5 The Untyped Lambda-Calculus

\[
\begin{align*}
\text{pair} \ & \text{C}_0 \text{ C}_0 \\
\text{ss} & \quad \text{add} \quad +1 \\
\text{pair} \ & \text{C}_0 \text{ C}_1 \\
\text{ss} & \quad \text{add} \quad +1 \\
\text{pair} \ & \text{C}_1 \text{ C}_2 \\
\text{ss} & \quad \text{add} \quad +1 \\
\text{pair} \ & \text{C}_2 \text{ C}_3 \\
\text{ss} & \quad \text{add} \quad +1 \\
\text{pair} \ & \text{C}_3 \text{ C}_4 \\
\vdots
\end{align*}
\]

Figure 5.1: The predecessor function's "inner loop"

\[
\text{iszero} \ = \ \lambda m \cdot \ m \ (\lambda x. \ \text{fals} \ \text{tru})
\]

\[
iszero \ \text{C}_0;
\]

\[
\ast \ (\lambda t. \ \text{fals} \ \text{fals})
\]

\[
iszero \ \text{times} \ \text{C}_0 \ \text{C}_2;
\]

\[
\ast \ (\lambda t. \ \text{fals} \ \text{fals})
\]

Surprisingly, subtraction using Church numerals is quite a bit more difficult than addition. It can be done using the following rather tricky "predecessor function," which, given \( \text{C}_0 \) as argument, returns \( \text{C}_0 \) and, given \( \text{C}_{n+1} \), returns \( \text{C}_n \):

\[
\text{zz} = \text{pair} \ \text{C}_0 \ \text{C}_0;
\]

\[
\text{ss} = \lambda p: \text{pair} \ \text{snd \ p} \ (\text{plus} \ \text{C}_1 \ (\text{snd \ p}));
\]

\[
\text{prd} = \lambda m. \ \text{fst} \ (m \ \text{ss} \ \text{zz});
\]

This definition works by using \( m \) as a function to apply \( m \) copies of the function \( ss \) to the starting value \( zz \). Each copy of \( ss \) takes a pair of numerals \( \text{pair} \ \text{C}_1 \ \text{C}_2 \) as its argument and yields \( \text{pair} \ \text{C}_1 \ \text{C}_{n+1} \) as its result (see Figure 5.1). So applying \( ss \) \( m \) times, to \( \text{pair} \ \text{C}_0 \ \text{C}_0 \) yields \( \text{pair} \ \text{C}_0 \ \text{C}_0 \) when \( m = 0 \) and \( \text{pair} \ \text{C}_{m-1} \ \text{C}_m \) when \( m \) is positive. In both cases, the predecessor of \( m \) is found in the first component.

5.2.5 Exercise [**]: Use \( \text{prd} \) to define a subtraction function.

5.2.6 Exercise [***]: Approximately how many steps of evaluation (as a function of \( n \)) are required to calculate \( \text{prd} \ \text{C}_n \)?

5.2.7 Exercise [***]: Write a function \( \text{equal} \) that tests two numbers for equality and returns a Church boolean. For example,

\[
\ast \ (\lambda t. \ \text{fals} \ \text{fals})
\]

\[
\ast \ (\lambda t. \ \text{fals} \ \text{fals})
\]

Other common datatypes like lists, trees, arrays, and variant records can be encoded using similar techniques.

5.2.8 Exercise [Recommended, ***]: A list can be represented in the lambda-calculus by its \( \text{fold} \) function. (OCaml's name for this function is \( \text{fold\_left} \); it is also sometimes called \( \text{reduce} \).) For example, the list \( [x, y, z] \) becomes a function that takes two arguments \( c \) and \( n \) and returns \( c \times (c \times (c \times n)) \). What would the representation of \( \text{nil} \) be? Write a function \( \text{cons} \) that takes an element \( h \) and a list \( (\text{that is, a fold function}) t \) and returns a similar representation of the list formed by prepending \( h \) to \( t \). Write \( \text{ins1} \) and \( \text{head} \), each taking a list parameter. Finally, write a \( \text{tail} \) function for this representation of lists (this is quite a bit harder and requires a trick analogous to the one used to define \( \text{prd} \) for numbers).

Enriching the Calculus

We have seen that booleans, numbers, and the operations on them can be encoded in the pure lambda-calculus. Indeed, strictly speaking, we can do all the programming we ever need to without going outside of the pure system. However, when working with examples it is often convenient to include the primitive booleans and numbers (and possibly other data types) as well. When we need to be clear about precisely which system we are working in, we will use the symbol \( \lambda \) for the pure lambda-calculus as defined in Figure 5.3 and ANB for the enriched system with booleans and arithmetic expressions from Figures 3.1 and 3.2.

In \( \lambda \text{NB} \), we actually have two different implementations of booleans and two of numbers to choose from when writing programs: the real ones and the encodings we've developed in this chapter. Of course, it is easy to convert back and forth between the two. To turn a Church boolean into a primitive boolean, we apply it to true and false:
realbool = \ab. \b \true \false

To go the other direction, we use an if expression:

churchbool = \ab. \if \b \then \true \else \false \fi

We can build these conversions into higher-level operations. Here is an equality function on Church numerals that returns a real boolean:

realeq = \mn. (equal \m \n) \true \false

In the same way, we can convert a Church numeral into the corresponding primitive number by applying it to succ and 0:

realnat = \mn. m (\x. succ x) 0;

We cannot apply \n to succ directly, because succ by itself does not make syntactic sense: the way we defined the syntax of arithmetic expressions, succ must always be applied to something. We work around this by packaging succ inside a little function that does nothing but return the succ of its argument.

The reasons that primitive booleans and numbers come in handy for examples have to do primarily with evaluation order. For instance, consider the term \scc \c_1. From the discussion above, we might expect that this term should evaluate to the Church numeral \c_2. In fact, it does not:

scc \c_1;
• (\s. \lambda z. s ((\s'. \lambda z'. s' z') s) z)

This term contains a redex that, if we were to reduce it, would bring us (in two steps) to \c_2, but the rules of call-by-value evaluation do not allow us to reduce it yet, since it is under a lambda-abstraction.

There is no fundamental problem here: the term that results from evaluation of \scc \c_1 is obviously behaviorally equivalent to \c_2, in the sense that applying it to any pair of arguments \v and \w will yield the same result as applying \c_2 to \v and \w. Still, the leftover computation makes it a bit difficult to check that our \scc function is behaving the way we expect it to. For more complicated arithmetic calculations, the difficulty is even worse. For example, \times \c_2 \c_2 evaluates not to \c_4 but to the following monstrosity:

\times \c_2 \c_2;
• (\s. \lambda z. (\s'. \lambda z'. s' (s' z')) s)

6. It is often called the call-by-value \Y combinator. Plotkin (1975) called it \texttt{Z}.

7. Note that the simpler call-by-name fixed point combinator

\Y = \texttt{fix} \equiv \lambda f. (\texttt{fix} \; \lambda \f. (\f \; (\f \; \texttt{x} \; \texttt{x})))

is useless in a call-by-value setting, since the expression \Y \; g diverges, for any \g.

8. It is also possible to derive the definition of \texttt{Fix} from first principles (e.g., Friedman and Felleisen, 1996, Chapter 9), but such derivations are also rather intricate.
of the form \( h = (\text{body containing } h) \) — i.e., we want to write a definition where the term on the right-hand side of the = uses the very function that we are defining, as in the definition of factorial on page 52. The intention is that the recursive definition should be "unrolled" at the point where it occurs; for example, the definition of factorial would intuitively be

\[
\begin{align*}
\text{if } n = 0 \text{ then } 1 \\
\text{else } n \times (\text{if } n-1 = 0 \text{ then } 1 \\
\text{else } (n-1) \times (\text{if } n-2 = 0 \text{ then } 1 \\
\text{else } (n-2) \times \ldots))
\end{align*}
\]

or, in terms of Church numerals:

\[
\begin{align*}
\text{if realeq } n \ c_0 \ \text{then } c_1 \\
\text{else times } n \ (\text{if realeq } (\text{prd } n) \ c_0 \ \text{then } c_1 \\
\text{else times } (\text{prd } n) \\
(\text{if realeq } (\text{prd } (\text{prd } n)) \ c_0 \ \text{then } c_1 \\
\text{else times } (\text{prd } (\text{prd } n)) \ldots))
\end{align*}
\]

This effect can be achieved using the fix combinator by first defining \( g = \text{M} \cdot (\text{body containing f}) \) and then \( h = \text{fix } g \). For example, we can define the factorial function by

\[
g = \lambda \text{fct. } \lambda n. \text{if realeq } n \ c_0 \ \text{then } c_1 \ \text{else } \text{times } n \ (\text{fct } (\text{prd } n)); \\
\text{factorial } = \text{fix } g;
\]

Figure 5.2 shows what happens to the term \text{factorial } c_3 during evaluation. The key fact that makes this calculation work is that \( \text{fct } n \rightarrow ^* \ g \ \text{fct } n \). That is, \text{fct} is a kind of "self-replicator" that, when applied to an argument, supplies itself and \( n \) as arguments to \( g \). Wherever the first argument to \( g \) appears in the body of \( g \), we will get another copy of \( \text{fct} \), which, when applied to an argument, will again pass itself and that argument to \( g \), etc. Each time we make a recursive call using \( \text{fct} \), we unroll one more copy of the body of \( g \) and equip it with new copies of \( \text{fct} \) that are ready to do the unrolling again.

5.2.9 Exercise [+] Why did we use a primitive if in the definition of \( g \), instead of the Church-boolean test function on Church booleans? Show how to define the \text{factorial} function in terms of \text{test} rather than if.

5.2.10 Exercise [**]: Define a function \text{churnout} that converts a primitive natural number into the corresponding Church numeral.

5.2.11 Exercise [Recommended, **]: Use \text{fix} and the encoding of lists from Exercise 5.2.8 to write a function that sums lists of Church numerals.
5 The Untyped Lambda-Calculus

- an operation \texttt{iszero} mapping numbers to booleans, and
- two operations, \texttt{succ} and \texttt{pred}, mapping numbers to numbers.

The behavior of the arithmetic operations is defined by the evaluation rules in Figure 3-2. These rules tell us, for example, that \( 3 \) is the successor of \( 2 \), and that \texttt{iszero} \( 0 \) is true.

The Church encoding of numbers represents each of these elements as a lambda-term (i.e., a function):

- The term \( \texttt{c0} \) represents the number 0.
  
  As we saw on page 64, there are also "non-canonical representations" of numbers as terms. For example, \( \lambda x . \lambda z . (\lambda x . x) z \), which is behaviorally equivalent to \( \texttt{c0} \), also represents 0.

- The terms \( \texttt{scc} \) and \( \texttt{prd} \) represent the arithmetic operations \( \text{succ} \) and \( \text{pred} \), in the sense that, if \( \tau \) is a representation of the number \( n \), then \( \texttt{scc} \tau \) evaluates to a representation of \( n + 1 \) and \( \texttt{prd} \tau \) evaluates to a representation of \( n - 1 \) (or of 0, if \( n \) is 0).

- The term \( \texttt{iszero} \) represents the operation \texttt{iszero}, in the sense that, if \( \tau \) is a representation of 0, then \( \texttt{iszero} \tau \) evaluates to \texttt{true}.\(^9\) and if \( \tau \) represents any number other than 0, then \( \texttt{iszero} \tau \) evaluates to \texttt{false}.

Putting all this together, suppose we have a whole program that does some complicated calculation with numbers to yield a boolean result. If we replace all the numbers and arithmetic operations with lambda-terms representing them and evaluate the program, we will get the same result. Thus, in terms of their effects on the overall results of programs, there is no observable difference between the real numbers and their Church-numeral representation.

5.3 Formalities

For the rest of the chapter, we consider the syntax and operational semantics of the lambda-calculus in more detail. Most of the structure we need is closely analogous to what we saw in Chapter 3 (to avoid repeating that structure verbatim, we address here just the pure lambda-calculus, unadorned with booleans or numbers). However, the operation of substituting a term for a variable involves some surprising subtleties.

\(^9\) Strictly speaking, as we defined it, \( \texttt{iszero} \tau \) evaluates to a representation of \( \texttt{true} \) as another term, but let's eschew that distinction to simplify the present discussion. An analogous story can be given to explain in what sense the Church booleans represent the real ones.

5.3.1 Definition [Terms]: Let \( \mathcal{V} \) be a countable set of variable names. The set of terms is the smallest set \( \mathcal{T} \) such that

1. \( x \in \mathcal{T} \) for every \( x \in \mathcal{V} \);
2. if \( \tau_1 \in \mathcal{T} \) and \( x \in \mathcal{V} \), then \( \lambda x . \tau_1 \in \mathcal{T} \);
3. if \( \tau_1 \in \mathcal{T} \) and \( \tau_2 \in \mathcal{T} \), then \( \tau_1 \tau_2 \in \mathcal{T} \).

The size of a term \( \tau \) can be defined exactly as we did for arithmetic expressions in Definition 3.3.2. More interestingly, we can give a simple inductive definition of the set of variables appearing free in a lambda-term.

5.3.2 Definition: The set of free variables of a term \( \tau \), written \( \text{FV}(\tau) \), is defined as follows:

\[
\begin{align*}
\text{FV}(x) &= \{x\} \\
\text{FV}(\lambda x . \tau) &= \text{FV}(\tau) \setminus \{x\} \\
\text{FV}(\tau_1 \tau_2) &= \text{FV}(\tau_1) \cup \text{FV}(\tau_2)
\end{align*}
\]

5.3.3 Exercise [★★★]: Give a careful proof that \( |\text{FV}(\tau)| \leq \text{size}(\tau) \) for every term \( \tau \).

Substitution

The operation of substitution turns out to be quite tricky, when examined in detail. In this book, we shall actually use two different definitions, each optimized for a different purpose. The first, introduced in this section, is compact and intuitive, and works well for examples and in mathematical definitions and proofs. The second, developed in Chapter 6, is notionally heavier, depending on an alternative "de Bruijn presentation" of terms in which named variables are replaced by numeric indices, but is more convenient for the concrete ML implementations discussed in later chapters.

It is instructive to arrive at a definition of substitution via a couple of wrong attempts. First, let's try the most naive possible recursive definition. Formally, we are defining a function \( [x \mapsto s] \) by induction over its argument \( \tau \).

\[
\begin{align*}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \quad \text{if } x = y \\
[x \mapsto s](\lambda y . \tau) &= \lambda y . [x \mapsto s] \tau_1 \\
[x \mapsto s](\tau_1 \tau_2) &= ([x \mapsto s] \tau_1) ([x \mapsto s] \tau_2)
\end{align*}
\]
This definition works fine for most examples. For instance, it gives
\[ (x \rightarrow (\lambda z . z w))(\lambda y . x) = \lambda y . \lambda z . z w, \]
which matches our intuitions about how substitution should behave. However, if we are unlucky with our choice of bound variable names, the definition breaks down. For example:
\[ (x \rightarrow y)(\lambda x . x) = \lambda x . y \]
This conflicts with the basic intuition about functional abstractions that the names of bound variables do not matter—the identity function is exactly the same whether we write it \( \lambda x . x \) or \( \lambda y . y \) or \( \lambda \text{franz} . \text{franz} \). If these do not behave exactly the same under substitution, then they will not behave the same under reduction either, which seems wrong.

Clearly, the first mistake that we've made in the naive definition of substitution is that we have not distinguished between free occurrences of a variable \( x \) in a term \( \tau \) (which should get replaced during substitution) and bound ones, which should not. When we reach an abstraction binding the name \( x \) inside of \( \tau \), the substitution operation should stop. This leads to the next attempt:

\[
\begin{align*}
(x \rightarrow s) x &= s & \text{if } y \neq x \\
(x \rightarrow s) y &= y & \text{if } y \neq x \\
(x \rightarrow s)(\lambda y . \tau) &= (\lambda y . (x \rightarrow s) \tau) & \text{if } y \neq x \\
(x \rightarrow s)(\tau_1 \tau_2) &= ((x \rightarrow s) \tau_1)(x \rightarrow s) \tau_2 & \text{if } y \neq x
\end{align*}
\]
This is better, but still not quite right. For example, consider what happens when we substitute the term \( z \) for the variable \( x \) in the term \( \lambda x . x \):
\[ (x \rightarrow z)(\lambda x . x) = \lambda x . z \]
This time, we have made essentially the opposite mistake: we've turned the constant function \( \lambda x . x \) into the identity function! Again, this occurred only because we happened to choose \( z \) as the name of the bound variable in the constant function, so something is clearly still wrong.

This phenomenon of free variables in a term \( \tau \) becoming bound when \( s \) is naively substituted into a term \( \tau \) is called variable capture. To avoid it, we need to make sure that the bound variable names of \( \tau \) are kept distinct from the free variable names of \( s \). A substitution operation that does this correctly is called capture-avoiding substitution. (This is almost always what is meant by the unqualified term "substitution.") We can achieve the desired effect by adding another side condition to the second clause of the abstraction case:

\[
\begin{align*}
(x \rightarrow s) x &= s & \text{if } y \neq x \\
(x \rightarrow s) y &= y & \text{if } y \neq x \\
(x \rightarrow s)(\lambda y . \tau_1) &= (\lambda y . (x \rightarrow s) \tau_1) & \text{if } y \neq x \text{ and } y \notin \text{FV}(s) \\
(x \rightarrow s)(\tau_1 \tau_2) &= ((x \rightarrow s) \tau_1)(x \rightarrow s) \tau_2 & \text{if } y \neq x
\end{align*}
\]
Now we are almost there: this definition of substitution does the right thing when \( \text{it does anything at all} \). The problem now is that our last fix has changed substitution into a partial operation. For example, the new definition does not give any result at all for \( (x \rightarrow y z)(\lambda y . x y) \): the bound variable \( y \) of the term being substituted into is not equal to \( x \), but it does appear free in \( y z \), so none of the clauses of the definition apply.

One common fix for this last problem in the type systems and lambda-calculus literature is to work with terms "up to renaming of bound variables." (Church used the term alpha-conversion for the operation of consistently renaming a bound variable in a term. This terminology is still common—we could just as well say that we are working with terms "up to alpha-conversion."
5 The Untyped Lambda-Calculus

5.4 Notes

fails, of course, when we add other constructs such as primitive booleans to the language, since these introduce forms of values other than abstractions.

5.3.7 Exercise [••−]: Exercise 3.5.16 gave an alternative presentation of the operational semantics of booleans and arithmetic expressions in which stuck terms are defined to evaluate to a special constant wrong. Extend this semantics to ANB.

5.3.8 Exercise [•+]: Exercise 4.2.2 introduced a "big-step" style of evaluation for arithmetic expressions, where the basic evaluation relation is "term t evaluates to final result v." Show how to formulate the evaluation rules for lambda-terms in the big-step style.

5.4 Notes

The untyped lambda-calculus was developed by Church and his co-workers in the early 1920s and 30s (Church, 1941). The standard text for all aspects of the untyped lambda-calculus is Barendregt (1984); Hindley and Seldin (1986) is less comprehensive, but more accessible. Barendregt’s article (1990) in the Handbook of Theoretical Computer Science is a compact survey. Material on lambda-calculus can also be found in many textbooks on functional programming languages (e.g. Abelson and Sussman, 1985; Friedman, Wand, and Haynes, 2001; Peyton Jones and Lester, 1992) and programming language semantics (e.g. Schmidt, 1986; Gunter, 1992; Winskel, 1993; Mitchell, 1996). A systematic method for encoding a wide variety of data structures as lambda-terms can be found in 80hm and Berarducci (1985).

Despite its name, Curry denied inventing the idea of currying. It is commonly credited to Schönfinkel (1924), but the underlying idea was familiar to a number of 19th-century mathematicians, including Frege and Cantor.

There may, indeed, be other applications of the system than its use as a logic.
—Alonzo Church, 1932