The Untyped Lambda-Calculus

This chapter reviews the definition and some basic properties of the untyped or pure lambda-calculus, the underlying "computational substrate" for most of the type systems described in the rest of the book.

In the mid 1960s, Peter Landin observed that a complex programming language can be understood by formulating it as a tiny core calculus capturing the language's essential mechanisms, together with a collection of convenient derived forms whose behavior is understood by translating them into the core (Landin 1964, 1965, 1966; also see Tennent 1981). The core language used by Landin was the lambda-calculus, a formal system invented in the 1920s by Alonzo Church (1936, 1941), in which all computation is reduced to the basic operations of function definition and application. Following Landin's insight, as well as the pioneering work of John McCarthy on Lisp (1959, 1981), the lambda-calculus has seen widespread use in the specification of programming language features, in language design and implementation, and in the study of type systems. Its importance arises from the fact that it can be viewed simultaneously as a simple programming language in which computations can be described and as a mathematical object about which rigorous statements can be proved.

The lambda-calculus is just one of a large number of core calculi that have been used for similar purposes. The pi-calculus of Milner, Parrow, and Walker (1992, 1991) has become a popular core language for defining the semantics of message-based concurrent languages, while Abadi and Cardelli's object calculus (1996) distills the core features of object-oriented languages. Most of the concepts and techniques that we will develop for the lambda-calculus can be transferred quite directly to these other calculi. One case study along these lines is developed in Chapter 19.

The examples in this chapter are terms of the pure untyped lambda-calculus, $\lambda$ (Figure 5-1), or of the lambda-calculus extended with booleans and arithmetic operations, $\lambda_{\text{BN}}$ (3-2). The associated OCaml implementation is full untyped.
The lambda-calculus can be enriched in a variety of ways. First, it is often convenient to add special concrete syntax for features like numbers, tuples, records, etc., whose behavior can already be simulated in the core language. More interestingly, we can add more complex features such as mutable reference cells or nonlocal exception handling, which can be modeled in the core language only by using rather heavy translations. Such extensions lead eventually to languages such as ML (Gordon, Milner, and Wadsworth, 1979; Milner, Tofte, and Harper, 1990; Weis, Aponte, Laville, Mauny, and Suárez, 1989; Milner, Tofte, Harper, and MacQueen, 1997), Haskell (Hudak et al., 1992), or Scheme (Sussman and Steele, 1975; Kelsey, Clinger, and Rees, 1998). As we shall see in later chapters, extensions to the core language often involve extensions to the type system as well.

5.1 Basics

Procedural (or functional) abstraction is a key feature of essentially all programming languages. Instead of writing the same calculation over and over, we write a procedure or function that performs the calculation generically, in terms of one or more named parameters, and then instantiate this function as needed, providing values for the parameters in each case. For example, it is second nature for a programmer to take a long and repetitive expression like

\[(5\times 4\times 3\times 2\times 1) + (7\times 6\times 5\times 4\times 3\times 2\times 1) - (3\times 2\times 1)\]

and rewrite it as \(\text{factorial}(5) + \text{factorial}(7) - \text{factorial}(3)\), where:

\[\text{factorial}(n) = \begin{cases} n & \text{if } n \neq 0 \\ \text{factorial}(n-1) & \text{otherwise} \end{cases}\]

For each nonnegative number \(n\), instantiating the function \(\text{factorial}\) with the argument \(n\) yields the factorial of \(n\) as result. If we write \("\lambda n \ldots"\) as a shorthand for "the function that, for each \(n\), yields \(\ldots\)" we can restate the definition of \(\text{factorial}\) as:

\[\text{factorial} = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \ast \text{factorial}(n-1)\]

Then \(\text{factorial}(0)\) means "the function \((\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } \ldots)\) applied to the argument 0," that is, "the value that results when the argument variable \(n\) in the function body \((\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } \ldots)\) is replaced by \(0\)," that is, "if \(0=0\) then \(1\) else \(\ldots\)," that is, 1.

The lambda-calculus (or \(\lambda\)-calculus) embodies this kind of function definition and application in the purest possible form. In the lambda-calculus \textit{everything} is a function: the arguments accepted by functions are themselves functions and the result returned by a function is another function.

The syntax of the lambda-calculus comprises just three sorts of terms.1 A variable \(x\) by itself is a term; the abstraction of a variable \(x\) from a term \(t\), written \(\lambda x.t\), is a term; and the application of a term \(t_1\) to another term \(t_2\), written \(t_1 \; t_2\), is a term. These ways of forming terms are summarized in the following grammar.

\[
\begin{align*}
\text{terms:} & \\
\text{variable} & \\
\text{abstraction} & \\
\text{application} & \\
\end{align*}
\]

The subsections that follow explore some fine points of this definition.

Abstract and Concrete Syntax

When discussing the syntax of programming languages, it is useful to distinguish two levels of structure. The concrete syntax (or \textit{surface syntax}) of the language refers to the strings of characters that programmers directly read and write. Abstract syntax is a much simpler internal representation of programs as labeled trees (called \textit{abstract syntax trees} or \textit{ASTs}). The tree representation renders the structure of terms immediately obvious, making it a natural fit for the complex manipulations involved in both rigorous language definitions (and proofs about them) and the internals of compilers and interpreters.

The transformation from concrete to abstract syntax takes place in two stages. First, a lexical analyzer (or lexer) converts the string of characters written by the programmer into a sequence of \textit{tokens}—identifiers, keywords, constants, punctuation, etc. The lexer removes comments and deals with issues such as whitespace and capitalization conventions, and formats for numeric and string constants. Next, a \textit{parser} transforms this sequence of tokens into an abstract syntax tree. During parsing, various conventions such as operator precedence and associativity reduce the need to clutter surface programs with parentheses to explicitly indicate the structure of compound expressions. For example, \(*\) binds more tightly than +, so the parser interprets the unparenthesized expression \((a + b) * c\) as \((a + (b * c))\) rather than \((a + b) * c\).

---

1. The phrase \(\lambda\)-term is used to refer to arbitrary terms in the lambda-calculus. Lambda-terms beginning with a \(\lambda\) are often called \(\lambda\)-abstractions.

2. Definitions of full-blown languages sometimes use even more levels. For example, following Lambda, it is often useful to define the behaviors of some languages constructs as derived forms, by translating them into combinations of other, more basic, features. The restricted sublanguage containing just these core features is then called the \textit{internal language} (or IL), while the full language including all derived forms is called the \textit{external language} (EL). The transformation from EL to IL is (at least conceptually) performed in a separate pass, following parsing. Derived forms are discussed in Section 11.3.
The sized expression $1 \cdot 2 \cdot 3$ as the abstract syntax tree to the left below rather than the one to the right:

```
     +
    /|
   1 3
  /  |
 2  1 2
```

The focus of attention in this book is on abstract, not concrete, syntax. Grammars like the one for lambda-terms above should be understood as describing legal tree structures, not strings of tokens or characters. Of course, when we write terms in examples, definitions, theorems, and proofs, we will need to express them in a concrete, linear notation, but we always have their underlying abstract syntax trees in mind.

To save writing too many parentheses, we adopt two conventions when writing lambda-terms in linear form. First, application associates to the left—that is, $s \; t \; u$ stands for the same tree as $(s \; t) \; u$:

```
apply
   /
  /|
 s  t
```

Second, the bodies of abstractions are taken to extend as far to the right as possible, so that, for example, $\lambda x. \; \lambda y. \; x \; y \; x$ stands for the same tree as $\lambda x. \; (\lambda y. \; (x \; y) \; x)$:

```
lambda
    |
   /|
  s  t
```

Variables and Metavariables

Another subtlety in the syntax definition above concerns the use of metavariables. We will continue to use the metavariable $t$ (as well as $s$, and $u$, with or without subscripts) to stand for an arbitrary term. Similarly, $x$ (as well as $y$ and $z$) stands for an arbitrary variable. Note, here, that $x$ is a metavariable ranging over variables! To make matters worse, the set of short names is limited, and we will also want to use $x$, $y$, etc. as object-language variables. In such cases, however, the context will always make it clear which is which. For example, in a sentence like "The term $\lambda x. \; \lambda y. \; x \; y$ has the form $\lambda z. \; s$, where $z = x$ and $s = \lambda y. \; x \; y", the names $z$ and $s$ are metavariables, whereas $x$ and $y$ are object-language variables.

Scope

A final point we must address about the syntax of the lambda-calculus is the scope of variables.

An occurrence of the variable $x$ is said to be bound when it occurs in the body $t$ of an abstraction $\lambda x. \; t$. (More precisely, it is bound by this abstraction. Equivalently, we can say that $\lambda x$ is a binder whose scope is $t$.) An occurrence of $x$ is free if it appears in a position where it is not bound by an enclosing abstraction on $x$. For example, the occurrences of $x$ in $x \; y$ and $\lambda y. \; x \; y$ are free, while the ones in $\lambda x. \; s$ and $\lambda z. \; \lambda y. \; s \; (x \; y) z$ are bound. In $(\lambda x. \; s) \; x$, the first occurrence of $x$ is bound and the second is free.

A term with no free variables is said to be closed; closed terms are also called combinators. The simplest combinator, called the identity function, $\text{id} = \lambda x. \; x$; does nothing but return its argument.

Operational Semantics

In its pure form, the lambda-calculus has no built-in constants or primitive operators—no numbers, arithmetic operations, conditionals, records, loops, sequencing, I/O, etc. The sole means by which terms "compute" is the application of functions to arguments (which themselves are functions). Each step in the computation consists of rewriting an application whose left-hand component is an abstraction, by substituting the right-hand component for the bound variable in the abstraction's body. Graphically, we write

$$(\lambda x. \; t_2) \; t_1 \rightarrow [x \rightarrow t_2] \; t_1,$$

where $[x \rightarrow t_2] \; t_1$ means "the term obtained by replacing all free occurrences of $x$ in $t_1$ by $t_2".$ For example, the term $(\lambda x. \; y) \; y$ evaluates to $y$ and

---

3 Naturally, in this chapter, $t$ ranges over lambda-terms, not arithmetic expressions. Throughout the book, $t$ will always range over the terms of calculus under discussion at the moment. A footnote on the first page of each chapter specifies which system this is.
the term \((\lambda x. x (\lambda x.x)) (\text{red} x)\) evaluates to \(\text{red} x\). Following Church, a term of the form \((\lambda x. x) \text{red} x\) is called a redex ("reducible expression"), and the operation of rewriting a redex according to the above rule is called beta-reduction.

Several different evaluation strategies for the lambda-calculus have been studied over the years by programming language designers and theorists. Each strategy defines which redex or redexes in a term can fire on the next step of evaluation.4

- **Under full beta-reduction**, any redex may be reduced at any time. At each step we pick some redex, anywhere inside the term we are evaluating, and reduce it. For example, consider the term

\((\lambda x. x) ((\lambda x.x) (\lambda x. x) z))\),

which we can write more readable as \(\text{id} (\lambda x. x) z\). This term contains three redexes:

\[
\begin{align*}
\text{id} (\lambda x. x) z \\
\text{id} (\lambda x. x) (\text{id} (\lambda x. x) z) \\
\text{id} (\lambda x. x) (\lambda x. x) z
\end{align*}
\]

Under full beta-reduction, we might choose, for example, to begin with the innermost redex, then do the one in the middle, then the outermost:

\[
\begin{align*}
\text{id} (\lambda x. x) z & \rightarrow \text{id} (\lambda x. x) z \\
& \rightarrow \lambda x. x z \\
& \rightarrow \lambda x. z \\
& \rightarrow \lambda x. z
\end{align*}
\]

- **Under the normal order strategy**, the leftmost, outermost redex is always reduced first. Under this strategy, the term above would be reduced as follows:

\[
\begin{align*}
\text{id} (\lambda x. x) z & \rightarrow \text{id} (\lambda x. x) z \\
& \rightarrow \lambda x. x z \\
& \rightarrow \lambda x. z \\
& \rightarrow \lambda x. z
\end{align*}
\]

Under this strategy (and the ones below), the evaluation relation is actually a partial function: each term \(t\) evaluates in one step to at most one term \(t'\).

- The **call by name** strategy is yet more restrictive, allowing no reductions inside abstractions. Starting from the same term, we would perform the first two reductions as under normal-order, but then stop before the last and regard \(\lambda x. \text{id} x\) as a normal form:

\[
\begin{align*}
\text{id} (\lambda x. \text{id} x) & \rightarrow \text{id} (\lambda x. \text{id} z) \\
& \rightarrow \lambda x. \text{id} z
\end{align*}
\]

Variants of call by name have been used in some well-known programming languages, notably Algol-60 (Naur et al., 1963) and Haskell (Hudak et al., 1992). Haskell actually uses an optimized version known as call by need (Wadsworth, 1971; Ariola et al., 1995) that, instead of re-evaluating an argument each time it is used, overwrites all occurrences of the argument with its value the first time it is evaluated, avoiding the need for subsequent re-evaluation. This strategy demands that we maintain some sharing in the run-time representation of terms—in effect, it is a reduction relation on abstract syntax graphs, rather than syntax trees.

- Most languages use a **call by value** strategy, in which only outermost redexes are reduced and where a redex is reduced only when its right-hand side has already been reduced to a value—a term that is finished computing and cannot be reduced any further.5 Under this strategy, our example term reduces as follows:

\[
\begin{align*}
\text{id} (\lambda x. \text{id} x) & \rightarrow \text{id} (\lambda x. \text{id} z) \\
& \rightarrow \lambda x. \text{id} z \\
& \rightarrow \lambda x. z
\end{align*}
\]

The call-by-value strategy is strict, in the sense that the arguments to functions are always evaluated, whether or not they are used by the body of the function. In contrast, non-strict (or lazy) strategies such as call-by-name and call-by-need evaluate only the arguments that are actually used.

---

4. Some people use the terms "reduction" and "evaluation" synonymously. Others use "evaluation" only for strategies that involve some notion of "value" and "reduction" otherwise.

5. In the present bare-bones calculus, the only values are lambda-abstractions. Richer calculi will include other values: numeric and boolean constants, strings, tuples of values, records of values, lists of values, etc.
5.2 Programming in the Lambda-Calculus

The lambda-calculus is much more powerful than its tiny definition might suggest. In this section, we develop a number of standard examples of programming in the lambda-calculus. These examples are not intended to suggest that the lambda-calculus should be taken as a full-blown programming language in its own right—all widely used high-level languages provide clearer and more efficient ways of accomplishing the same tasks—but rather are intended as warm-up exercises to get the feel of the system.

Multiple Arguments

To begin, observe that the lambda-calculus provides no built-in support for multi-argument functions. Of course, this would not be hard to add, but it is even easier to achieve the same effect using higher-order functions that yield functions as results. Suppose that s is a term involving two free variables x and y and that we want to write a function f that, for each pair (v,w) of arguments, yields the result of substituting v for x and w for y in s. Instead of writing f = \( \lambda(x,y).s \), as we might in a richer programming language, we write f = \( \lambda x. \lambda y. s \). That is, f is a function that, given a value v for x, yields a function that, given a value w for y, yields the desired result. We then apply f to its arguments one at a time, writing f v w (i.e., \( f(v) w \)), which reduces to \( (\lambda y. (x \rightarrow v) y) w \) and thence to \( (y \rightarrow w)(x \rightarrow v) s \). This transformation of multi-argument functions into higher-order functions is called currying in honor of Haskell Curry, a contemporary of Church.

Church Booleans

Another language feature that can easily be encoded in the lambda-calculus is boolean values and conditionals. Define the terms tru and fls as follows:

\[
\text{tru} = \lambda t. \text{Af. t}; \\
\text{fls} = \lambda t. \text{Af. f};
\]

The terms tru and fls can be viewed as representing the boolean values "true" and "false," in the sense that we can use these terms to perform the operation of testing the truth of a boolean value. In particular, we can use application to define a combinator test with the property that test b v w reduces to v when b is tru and reduces to w when b is fls.

\[
\text{test} = \lambda l. \lambda m. \lambda n. l \text{ m n};
\]

The test combinator does not actually do much: test b v w just reduces to b v w. In effect, the boolean b itself is the conditional: it takes two arguments and chooses the first (if it is tru) or the second (if it is fls). For example, the term test tru v w reduces as follows:

\[
\text{test tru v w} = (\lambda l. \lambda m. \lambda n. l \text{ m n}) \text{ tru v w} \quad \text{by definition} \\
\to \quad (\lambda m. \lambda n. \text{tru m n}) v w \quad \text{reducing the underlined redex} \\
\to \quad (\lambda n. \text{tru v n}) w \quad \text{reducing the underlined redex} \\
\to \quad \text{tru v w} \quad \text{by definition} \\
\to \quad \text{fls v} \quad \text{reducing the underlined redex} \\
\to \quad v \quad \text{reducing the underlined redex}
\]

We can also define boolean operators like logical conjunction as functions:

\[
\text{and} = \lambda b. \lambda c. b \text{ c fls};
\]

That is, and is a function that, given two boolean values b and c, returns c if b is tru and fls if b is fls; thus and b c yields tru if both b and c are tru and fls if either b or c is fls.

\[
\text{and tru}; \\
\rightarrow \quad (\lambda t. \lambda f. t) \\
\rightarrow \quad \text{and fls}; \\
\rightarrow \quad (\lambda t. \lambda f. f)
\]

5.2.1 Exercises [★]: Define logical or and not functions.
5.2 Programming in the Lambda-Calculus

Pairs

Using booleans, we can encode pairs of values as terms.

\[ \text{pair} = \lambda f. \lambda s. \lambda b. \lambda f. s; \]
\[ \text{fst} = \lambda p. p \text{true}; \]
\[ \text{snd} = \lambda p. p \text{false}; \]

That is, \( \text{pair} v w \) is a function that, when applied to a boolean value \( b \), applies \( v \) to \( v \) and \( w \). By the definition of booleans, this application yields \( v \) if \( b \) is \( \text{true} \) and \( w \) if \( b \) is \( \text{false} \), so the first and second projection functions \( \text{fst} \) and \( \text{snd} \) can be implemented simply by supplying the appropriate boolean. To check that \( \text{fst} (\text{pair} v w) = v \), calculate as follows:

\[
\text{fst} (\text{pair} v w) \\
\quad = \text{fst} ((\lambda f. \lambda s. \lambda b. \lambda f. s) v w) \quad \text{by definition} \\
\quad = \text{fst} ((\lambda s. \lambda b. \lambda f. s) v w) \quad \text{reducing the underlined redex} \\
\quad = \text{fst} (\lambda b. \lambda f. s) v w) \quad \text{by definition} \\
\quad = (\lambda b. \lambda f. s) (\lambda f. s) v w) \quad \text{reducing the underlined redex} \\
\quad = (\lambda f. s) v w) \quad \text{reducing the underlined redex} \\
\quad = v \quad \text{as before.}
\]

Church Numerals

Representing numbers by lambda-terms is only slightly more intricate than what we have just seen. Define the Church numerals \( c_0, c_1, c_2, \) etc., as follows:

\[ c_0 = \lambda s. \lambda z. z; \]
\[ c_1 = \lambda s. \lambda z. s z; \]
\[ c_2 = \lambda s. \lambda z. s (s z); \]
\[ c_n = \lambda s. \lambda z. s (s (s \cdots s z)) ; \]

That is, each number \( n \) is represented by a combinator \( c_n \) that takes two arguments, \( s \) and \( z \) (for "successor" and "zero"), and applies \( s \) \( n \) times, to \( z \). As with booleans and pairs, this encoding makes numbers into active entities: the number \( n \) is represented by a function that does something \( n \) times—a kind of active unary numeral.

(The reader may already have observed that \( c_0 \) and \( \text{false} \) are actually the same term. Similar “puns” are common in assembly languages, where the same pattern of bits may represent many different values—an int, a float, an address, four characters, etc.—depending on how it is interpreted, and in low-level languages such as C, which also identifies \( 0 \) and \( \text{false} \).)

We can define the successor function on Church numerals as follows:

\[ \text{scc} = \lambda n. \lambda s. \lambda z. s (n s z); \]

The term \( \text{scc} \) is a combinator that takes a Church numeral \( n \) and returns another Church numeral—that is, it yields a function that takes arguments \( s \) and \( z \) and applies \( s \) \( n \) times to \( z \). We get the right number of applications of \( s \) to \( z \) by first passing \( s \) and \( z \) as arguments to \( n \), and then explicitly applying \( s \) one more time to the result.

5.2.2 Exercise [**]: Find another way to define the successor function on Church numerals.

Similarly, addition of Church numerals can be performed by a term \( \text{plus} \) that takes two Church numerals, \( m \) and \( n \), as arguments, and yields another Church numeral—i.e., a function—that accepts arguments \( s \) and \( z \), applies \( s \) \( m \) times to \( z \) (by passing \( s \) and \( z \) as arguments to \( m \)), and then applies \( s \) \( n \) times to the result:

\[ \text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z); \]

The implementation of multiplication uses another trick: since \( \text{plus} \) takes its arguments one at a time, applying it to just one argument \( n \) yields the function that adds \( n \) to whatever argument it is given. Passing this function as the first argument to \( m \) and \( c_0 \) as the second argument means "apply the function that adds \( n \) to its argument, \( n \) times, to zero," i.e., "add together \( n \) copies of \( n \)."

\[ \text{times} = \lambda m. \lambda n. \lambda s. \lambda z. m \text{plus} n \text{c}_0; \]

5.2.3 Exercise [**]: Is it possible to define multiplication on Church numerals without using \( \text{plus} \)?

5.2.4 Exercise [Recommended, **]: Define a term for raising one number to the power of another.

To test whether a Church numeral is zero, we must find some appropriate pair of arguments that will give us back this information—specifically, we must apply our numeral to a pair of terms \( z z \) and \( s s \) such that applying \( s s \) to \( z z \) one or more times yields \( \text{false} \) while not applying it at all yields \( \text{true} \). Clearly, we should take \( z z \) to be just \( \text{true} \). For \( s s \), we use a function that throws away its argument and always returns \( \text{false} \):