(j+1)}$, $E_{(i+1)j} \Rightarrow^* E_{(i+1)(j+1)}$, and $E_{i(j+1)} \Rightarrow^* E_{(i+1)(j+1)}$. □

To show the last, missing result, that closure and the parallel moves property are equivalent, we must introduce yet another version of confluence. An SRS is *weakly confluent* if $E \Rightarrow E_1$ and $E \Rightarrow E_2$ implies that there is some E_3 such that $E_1 \Rightarrow^* E_3$ and $E_2 \Rightarrow^* E_3$. Weak confluence looks like confluence, but it is indeed weaker; the following system is weakly confluent but *not* confluent:

$a \Rightarrow b$	$a \Rightarrow c$
$b \Rightarrow a$	$b \Rightarrow d$

The counter example to confluence is $a \Rightarrow^* c$ and $a \Rightarrow^* d$. But an important result is

6.20 Lemma

If an SRS is strongly normalizing (that is, it has no infinite reduction sequences) and weakly confluent, then it is confluent.

Proof: Our proof of this result is somewhat informal, but it gives the main intuition. (A precise proof is in Huet 1980.) As in the proof of Theorem 6.18, we tile the plane bounded by E, E_m, and F_n at the corners, where $E \Rightarrow E_1 \Rightarrow E_2 \Rightarrow \cdots \Rightarrow E_m$ and $E \Rightarrow F_1 \Rightarrow F_2 \Rightarrow \cdots \Rightarrow F_n$, $m, n \geq 0$. Clearly, we can lay the first tile, which is bounded at its corners by E, E_1, F_1, and there is some E_{11p}, $p \geq 0$, such that $E_1 \Rightarrow E_{111} \Rightarrow E_{112} \Rightarrow \cdots \Rightarrow^* E_{11p}$ and $F_1 \Rightarrow^* E_{11p}$. Now we wish to lay a tile underneath the first one, but we need p tiles, not just one! We can lay the p tiles, but to tile underneath them, we will need $q_{11} \times q_{12} \times \cdots \times q_{1p}$ tiles, where each q_{1i} represents the number of tiles needed underneath the ith tile in the row above. Figure 6.5 pictures the situation. There is danger of infinite regress: Perhaps the tiling underneath the initial tile can never complete because an infinite number of tiles are ultimately needed. But this would imply the existence of an infinite reduction sequence, which is impossible, since the SRS is strongly normalizing. Hence, the tiling, as tedious as it is, must complete. □

We use Lemma 6.20 to prove the last result:

6.21 Lemma

An SRS has closure iff it has the parallel moves property.

Proof: The ‘‘if’’ part is easy: A system with the parallel moves property has closure, because closure is just the parallel moves property with redex sets that are singleton sets. For the ‘‘only if’’ part, assume that the SRS has closure. We build a new, ‘‘colored SRS’’ such that its rewriting rules reduce redexes only when they are colored red. By Proposition 6.17, the colored SRS has the strong normalization property. By the closure

$$\begin{tikzcd}
E \arrow[rrrr, Rightarrow] \arrow[d, Rightarrow] & & & & F_1 \arrow[d, Rightarrow, "*"] \\
E_1 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & E_{111} \arrow[r, Rightarrow] \arrow[d, Rightarrow, "*"] & E_{112} \arrow[r, Rightarrow] & \cdots \arrow[r, Rightarrow] & E_{11p} \\
E_2 \arrow[r, Rightarrow, "\Rightarrow \cdots"] & E_{21q} & & & \\
\vdots & & & &
\end{tikzcd}$$

Figure 6.5

property, the colored SRS is weakly confluent, since the residuals of a colored redex are always colored. By Lemma 6.19, the SRS is confluent. But confluence in the colored SRS is just the parallel moves property in the original SRS, when the redexes in the sets S_1 and S_2 are exactly the ones colored red. $\square$

The ‘‘only if’’ part of Lemma 6.21 is sometimes called the ‘‘parallel moves lemma,’’ hence our use of the term ‘‘parallel moves property.’’

Perhaps you have noted that the (untyped or simply typed) lambda calculus is not an SRS; in particular, the β-rule does *not* fit the format for a rewriting rule. The problem is, as usual, substitution. The closest we can come to formalizing the β-rule as a rewriting rule is

$$((\lambda \mathrm{I}.\ \cdots \mathrm{I} \cdots \mathrm{I} \cdots)\ Y) \ \Rightarrow\ \cdots Y \cdots Y \cdots$$

but neither the left-hand nor the right-hand side of the rule is a polynomial expression. Apparently, what is needed is a form of variable that can match a tree whose leaves are labeled by free occurrences of an identifier, I. We write such a variable as ${}_{\mathrm{I}}\mathbf{X}$ and code the β-rule as

$$((\lambda \mathrm{I}.\ {}_{\mathrm{I}}\mathbf{X})\ Y) \ \Rightarrow\ {}_{Y}\mathbf{X}$$

where ${}_{Y}\mathbf{X}$ represents the tree that matches ${}_{\mathrm{I}}\mathbf{X}$ when its free occurrences of I are replaced by Y. (Note: We blithely ignore issues related to the clash of free and bound identifiers!) A variable like **X** is sometimes called a ‘‘higher-order variable,’’ since it uses arguments like I and Y.

Now, the pattern of a β-redex is $((\lambda \mathrm{I}.\ {}_{\mathrm{I}}[\ ])\ [\ ])$; that is, the pattern includes the combination, the λI., and the free occurrences of I. We see that patterns of nested β-redexes can intertwine, but they cannot overlap, and in this sense the lambda calculus is an orthogonal SRS. A crucial feature of this sense of orthogonality is that an outer β-redex remains a redex even if an inner redex is contracted and some of the outer redex’s free identifiers are replicated or deleted.

An orthogonal SRS without lambda abstraction can be augmented by lambda abstraction and the β-rule, and the result is an orthogonal system. (This assumes that the syntax of lambda abstraction, combinations, and identifiers is new to the SRS; otherwise, patterns of rewriting rules might overlap.) Also, the lambda calculus with the β-*val* rule can be seen as orthogonal, since β-*val* can be written as a family of orthogonal rules, one rule for each form of Value, for example,

$(\lambda \mathrm{I}.{}_{\mathrm{I}}\mathbf{X})\ n \ \Rightarrow\ {}_{n}\mathbf{X}$, where n is a numeral
$(\lambda \mathrm{I}.{}_{\mathrm{I}}\mathbf{X})\ \mathit{true} \ \Rightarrow\ {}_{\mathit{true}}\mathbf{X}$
$(\lambda \mathrm{I}.{}_{\mathrm{I}}\mathbf{X})\ (\lambda \mathrm{J}.Y) \ \Rightarrow\ {}_{(\lambda \mathrm{J}.Y)}\mathbf{X}$
. . .

We will not formalize higher-order variables here; details can be found in Klop 1980 and Khasidashvili 1993. But it is worthwhile to ponder their origin. Identifiers, as used in the lambda calculus, are not ordinary phrases—they are placeholders, or literally, ‘‘hole labels,’’ because they label holes in a phrase where other phrases should be inserted. Imagine a phrase that contains a number of holes; it should be possible to match that phrase to a variable that remembers the locations of the holes. This is the idea behind the variable, $_{\mathrm{I}}\mathbf{X}$, where I marks the holes.

If we develop this idea, we naturally represent a phrase, E, with holes in it as $_{\mathrm{I}}\mathrm{E}$ (the traditional representation is (I)E) to state that E has holes and they are labeled by I. The purpose of the λ is to delimit or bind together the holes, so that when one hole is filled by a phrase, all of them are filled simultaneously by the same phrase. We write λ(I)E, a lambda abstraction, to do this. This suggests there might be additional delimiters for holes besides λ; the idea is explored in Chapters 8–10.

6.9 Standardization

A second major property of an SRS is strong normalization. Unfortunately, strong normalization is undecidable for an arbitrary SRS, and many natural systems lack the property, anyway. So, we concentrate on a standardization property, that is, we demonstrate a rewriting strategy that always finds an expression’s normal form, if one exists.

We say that a redex, R, in an expression, E, is *outermost* if R is not properly contained in another redex within E. Notice that the residual of an outermost redex might not itself be outermost; for the rules

$f\ X\ a \Rightarrow a$

$b \Rightarrow a$

$c \Rightarrow a$

the expression $f\ c\ b$ contains c as an outermost redex, but the reduction of redex b, giving $f\ c\ a$, means that c’s residual is not outermost.

An outermost redex, R, in E is *eliminated* by a reduction $\mathrm{E} \Rightarrow \mathrm{E}'$ if either (i) R has no residual in E, or (ii) R’s residual is not outermost in E′. Notice that an outermost redex might be eliminated only after several reduction steps; for example, the (descendant of the) outermost redex b in the expression $f\ c\ b$ is eliminated in two reduction steps in this sequence: $f\ c\ b \Rightarrow f\ a\ b \Rightarrow f\ a\ a \Rightarrow a$. A reduction sequence is called *eventually outermost* if every outermost redex that appears in some expression in the reduction sequence is eventually eliminated. The previous example is an eventually outermost reduction sequence.

6.22 Theorem

If an SRS is orthogonal, then if an expression E has a normal form, then any eventually outermost reduction sequence starting from E will find it.

So, a proper rewriting strategy for an orthogonal SRS is one that computes eventually outermost reduction sequences. One such strategy, called *parallel outermost*, reduces all the outermost redexes in an expression at once, in parallel. Often the leftmost-outermost strategy computes eventually outermost reduction sequences—as it did in the previous section—but this is not guaranteed. Consider these rewriting rules:

$$if\ X\ Y\ true \Rightarrow X$$
$$loop \Rightarrow loop$$
$$not\ false \Rightarrow true$$

and the expression $if\ a\ loop\ (not\ false)$. The leftmost-outermost reduction strategy is inadequate in this case.

But there is a form of SRS for which leftmost-outermost reduction will discover normal forms if they exist. Say that a rewriting rule, $lhs \Rightarrow rhs$, is *left normal* if no variable in *lhs* precedes a constant or operator in *lhs*. The first rule in the above example violates this condition, because both *X* and *Y* precede the constant *true*. An orthogonal SRS is left normal if all its rules are. The example SRSs in the previous section are left normal. (The β-rule is discussed below.)

6.23 Theorem

If an SRS is left normal, then leftmost-outermost reduction generates eventually outermost reduction sequences.

Thus, leftmost-outermost reduction for a left normal SRS will discover a normal form for any expression that has one.

The intuition in the preceding development can be generalized to handle the β-rule, since the rule is left normal in the sense that, when we write its left-hand side as $apply\ (\lambda \mathrm{I}.\,{}_{\mathrm{I}}\mathbf{X})\ Y$ (*apply* is an explicit operator that stands for application), we see that no variable precedes the operators *apply* and λI. (The identifiers, I, in ${}_{\mathrm{I}}\mathbf{X}$ are ‘‘hole markers’’ and not constants in the usual sense.) The β-*val* rule is problematic, however, because the goal of a reduction sequence that uses β-*val* is to reduce a phrase to a Value, which is not necessarily the same as a normal form. Also, when the β-*val* rule is written as a family of orthogonal rules, it is clear that β-*val* is not left normal.

Given a left normal SRS, where Values are also normal forms, say that we add lambda abstractions and the β-*val* rule. Then, the strategy of reducing the leftmost-outermost redex not contained within the body of a lambda abstraction suffices for computing a Value. The proof is nontrivial, but some intuition can be gained in terms of the development in this section. The body of a lambda abstraction should be invisible to the rewriting rules, so imagine that the bodies of lambda abstractions are covered by boxes, for example, $\lambda X.\,(\lambda Y.Y)A$ looks like $\lambda X.\square$ to the rewriting rules. The box is a kind of constant, and this makes the β-*val* rule left normal. Further, the lambda abstraction

$\lambda X.\square$ is a kind of normal form. The leftmost-outermost reduction strategy on phrases with boxed lambda abstractions is exactly the strategy of reducing leftmost redexes not contained within lambda abstractions. Such a strategy suffices for finding normal forms and therefore Values.

Suggested Reading

Church 1941 remains a readable presentation of the lambda calculus, although a beginner might prefer Hindley and Seldin 1986, and the advanced reader might consult Barendregt 1984. Curry and Feys 1958 is a standard and important reference. The simply typed lambda calculus is described in Hindley and Seldin 1986, and extensions are covered in Barendregt and Hemerik 1990 and Barendregt 1992. Morris 1968 shows how to use δ-rules to model a core programming language. The approach led to denotational semantics; see Strachey 1966, 1968, and 1973 for evidence. The standard reference for the β-*val* rule is Plotkin 1975; see also Felleisen and Hieb 1992.

Fundamental concepts of subtree replacement systems can be found in Klop 1992 and Dershowitz and Jouannaud 1990; background can be found in Huet 1980, Huet and Oppen 1980, and Klop 1980. The standardization results in this chapter are based on O'Donnell 1977; another reference is Dershowitz 1987.

Exercises

1.1. Abbreviate these lambda expressions by removing all superfluous parentheses:
   a. $((\lambda Y.\ (\lambda X.\ ((Y\ Z)X)))(\lambda X.\ X))$
   b. $(\lambda X.\ ((\lambda Y.\ ((\lambda X.\ Y)Y))X))$
   c. $((\lambda Y.\ ((\lambda Z.\ A)(Y\ Y)))(\lambda X.\ (X\ X)))$

1.2. Reinsert all parentheses in these lambda expressions:
   a. $\lambda X.\ (\lambda Y.\ Y\ Y)Z\ X$
   b. $(\lambda Y.\ Y\ Y\ Y)(\lambda X.\ X\ X)$

1.3. a. Using just the β-rule, reduce the expressions in Exercises 1.1 and 1.2 to their β-normal forms. If an expression does not appear to have a normal form, explain why.
   b. Using the β- and η-rules, reduce the expressions in Exercises 1.1 and 1.2 to their βη-normal forms, if they exist.

1.4. Demonstrate that the α-rule is crucial to Theorem 6.1, that is, confluence cannot be strengthened to ''for any lambda expression E, if $E \Rightarrow^* E_1$ and $E \Rightarrow^* E_2$, then there exists a lambda expression E_3 such that $E_1 \Rightarrow^* E_3$ and $E_2 \Rightarrow^* E_3$.''

1.5. An important consequence of the standardization is that it can be used to prove that a lambda expression does *not* have a normal form. Use induction on the length of a leftmost-outermost reduction sequence to prove that these expressions do not have normal forms:

a. $(\lambda X.\ X\ X)(\lambda X.\ X\ X)$

b. $(\lambda X.\ F(X\ X))(\lambda X.\ F(X\ X))$

1.6. Using Church numerals and the encoding for the addition operation

a. verify that $add\ \overline{2}\ \overline{3} \Rightarrow^* \overline{5}$.

b. prove, by mathematical induction, for all m and n, that $add\ \overline{m}\ \overline{n} \Rightarrow^* \overline{m{+}n}$.

c. let $\overline{true}$ be encoded by $(\lambda X.\lambda Y.\ X)$ and $\overline{false}$ be encoded by $(\lambda X.\lambda Y.\ Y)$. Code an operation, *not*, such that $not\ \overline{false} \Rightarrow^* \overline{true}$ and $not\ \overline{true} \Rightarrow^* \overline{false}$. Similarly, encode *if* such that $if\ \overline{true}\ E_1\ E_2 \Rightarrow^* E_1$ and $if\ \overline{false}\ E_1\ E_2 \Rightarrow^* E_2$. Finally, encode logical conjunction and disjunction.

d. let $eq0$ be $\lambda N.\ N\ (\lambda X.\ \overline{false})\ \overline{true}$. Verify that $eq0\ \overline{0} \Rightarrow^* \overline{true}$ and $eq0\ \overline{n{+}1} \Rightarrow^* \overline{false}$.

e. code the multiplication and exponentiation operations on Church numerals.

f. The Church numerals simulate *simple recursion* on the natural numbers, namely, $\overline{n}\ k\ g$ implements $f(n)$, where: $f(0) = k$, and $f(n{+}1) = g(f(n))$. Now, consider primitive recursion, which takes the form: $f(0) = k$, and $f(n{+}1) = g(n, f(n))$. Show how to use Church numerals to encode primitive recursion. (Hint: Consider this form of simple recursive function

$$f(0) = (0, k)$$
$$f(n{+}1) = (n{+}1, g(f(n)))$$

where f returns a pair, consisting of the argument value and its answer. Simulate pairs, and use them to simulate primitive recursion.) Use your construction to encode the predecessor (‘‘minus one’’) function; the factorial function.

1.7. When β-reducibility is extended to a congruence relation, the result is β-*convertibility*. Let *cnv* be the reflexive, symmetric, transitive, substitutive closure of the (α- and) β-rule(s). That is, $E_1\ cnv\ E_n$ iff there exist $E_2, \ldots, E_{n-1}$ such that, for all $0 \le i \le n$, $E_i \Rightarrow E_{i+1}$ or $E_{i+1} \Rightarrow E_i$. The *Church-Rosser property* is: If $E_1\ cnv\ E_2$, then there is some E_3 such that $E_1 \Rightarrow^* E_3$ and $E_2 \Rightarrow^* E_3$ (modulo application of the α-rule). Prove that the Church-Rosser property is equivalent to the confluence property.

1.8. Read Exercise 1.6. Now, let $\mathbf{Y} = \lambda F.\ (\lambda X.\ F(X\ X))(\lambda X.\ F(X\ X))$.

a. Show that $\mathbf{Y}\ F\ cnv\ F(\mathbf{Y}\ F)$.

b. If you worked Exercise 1.6, proceed. Let *pred* be some lambda expression such that $pred\ \overline{0} \Rightarrow^* \overline{0}$ and $pred\ \overline{n{+}1} \Rightarrow^* \overline{n}$. Use **Y**, *pred*, and the expressions from Exercise 1.6 to encode the factorial function on Church numerals; to encode Ackermann's function on Church numerals. Note: Ackermann's function is defined as follows:

$ack\ 0\ n = n{+}1$
$ack\ (m{+}1)\ 0 = ack\ m\ 1$
$ack\ (m{+}1)\ (n{+}1) = ack\ m\ (ack\ (m{+}1)\ n)$

1.9. Here is a structural operational semantics for the lambda calculus and its β-rule. A computation step is a proof of $E_1 \rhd E_2$, and a computation is a sequence of steps $E_1 \rhd E_2, E_2 \rhd E_3, \cdots, E_{n-1} \rhd E_n$. Here are the rules:

$$(\lambda I.\ E_1)E_2 \rhd [E_2/I]E_1 \qquad \frac{E_1 \rhd E_1'}{E_1\ E_2 \rhd E_1'\ E_2}$$

a. Calculate the semantics of the expressions in Exercises 1.1 and 1.2. What reduction strategy is encoded by the semantics?

b. Alter the semantics so that any redex whatsoever can be selected for a computation step. Prove that $E_1 \Rightarrow^* E_n$ iff $E_1 \rhd E_2 \rhd \cdots \rhd E_n$.

1.10 Here is a natural semantics for the lambda calculus and its β-rule. A computation is a proof of $E_1 \rhd E_2$. These are the rules:

$$\lambda I.\ E \rhd \lambda I.\ E \qquad I \rhd I \qquad \frac{E_1 \rhd \lambda I.\ E_1' \quad [E_2/I]E_1' \rhd E}{E_1\ E_2 \rhd E}$$

a. Calculate the semantics of the expressions in Exercises 1.1 and 1.2. What reduction strategy is encoded by the semantics?

b.* Alter the semantics so that it encodes a leftmost-outermost reduction strategy; prove this.

1.11. Here is yet another logic for the lambda calculus, which we call the *theory of β-reducibility* (cf. Hindley and Seldin 1986). Its formulas are phrases of the form $E_1 \rhd E_2$, and its axioms and rules follow:

$$\lambda I.E \rhd \lambda I'.[I'/I]E, \text{ if } I' \notin FV(E) \qquad (\lambda I.\ E_1)E_2 \rhd [E_2/I]E_1 \qquad E \rhd E$$

$$\frac{E_1 \rhd E_1' \quad E_2 \rhd E_2'}{E_1\ E_2 \rhd E_1'\ E_2'} \qquad \frac{E \rhd E'}{\lambda I.E \rhd \lambda I.E'} \qquad \frac{E_1 \rhd E_2 \quad E_2 \rhd E_3}{E_1 \rhd E_3}$$

a. Prove that $E_1 \Rightarrow^* E_2$ (with the α- and β-rules) iff $E_1 \rhd E_2$ holds.

b. The *theory of β-convertibility* is the theory of β-reducibility, where the ‘‘$\rhd$’’ symbol is changed to the ‘‘≡’’ symbol and this rule is added:

$$\frac{E_1 \equiv E_2}{E_2 \equiv E_1}$$

Prove that E_1 *cnv* E_2 iff $E_1 \equiv E_2$ holds. (See Exercise 1.7 for the definition of *cnv*.)

1.12. An expression, E, is in *head normal form* if it has the form $(\lambda I_1.\lambda I_2. \cdots \lambda I_m.\ I\ E_1\ E_2 \cdots E_n)$, $m, n \geq 0$. I may be an I_j, $1 \leq j \leq m$, but it is not necessary; for example, $\lambda X.\ Y\ X$ is in head normal form. (See Barendregt 1984.)

a. Prove that $\Omega = (\lambda X.\ X\ X)(\lambda X.\ X\ X)$ has no head normal form but **Y** (see Exercise 1.8) does. Hence, an expression that has no normal form can have a head normal form.

The intuition is that expressions that have head normal forms are computationally useful. This can be formalized. A closed expression, E, is *solvable* iff there exist expressions, $E_1, E_2, \ldots, E_n$, $n \geq 0$, such that $(E\ E_1\ E_2 \cdots E_n)$ *cnv* $(\lambda X.\ X)$.

b. Prove that Ω is not solvable (hint: use Theorem 6.3) but that **Y** is solvable.

c.* Prove, for every closed expression, E, that E is solvable iff E has a head normal form.

For an expression, $(\lambda I_1.\lambda I_2. \cdots \lambda I_m.\ (\lambda I.E_0)\ E_1\ E_2 \cdots E_n)$, $m \geq 0$, $n > 0$, say that $(\lambda I.E_0)E_1$ is the *head redex* of the expression. The *head redex reduction strategy* reduces the head redex at each stage of a reduction sequence.

d. Use Theorem 6.3 to prove that E has a head normal form iff the head redex reduction strategy applied to E terminates.

An expression, E, is in *weak head normal form* if it is in head normal form or it is a lambda abstraction. (See Abramsky 1990 or Peyton Jones 1987.)

e. Give an example of an expression that has weak head normal form but not head normal form.

f. Define a reduction strategy such that an expression, E, has weak head normal form iff the reduction strategy applied to E terminates. Prove this.

g. If you understand the implementation of a functional language like Lisp, Scheme, or ML, comment as to which of the notions of normal form, head normal form, or weak head normal form is most suited to describing the implementation.

2.1. Apply the β-*val* rule to these examples:

a. $(\lambda X.\ (\lambda Y.\ Y\ Y)(\lambda Y.\ X\ X))((\lambda Y.\ Y)(\lambda X.\ X\ X))$

b. $(\lambda X.\ (\lambda Z.\ (\lambda Y.\ Y)\ X)(X\ X))(\lambda X.\ X\ X)$

2.2. Say that the β-*val* rule is altered to read as follows:

$(\lambda I.\ E_1)E_2 \ \Rightarrow\ [E_2/I]E_1$, if E_2 is in normal form

Show that the confluence property fails. (Hint: Consider when E_2 is $(X\ X)$.)

2.3. Define a reduction strategy for the β-*val* rule and prove that it discovers a Value for an expression, if one exists.

2.4. If you worked Exercises 1.9, 1.10, or 1.11, proceed. Alter those semantics definitions so that the β-*val* rule is used in place of the β-rule.

2.5. Here is another logistic system for the lambda calculus. Let e stand for an *environment* of the usual form $\{I_1=v_1, \ldots, I_n=v_n\}$. Configurations have the form $e \vdash E \Rightarrow v$. Here are the axioms and inference rules:

$$e \vdash (\lambda I.\, E) \Rightarrow \langle e, I, E\rangle \qquad e \vdash I \Rightarrow v, \text{ if } (I=v) \in e$$

$$\frac{e \vdash E_1 \Rightarrow \langle e', I, E'\rangle \qquad e \vdash E_2 \Rightarrow v \qquad e' \uplus \{I=v\} \vdash E' \Rightarrow v'}{e \vdash (E_1\ E_2) \Rightarrow v'}$$

The system is another natural semantics of the lambda calculus.

a. Prove: If $\{I_1=v_1, \ldots, I_n=v_n\} \vdash E \Rightarrow v$ can be proved, then $[v_1/I_1, \ldots, v_n/I_n]E \Rightarrow^* v$.

b. Show that the converse to Part a does not hold.

c. Describe the form of reduction strategy defined by the logistic system.

3.1. Let $E_1 \equiv E_2$ mean that E_1 and E_2 are the same expression, modulo application of the α-rule, and let $E_1 \Rightarrow^* E_2$ mean that E_1 reduces, by means of the β-rule, to E_2, modulo applications of the α-rule. Use induction on rank to prove the following:

a. If $J \notin FV(E_1)$, then $[E_2/J][J/I]E_1 \equiv [E_2/I]E_1$.

b. If $J \notin FV(E_2)$, then $[E_2/I][E_3/J]E_1 \equiv [([E_2/I]E_3)/J][E_2/I]E_1$.

c. $[E_2/I][E_3/I]E_1 \equiv [([E_2/I]E_3)/I]E_1$.

d. If $E_1 \Rightarrow^* E_2$, then $[E_1/I]E_3 \Rightarrow^* [E_2/I]E_3$.

e. If $E_1 \Rightarrow^* E_2$, then $[E_3/I]E_1 \Rightarrow^* [E_3/I]E_2$.

3.2. Following are three alternatives to the classic definition of substitution. For each, (i) reformulate the β-rule; (ii) re-prove Exercise 3.1.

a. From Barendregt 1984, the ‘‘expressions’’ of the lambda calculus are equivalence classes with respect to the α-rule. Let $[E]_\alpha$ represent an equivalence class. For example, the equivalence classes $[\lambda X.X]_\alpha$ and $[\lambda Y.Y]_\alpha$ are the same ‘‘expression.’’ Substitution is defined as follows:

$[[E]_\alpha/I]\,[I]_\alpha = [E]_\alpha$
$[[E]_\alpha/I]\,[J]_\alpha = [J]_\alpha$, if $J \neq I$
$[[E]_\alpha/I]\,[E_1\ E_2]_\alpha = [E_1{}'\ E_2{}']_\alpha$, where $[E_1{}']_\alpha = [[E]_\alpha/I]\,[E_1]_\alpha$,
and $[E_2{}']_\alpha = [[E]_\alpha/I]\,[E_2]_\alpha$
$[[E]_\alpha/I]\,[\lambda J.\, E_1]_\alpha = [\lambda J.\, E_1{}']_\alpha$
where $J \neq I$, J is not free in E, and $[E_1{}']_\alpha = [[E]_\alpha/I][E_1]_\alpha$

Prove that the definition of substitution is well defined; that is: (i) for all expressions, E_1, $[[E]_\alpha/I][E_1]_\alpha$ is defined; (ii) if expressions E_1 and E_2 are α-convertible, then $[[E]_\alpha/I][E_1]_\alpha = [[E]_\alpha/I][E_2]_\alpha$. The proof method for the calculus is structural induction. Explain why structural induction is sound for expressions-as-equivalence-classes.

b. From Stoughton 1984b, the expressions for the lambda calculus are the traditional ones. A *substitution*, σ, is a mapping from identifiers to expressions. For convenience, we represent a substitution as a (finite) set of the form $\{I_1{=}E_1, \ldots, I_n{=}E_n\}$. We write $\sigma.I = E$, when $(I{=}E) \in \sigma$, and write $\sigma.I = I$ otherwise. Application of a substitution, σ, to an expression, E, is written σE. It is defined as

$$\sigma I = \sigma.I$$
$$\sigma(E_1\ E_2) = (\sigma E_1\ \ \sigma E_2)$$
$$\sigma(\lambda I.E) = (\lambda J.\,(\sigma \uplus \{I{=}J\})E)$$

where J is the ‘‘first’’ identifier in an enumeration of all identifiers that do not appear free in any $\sigma.I'$, for all $I' \in FV(\lambda I.E)$. The proof method for the calculus is structural induction. Define $\sigma_2 \circ \sigma_1$ to be the mapping: $(\sigma_2 \circ \sigma_1).I = \sigma_2(\sigma_1.I)$. Prove that for all σ_1, σ_2, and E, $(\sigma_2 \circ \sigma_1)E = \sigma_2(\sigma_1 E)$.

c. From de Bruijn 1972 and Barendregt 1984, the expressions for the lambda calculus have the following syntax:

$$E ::= E_1\ E_2 \mid \lambda E \mid n, \text{ for } n > 0$$

The intuition is that a numeral stands for an identifier that binds to the *n*th enclosing lambda. For example, $(\lambda\ 1\ (\lambda\ 2\ 1\ 4))\ 1$ represents $(\lambda X.\ X\ (\lambda Y.\ X\ Y\ F_2))\ F_1$, where the free identifiers are enumerated as F_1, F_2, ... Define $[E_1/n]E_2$. (Be careful about the ‘‘free identifiers’’ in E_1.) The proof method for the calculus is structural induction.

4.1. a. Prove the unicity of typing property for the simply typed lambda calculus: For all π, E, and τ, if $\pi \vdash E{:}\,\tau$ holds, then τ is unique.

b. Prove that if $\pi \uplus \{I{:}\tau\} \vdash E{:}\,\tau'$ holds, then $\pi \uplus \{J{:}\tau\} \vdash [J/I]E{:}\tau'$ holds.

c. Prove that if $\pi \vdash E{:}\,\tau$ holds and K is a fresh identifier, then $\pi \uplus \{K{:}\tau'\} \vdash E{:}\,\tau$ holds.

d. Prove the converse to Theorem 6.7: if $\pi \vdash E_2{:}\,\tau_2$ and $\pi \vdash [E_2/I]E_1{:}\,\tau_1$ hold, then so does $\pi \uplus \{I{:}\tau_2\} \vdash E_1{:}\,\tau_1$.

e. Prove that for all π, I, and τ, that $\pi \vdash (I\ I){:}\,\tau$ can*not* hold. Conclude from Part d that if $\pi \vdash E{:}\,\tau$ holds, then $\pi \vdash E\ E{:}\,\tau'$ cannot.

4.2. A useful tool for studying reduction strategies is the *evaluation context* (cf. Felleisen and Hieb 1992). An evaluation context is a phrase with a single ‘‘hole,’’ into which a redex can fit. Here is a syntax rule for evaluation contexts:

$$X ::= [\,] \mid (X\,E)$$

E is a lambda calculus expression, as usual, and [ ] is a hole; we write $C[\,]$ to denote an evaluation context.

a. Prove that if E_0 contains a β-redex, R, then E_0 can be derived in the form $C[R]$ iff R is the leftmost redex in E_0 not contained within a lambda abstraction.

b. Say that a reduction step, $E \Rightarrow E'$, must match exactly this format:

$$C[(\lambda I.\, E_1)E_2] \;\Rightarrow\; C[[E_2/I]E_1]$$

What reduction strategy is encoded by this rewriting rule?

c. Consider this syntax for evaluation contexts:

$$\begin{aligned} X &::= [\,] \mid (X\,E) \mid (V\,X) \\ V &::= \lambda I.\, E \mid I \end{aligned}$$

If the β-*val* rule is used in place of the β-rule (cf. Part b), what reduction strategy is defined?

d. We can perform simple proofs of reduction properties by induction on the structure of evaluation contexts. Re-prove the subject-reduction property for the reduction strategies in Parts b and c.

4.3. Add records to the simply typed lambda calculus, with the syntax and typing rules in Figure 5.2. The rewriting rule is

$$\textit{with}\ \{I_1{=}E_1, I_2{=}E_2, \ldots, I_n{=}E_n\}\ \textit{do}\ E \;\Rightarrow\; [E_1/I_1, E_2/I_2, \ldots, E_n/I_n]E$$

So, parallel substitution is required.

a. Give a complete, formal definition of parallel substitution for the simply typed lambda calculus with records. (Take care with the definition of substitution into a *with* expression!)

b. Define the rank of a record expression and a *with* expression.

c. Reprove Theorem 6.4.

Confluence and strong normalization hold for this calculus.

5.1. Alter the lazy evaluation semantics of the lambda calculus in the following ways. First, for ground type, ι, add the constant, *loop*, such that $\pi \vdash \textit{loop}: \iota$ holds. Say that $[\![\pi \vdash \textit{loop}: \iota]\!]e = \bot$. Next, replace the semantics equations for lambda abstraction and application by these:

$$\begin{aligned}
&[\![\pi \vdash \lambda I{:}\tau_1.\, E: \tau_1 \to \tau_2]\!]e = f, \\
&\qquad \text{where } f\ \bot = \bot \\
&\qquad \text{and } f\ u = [\![\pi \uplus \{I{:}\tau_1\} \vdash E: \tau_2]\!](e \uplus \{I{=}u\}),\ \text{if } u \neq \bot \\
&[\![\pi \vdash E_1\, E_2: \theta_2]\!]e = \bot,\ \text{if } [\![\pi \vdash E_1: \tau_1 \to \tau_2]\!]e = \bot \\
&[\![\pi \vdash E_1\, E_2: \theta_2]\!]e = ([\![\pi \vdash E_1: \theta_1 \to \theta_2]\!]e)\ ([\![\pi \vdash E_2: \theta_1]\!]e),\ \text{otherwise}
\end{aligned}$$

Prove that the β-*val* rule is sound for this semantics but the β-rule is not.

6.1. Verify that the following expressions are well typed and reduce them to normal forms, if normal forms exist.

a. $(\mathit{equals}\ (\mathit{if}\ (\mathit{not}(\mathit{equals}\ 3\ 2))\ 4\ (\mathit{plus}\ 5\ (\mathit{if}\ \mathit{true}\ 6\ 7)))\ 8)$

b. $(\lambda X{:}\mathit{int}.\lambda Y{:}\mathit{int}.\ \mathit{plus}\ X\ Y))((\mathit{if}\ \mathit{true}\ (\lambda X{:}\mathit{int}.\ \mathit{plus}\ X\ 1)\ (\lambda Y{:}\mathit{int}.\ Y))\ 3)$

c. $(\mathit{fix}\ (\lambda F{:}\mathit{int}{\rightarrow}\mathit{int}.\lambda X{:}\mathit{int}.\ \mathit{plus}\ X\ (F\ (\mathit{plus}\ X\ 1))))\ 0$

6.2. Use the *fix* operator to encode the multiplication function in terms of the addition operator; use *fix* to encode exponentiation in terms of multiplication.

7.1. Calculate the substitution semantics and the environment semantics of these programs with the input store $\langle 0,1\rangle$:

a. $(\lambda X{:}\mathit{intloc}.\ (\lambda P{:}\mathit{comm}{\rightarrow}\mathit{comm}.\ X{:=}0;\ (P\ (X{:=}@X{+}1)))(\lambda Q{:}\mathit{comm}.\ Q))\mathit{loc}_1$

b. **while** $\mathit{loc}_1{=}0$ **do** $(\lambda A{:}\mathit{intexp}.\ \mathit{loc}_2{:=}A)@\mathit{loc}_1$ **od**

7.2. Say that the β-rule is used in place of the β-*val* rule to reduce lambda abstractions in the metalanguage. Does this make any difference to the semantics of the source language?

8.1. Consider this SRS:

$$(\ \mathrm{E}\ ::=\ a \mid b \mid f\,\mathrm{E} \mid g\,\mathrm{E}_1\,\mathrm{E}_2,$$
$$\{f\,a \Rightarrow b,\quad f\,b \Rightarrow a,\quad g\,X\,(f\,Y) \Rightarrow g\,(f\,Y)\,(g\,X\,X)\}\ )$$

a. Is the SRS orthogonal?

b. Calculate the residuals of $(f\,a)$ at each stage in the leftmost outermost reduction sequence of $g\,(f\,a)\,(f\,b)$.

c. Do a complete development of the redex set $\{(f\,a), (g\,(f\,a)\,(f\,b))\}$ for $g\,(f\,a)\,(f\,b)$. Is there more than one possible complete development?

d. Does the SRS have the closure property? Confluence?

8.2. a. Give an example of an SRS that is not orthogonal but has the closure property.

b. Give an example of an SRS that does not have closure but has confluence.

8.3. Verify that the rewriting rules in Section 6.6 form an orthogonal SRS.

8.4.* Prove Proposition 6.17.

8.5.* Without appealing to orthogonality, prove that the lambda calculus and its β-rule have the closure property.

8.6. There is a close relationship between certain orthogonal SRSs and natural semantics definitions. The intuition is that a natural semantics rule

$$\frac{\mathrm{E}_1 \triangleright v_1 \quad \cdots \quad \mathrm{E}_n \triangleright v_n}{\mathit{op}\,\mathrm{E}_1 \cdots \mathrm{E}_n \triangleright f_{v_1 \cdots v_n}}$$

corresponds to the rewriting rule $op\, \mathrm{E}_1 \cdots \mathrm{E}_n \Rightarrow f_{v_1 \cdots v_n}$, where $v_1, \ldots, v_n$ are Values.

a. Define the class of orthogonal SRSs and natural semantics definitions that ‘‘correspond’’ and give translations between the two.

b. An advantage of natural semantics definitions is that proofs about computations can be performed by induction on the structure of the proof trees. Based on your answer to Part a, state an induction principle for proofs of computations in orthogonal SRSs.

9.1. Which of these reduction strategies for orthogonal SRSs are eventually outermost? Justify your answer.

a. Full reduction: At each stage, all redexes in an expression are reduced in parallel.

b. Parallel innermost: At each stage, all redexes not containing other redexes are reduced in parallel.

c. Rightmost outermost: At each stage, the rightmost redex is reduced.

d. Round robin: A queue is kept of the redexes that were outermost at some stage. At each stage, the front element of the queue is selected, and all residuals of the redex are reduced. Any new outermost redexes are added to the queue. (The queue is initialized with all the outermost redexes in the initial expression.)

9.2. Say that an SRS is not orthogonal. Will an eventually outermost reduction strategy always discover a normal form?