Chapter 4

Introduction to the $\lambda$-calculus

The $\lambda$-calculus notation for specifying functions is introduced. Various technical definitions are explained and motivated, including the rules of $\alpha$-, $\beta$- and $\eta$-conversion.

The $\lambda$-calculus (or lambda-calculus) is a theory of functions that was originally developed by the logician Alonzo Church as a foundation for mathematics. This work was done in the 1930s, several years before digital computers were invented. A little earlier (in the 1920s) Moses Schönfinkel developed another theory of functions based on what are now called 'combinators'. In the 1930s, Haskell Curry rediscovered and extended Schönfinkel's theory and showed that it was equivalent to the $\lambda$-calculus. About this time Kleene showed that the $\lambda$-calculus was a universal computing system; it was one of the first such systems to be rigorously analysed. In the 1950s John McCarthy was inspired by the $\lambda$-calculus to invent the programming language LISP. In the early 1960s Peter Landin showed how the meaning of imperative programming languages could be specified by translating them into the $\lambda$-calculus. He also invented an influential prototype programming language called ISWIM [42]. This introduced the main notations of functional programming and influenced the design of both functional and imperative languages. Building on this work, Christopher Strachey laid the foundations for the important area of denotational semantics [23,67]. Technical questions concerning Strachey's work inspired the mathematical logician Dana Scott to invent the theory of domains, which is now one of the most important parts of theoretical computer science. During the 1970s Peter Henderson and Jim Morris took up Landin's work and wrote a number of influential papers arguing that functional programming had important advantages for software engineering [29,28]. At about the same time David Turner proposed that Schönfinkel and Curry's combinators could be used as the machine code of computers for executing functional programming languages. Such computers could exploit mathematical properties of the
\( \lambda \)-calculus for the parallel evaluation of programs. During the 1980s several research groups took up Henderson’s and Turner’s ideas and started working on making functional programming practical by designing special architectures to support it, some of them with many processors.

We thus see that an obscure branch of mathematical logic underlies important developments in programming language theory, such as:

(i) The study of fundamental questions of computation.

(ii) The design of programming languages.

(iii) The semantics of programming languages.

(iv) The architecture of computers.

4.1 Syntax and semantics of the \( \lambda \)-calculus

The \( \lambda \)-calculus is a notation for defining functions. The expressions of the notation are called \( \lambda \)-\textit{expressions} and each such expression denotes a function. It will be seen later how functions can be used to represent a wide variety of data and data-structures including numbers, pairs, lists etc. For example, it will be demonstrated how an arbitrary pair of numbers \((x, y)\) can be represented as a \( \lambda \)-expression. As a notational convention, mnemonic names are assigned in bold or underlined to particular \( \lambda \)-expressions; for example \( \mathbf{1} \) is the \( \lambda \)-expression (defined in Section 5.3) which is used to represent the number one.

There are just three kinds of \( \lambda \)-expressions:

(i) Variables: \( x, y, z \) etc. The functions denoted by variables are determined by what the variables are bound to in the environment. Binding is done by abstractions (see 3 below). We use \( V, V_1, V_2 \) etc. for arbitrary variables.

(ii) Function applications or combinations: if \( E_1 \) and \( E_2 \) are \( \lambda \)-expressions, then so is \( (E_1 \, E_2) \); it denotes the result of applying the function denoted by \( E_1 \) to the function denoted by \( E_2 \). \( E_1 \) is called the \textit{rator} (from ‘operator’) and \( E_2 \) is called the \textit{rand} (from ‘operand’). For example, if \( (m, n) \) denotes a function representing the pair of numbers \( m \) and \( n \) (see Section 5.2) and \text{sum} denotes the addition function \(^1\) \( \lambda \)-calculus (see Section 5.5), then the application \((\text{sum}(m, n))\) denotes \( m+n \).

\(^1\) Note that \text{sum} is a \( \lambda \)-expression, whereas \( + \) is a mathematical symbol in the ‘metalanguage’ (i.e. English) that we are using for talking about the \( \lambda \)-calculus.
(iii) Abstractions: if \( V \) is a variable and \( E \) is a \( \lambda \)-expression, then \( \lambda V. E \) is an abstraction with bound variable \( V \) and body \( E \). Such an abstraction denotes the function that takes an argument \( a \) and returns as result the function denoted by \( E \) in an environment in which the bound variable \( V \) denotes \( a \). More specifically, the abstraction \( \lambda V. E \) denotes a function which takes an argument \( E' \) and transforms it into the thing denoted by \( E[E'/V] \) (the result of substituting \( E' \) for \( V \) in \( E \), see Section 4.8). For example, \( \lambda x. \text{sum}(x,1) \) denotes the add-one function.

Using BNF, the syntax of \( \lambda \)-expressions is just:

\[
< \text{\lambda-expression} > \ ::= \ < \text{variable} > \\
\quad | \ ( < \text{\lambda-expression} > \ < \text{\lambda-expression} > ) \\
\quad | \ ( \lambda < \text{variable} > . \ < \text{\lambda-expression} > )
\]

If \( V \) ranges over the syntax class \( < \text{variable} > \) and \( E_1, E_2, \ldots \) etc. range over the syntax class \( < \text{\lambda-expression} > \), then the BNF simplifies to:

\[
E ::= V | (E_1 E_2) | \lambda V. E \\
\quad \text{(variables)} \quad \text{(applications)} \quad \text{(combinations)}
\]

The description of the meaning of \( \lambda \)-expressions just given above is vague and intuitive. It took about 40 years for logicians (Dana Scott, in fact [66]) to make it rigorous in a useful way. We shall not be going into details of this.

Example: \( (\lambda x. x) \) denotes the 'identity function': \( ((\lambda x. x) \ E) = E. \Box \)

Example: \( (\lambda x. (\lambda f. (f \ x))) \) denotes the function which when applied to \( E \) yields \( (\lambda f. (f \ x))[E/x] \), i.e. \( (\lambda f. (f \ E)) \). This is the function which when applied to \( E' \) yields \( (f \ E)[E'/f] \) i.e. \( (E' \ E) \). Thus

\[
((\lambda x. (\lambda f. (f \ x))) \ E) = (\lambda f. (f \ E))
\]

and

\[
((\lambda f. (f \ E)) \ E') = (E' \ E)
\]

\( \Box \)

Exercise 41

Describe the function denoted by \( (\lambda x. (\lambda y. y)) \). \( \Box \)
Example: Section 5.3 describes how numbers can be represented by \( \lambda \)-expressions. Assume that this has been done and that \( 0, 1, 2, \ldots \) are \( \lambda \)-expressions which represent \( 0, 1, 2, \ldots \), respectively. Assume also that \textit{add} is a \( \lambda \)-expression denoting a function satisfying:

\[
  ((\textit{add } m) \ n) = m + n.
\]

Then \( (\lambda x. ((\textit{add } 1) \ x)) \) is a \( \lambda \)-expression denoting the function that transforms \( n \) to \( 1 + n \), and \( (\lambda x. (\lambda y. ((\textit{add } x) y))) \) is a \( \lambda \)-expression denoting the function that transforms \( m \) to the function which when applied to \( n \) yields \( m + n \), namely \( \lambda y. ((\textit{add } m) y) \). \( \square \)

The relationship between the function \textit{sum} in (ii) at the beginning of this section (page 60) and the function \textit{add} in the previous example is explained in Section 5.5.

### 4.2 Notational conventions

The following conventions help minimize the number of brackets one has to write.

1. Function application associates to the left, i.e. \( E_1 \ E_2 \cdots E_n \) means \( ((\cdots (E_1 \ E_2) \cdots) \ E_n) \). For example:

\[
\begin{align*}
E_1 \ E_2 & \quad \text{means} \quad (E_1 \ E_2) \\
E_1 \ E_2 \ E_3 & \quad \text{means} \quad ((E_1 \ E_2) \ E_3) \\
E_1 \ E_2 \ E_3 \ E_4 & \quad \text{means} \quad (((E_1 \ E_2) \ E_3) \ E_4)
\end{align*}
\]

2. \( \lambda \ V. E_1 \ E_2 \cdots E_n \) means \( \lambda V. (E_1 \ E_2 \cdots E_n) \). Thus the scope of \('\lambda V'\) extends as far to the right as possible.

3. \( \lambda V_1 \cdots V_n. E \) means \( \lambda V_1. (\cdots (\lambda V_n. E) \cdots) \). For example:

\[
\begin{align*}
\lambda x \ y. \ E & \quad \text{means} \quad (\lambda x. (\lambda y. E)) \\
\lambda x \ y \ z. \ E & \quad \text{means} \quad (\lambda x. (\lambda y. (\lambda z. E))) \\
\lambda x \ y \ z \ w. \ E & \quad \text{means} \quad (\lambda x. (\lambda y. (\lambda z. (\lambda w. E))))
\end{align*}
\]

Example: \( \lambda x \ y. \textit{add } y \ x \) means \( (\lambda x. (\lambda y. ((\textit{add } y) \ x))) \). \( \square \)
### 4.3 Free and bound variables

An occurrence of a variable \( V \) in a \( \lambda \)-expression is free if it is not within the scope of a ‘\( \lambda V \)’, otherwise it is bound. For example

\[
(\lambda x. y x)(\lambda y. x y)
\]

\[
\begin{array}{c|c|c|c}
\text{free} & \text{free} \\
\text{bound} & \text{bound}
\end{array}
\]

### 4.4 Conversion rules

In Chapter 5 it is explained how \( \lambda \)-expressions can be used to represent data objects like numbers, strings etc. For example, an arithmetic expression like \((2 + 3) \times 5\) can be represented as a \( \lambda \)-expression and its ‘value’ 25 can also be represented as a \( \lambda \)-expression. The process of ‘simplifying’ \((2 + 3) \times 5\) to 25 will be represented by a process called conversion (or reduction). The rules of \( \lambda \)-conversion described below are very general, yet when they are applied to \( \lambda \)-expressions representing arithmetic expressions they simulate arithmetical evaluation.

There are three kinds of \( \lambda \)-conversion called \( \alpha \)-conversion, \( \beta \)-conversion and \( \eta \)-conversion (the original motivation for these names is not clear). In stating the conversion rules the notation \( E[E'/V] \) is used to mean the result of substituting \( E' \) for each free occurrence of \( V \) in \( E \). The substitution is called valid if and only if no free variable in \( E' \) becomes bound in \( E[E'/V] \). Substitution is described in more detail in Section 4.8.
The rules of \( \lambda \)-conversion

- \( \alpha \)-conversion.
  Any abstraction of the form \( \lambda V. E \) can be converted to \( \lambda V'. E[V'/V] \) provided the substitution of \( V' \) for \( V \) in \( E \) is valid.

- \( \beta \)-conversion.
  Any application of the form \( (\lambda V. E_1) E_2 \) can be converted to \( E_1[E_2/V] \), provided the substitution of \( E_2 \) for \( V \) in \( E_1 \) is valid.

- \( \eta \)-conversion.
  Any abstraction of the form \( \lambda V. (E \, V) \) in which \( V \) has no free occurrence in \( E \) can be reduced to \( E \).

The following notation will be used:

- \( E_1 \rightarrow^\alpha E_2 \) means \( E_1 \) \( \alpha \)-converts to \( E_2 \).

- \( E_1 \rightarrow^\beta E_2 \) means \( E_1 \) \( \beta \)-converts to \( E_2 \).

- \( E_1 \rightarrow^\eta E_2 \) means \( E_1 \) \( \eta \)-converts to \( E_2 \).

In Section 4.4.4 below this notation is extended.

The most important kind of conversion is \( \beta \)-conversion; it is the one that can be used to simulate arbitrary evaluation mechanisms. \( \alpha \)-conversion is to do with the technical manipulation of bound variables and \( \eta \)-conversion expresses the fact that two functions that always give the same results on the same arguments are equal (see Section 4.7). The next three subsections give further explanation and examples of the three kinds of conversion (note that 'conversion' and 'reduction' are used below as synonyms).

4.4.1 \( \alpha \)-conversion

A \( \lambda \)-expression (necessarily an abstraction) to which \( \alpha \)-reduction can be applied is called an \( \alpha \)-redex. The term 'redex' abbreviates 'reducible expression'. The rule of \( \alpha \)-conversion just says that bound variables can be renamed provided no 'name-clashes' occur.
Examples

\[ \lambda x.\, x \vdash \lambda y.\, y \]

\[ \lambda x.\, f\, x \vdash \lambda y.\, f\, y \]

It is not the case that

\[ \lambda x.\, \lambda y.\, \text{add}\, x\, y \vdash \lambda y.\, \lambda y.\, \text{add}\, y\, y \]

because the substitution \((\lambda y.\, \text{add}\, x\, y)[y/x]\) is not valid since the \(y\) that replaces \(x\) becomes bound. \(\square\)

4.4.2 \(\beta\)-conversion

A \(\lambda\)-expression (necessarily an application) to which \(\beta\)-reduction can be applied is called a \(\beta\)-redex. The rule of \(\beta\)-conversion is like the evaluation of a function call in a programming language: the body \(E_1\) of the function \(\lambda V.\, E_1\) is evaluated in an environment in which the 'formal parameter' \(V\) is bound to the 'actual parameter' \(E_2\).

Examples

\[ (\lambda x.\, f\, x)\, E \vdash f\, E \]

\[ (\lambda x.\, (\lambda y.\, \text{add}\, x\, y))\, 3 \vdash \lambda y.\, \text{add}\, 3\, y \]

\[ (\lambda y.\, \text{add}\, 3\, y)\, 4 \vdash \text{add}\, 3\, 4 \]

It is not the case that

\[ (\lambda x.\, (\lambda y.\, \text{add}\, x\, y))\, (\text{square}\, y) \vdash \lambda y.\, \text{add}\, (\text{square}\, y)\, y \]

because the substitution \((\lambda y.\, \text{add}\, x\, y)[(\text{square}\, y)/x]\) is not valid, since \(y\) is free in \((\text{square}\, y)\) but becomes bound after substitution for \(x\) in \((\lambda y.\, \text{add}\, x\, y)\). \(\square\)

It takes some practice to parse \(\lambda\)-expressions according to the conventions of Section 4.2 so as to identify the \(\beta\)-redexes. For example, consider the application:

\[ (\lambda x.\, \lambda y.\, \text{add}\, x\, y)\, 3\, 4 \]
Putting in brackets according to the conventions expands this to:

\[((\lambda x. (\lambda y. ((\text{add } x) y)))\ 3)\ 4\]

which has the form:

\[((\lambda x. E)\ 3)\ 4\]

where

\[E = (\lambda y. \text{add } x\ y)\]

\((\lambda x. E)\ 3\) is a \(\beta\)-redex and could be reduced to \(E[3/x]\).

4.4.3 \(\eta\)-conversion

A \(\lambda\)-expression (necessarily an abstraction) to which \(\eta\)-reduction can be applied is called an \(\eta\)-redex. The rule of \(\eta\)-conversion expresses the property that two functions are equal if they give the same results when applied to the same arguments. This property is called extensionality and is discussed further in Section 4.7. For example, \(\eta\)-conversion ensures that \(\lambda x. (\sin x)\) and \(\sin\) denote the same function. More generally, \(\lambda V. (E\ V)\) denotes the function which when applied to an argument \(E'\) returns \((E\ V)[E'/V]\). If \(V\) does not occur free in \(E\) then \((E\ V)[E'/V] = (E\ E')\). Thus \(\lambda V. E\ V\) and \(E\) both yield the same result, namely \(E\ E'\), when applied to the same arguments and hence they denote the same function.

Examples

\[\lambda x. \text{add } x \rightarrow_{\eta} \text{add}\]

\[\lambda y. \text{add } x\ y \rightarrow_{\eta} \text{add } x\]

It is not the case that

\[\lambda x. \text{add } x\ x \rightarrow_{\eta} \text{add } x\]

because \(x\) is free in \(\text{add } x\). \(\square\)

4.4.4 Generalized conversions

The definitions of \(\rightarrow_{\alpha}\), \(\rightarrow_{\beta}\), and \(\rightarrow_{\eta}\) can be generalized as follows:

- \(E_1 \rightarrow_{\alpha} E_2\) if \(E_2\) can be got from \(E_1\) by \(\alpha\)-converting any subterm.
Conversion rules

- $E_1 \xrightarrow{\beta} E_2$ if $E_2$ can be got from $E_1$ by $\beta$-converting any subterm.
- $E_1 \xrightarrow{\eta} E_2$ if $E_2$ can be got from $E_1$ by $\eta$-converting any subterm.

Examples

$((\lambda x. \lambda y. \text{add } x \ y) \ 3) \ 4 \xrightarrow{\beta} (\lambda y. \text{add } 3 \ y) \ 4$

$(\lambda y. \text{add } 3 \ y) \ 4 \xrightarrow{\beta} \text{add } 3 \ 4$

The first of these is a $\beta$-conversion in the generalized sense because $(\lambda y. \text{add } 3 \ y) \ 4$ is obtained from $((\lambda x. \lambda y. \text{add } x \ y) \ 3) \ 4$ (which is not itself a $\beta$-redex) by reducing the subexpression $(\lambda x. \lambda y. \text{add } x \ y) \ 3$. We will sometimes write a sequence of conversions like the two above as:

$((\lambda x. \lambda y. \text{add } x \ y) \ 3) \ 4 \xrightarrow{\beta} (\lambda y. \text{add } 3 \ y) \ 4 \xrightarrow{\beta} \text{add } 3 \ 4$

Exercise 42

Which of the three $\beta$-reductions below are generalized conversions (i.e. reductions of subexpressions) and which are conversions in the sense defined on page 63?

(i) $(\lambda x. x) \ 1 \xrightarrow{\beta} 1$

(ii) $(\lambda y. y) ((\lambda x. x) \ 1) \xrightarrow{\beta} (\lambda y. y) \ 1 \xrightarrow{\beta} 1$

(iii) $(\lambda y. y) ((\lambda x. x) \ 1) \xrightarrow{\beta} (\lambda x. x) \ 1 \xrightarrow{\beta} 1$

In reductions (ii) and (iii) in the exercise above one starts with the same $\lambda$-expression, but reduce redexes in different orders.

An important property of $\beta$-reductions is that no matter in which order one does them, one always ends up with equivalent results. If there are several disjoint redexes in an expression, one can reduce them in parallel. Note, however, that some reduction sequences may never terminate. This is discussed further in connection with the normalization theorem of Chapter 7. It is a current hot research topic in 'fifth-generation computing' to design processors which exploit parallel evaluation to speed up the execution of functional programs.
4.5 Equality of \( \lambda \)-expressions

The three conversion rules preserve the meaning of \( \lambda \)-expressions, i.e. if \( E_1 \) can be converted to \( E_2 \) then \( E_1 \) and \( E_2 \) denote the same function. This property of conversion should be intuitively clear. It is possible to give a mathematical definition of the function denoted by a \( \lambda \)-expression and then to prove that this function is unchanged by \( \alpha \), \( \beta \), or \( \eta \)-conversion. Doing this is surprisingly difficult [67] and is beyond the scope of this book.

We will simply define two \( \lambda \)-expressions to be equal if they can be transformed into each other by a sequence of (forwards or backwards) \( \lambda \)-conversions. It is important to be clear about the difference between equality and identity. Two \( \lambda \)-expressions are identical if they consist of exactly the same sequence of characters; they are equal if one can be converted to the other. For example, \( \lambda x. x \) is equal to \( \lambda y. y \), but not identical to it. The following notation is used:

- \( E_1 \equiv E_2 \) means \( E_1 \) and \( E_2 \) are identical.
- \( E_1 = E_2 \) means \( E_1 \) and \( E_2 \) are equal.

Equality (\( \equiv \)) is defined in terms of identity (\( \equiv \)) and conversion (\( \alpha \rightarrow, \beta \rightarrow \) and \( \eta \rightarrow \)) as follows.

Equality of \( \lambda \)-expressions

If \( E \) and \( E' \) are \( \lambda \)-expressions then \( E = E' \) if \( E \equiv E' \) or there exist expressions \( E_1, E_2, \ldots, E_n \) such that:

1. \( E \equiv E_1 \)
2. \( E' \equiv E_n \)
3. For each \( i \) either
   (a) \( E_i \xrightarrow{\alpha} E_{i+1} \) or \( E_i \xrightarrow{\beta} E_{i+1} \) or \( E_i \xrightarrow{\eta} E_{i+1} \) or
   (b) \( E_{i+1} \xrightarrow{\alpha} E_i \) or \( E_{i+1} \xrightarrow{\beta} E_i \) or \( E_{i+1} \xrightarrow{\eta} E_i \).

Examples

\( (\lambda x. x) \, 1 = 1 \)
4.5 Equality of λ-expressions

\[ (λx. z) \ (λy. y \ 1) \ = \ 1 \]
\[ (λx. \ λy. \ add \ x \ y) \ 3 \ 4 \ = \ \text{add} \ 3 \ 4 \]

\[ \square \]

From the definition of equality it follows that:

(i) For any \( E \) it is the case that \( E = E \) (equality is reflexive).

(ii) If \( E = E' \), then \( E' = E \) (equality is symmetric).

(iii) If \( E = E' \) and \( E' = E'' \), then \( E = E'' \) (equality is transitive).

If a relation is reflexive, symmetric and transitive then it is called an equivalence relation. Thus \( = \) is an equivalence relation.

Another important property of equality is that if \( E_1 = E_2 \) and if \( E'_1 \) and \( E'_2 \) are two λ-expressions that only differ in that where one contains \( E_1 \) the other contains \( E_2 \), then \( E'_1 = E'_2 \). This property is called Leibnitz's law. It holds because the same sequence of reduction for getting from \( E_1 \) to \( E_2 \) can be used for getting from \( E'_1 \) to \( E'_2 \). For example, if \( E_1 = E_2 \), then by Leibnitz's law \( \text{λ}V \ . \ E_1 = \text{λ}V \ . \ E_2 \).

It is essential for the substitutions in the \( α \)- and \( β \)-reductions to be valid. The validity requirement disallows, for example, \( (λx. \ (λy. x)) \ 1 \ 2 \ → \ (λy. \ 1) \ 2 \ → \ 1 \)

But then since:

\[ (λx. \ (λy. x)) \ 1 \ 2 \ →_β (λy. \ 1) \ 2 \ →_β 1 \]

and

\[ (λy. \ (λy. y)) \ 1 \ 2 \ →_β (λy. \ y) \ 2 \ →_β 2 \]

one would be forced to conclude that \( 1 = 2 \). More generally by replacing \( 1 \) and \( 2 \) by any two expressions, it could be shown that any two expressions are equal!

**Exercise 43**

Find an example which shows that if substitutions in \( β \)-reductions are allowed to be invalid, then it follows that any two λ-expressions are equal.

\[ \square \]
Example: If $V_1, V_2, \ldots, V_n$ are all distinct and none of them occur free in any of $E_1, E_2, \ldots, E_n$, then

$$
\begin{align*}
(\lambda V_1 V_2 \cdots V_n. E) E_1 E_2 \cdots E_n &= ((\lambda V_1. (\lambda V_2 \cdots V_n. E))E_1) E_2 \cdots E_n \\
&\xrightarrow{\beta} ((\lambda V_2 \cdots V_n. E)[E_1/V_1]) E_2 \cdots E_n \\
&= (\lambda V_2 \cdots V_n. E[E_1/V_1]) E_2 \cdots E_n \\
&= \[E[E_1/V_1][E_2/V_2] \cdots [E_n/V_n] \]
\end{align*}
$$

□

Exercise 44
In the last example, where was the assumption used that $V_1, V_2, \ldots, V_n$ are all distinct and that none of them occur free in any of $E_1, E_2, \ldots, E_n$? □

Exercise 45
Find an example to show that if $V_1 = V_2$, then even if $V_2$ is not free in $E_1$, it is not necessarily the case that:

$$(\lambda V_1 V_2. E) E_1 E_2 = E[E_1/V_1][E_2/V_2]$$

□

Exercise 46
Find an example to show that if $V_1 \neq V_2$, but $V_2$ occurs free in $E_1$, then it is not necessarily the case that:

$$(\lambda V_1 V_2. E) E_1 E_2 = E[E_1/V_1][E_2/V_2]$$

□

4.6 The $\rightarrow$ relation

In the previous section $E_1 = E_2$ was defined to mean that $E_2$ could be obtained from $E_1$ by a sequence of forwards or backwards conversions. A special case of this is when $E_2$ is got from $E_1$ using only forwards conversions. This is written $E_1 \rightarrow E_2$. 
Definition of $\rightarrow$

If $E$ and $E'$ are $\lambda$-expressions, then $E \rightarrow E'$ if $E \equiv E'$ or there exist expressions $E_1, E_2, \ldots, E_n$ such that:

1. $E \equiv E_1$
2. $E' \equiv E_n$
3. For each $i$ either $E_i \xrightarrow{\alpha} E_{i+1}$ or $E_i \xrightarrow{\beta} E_{i+1}$ or $E_i \xrightarrow{\eta} E_{i+1}$.

Notice that the definition of $\rightarrow$ is just like the definition of $=$ on page 68 except that part (b) of 3 is missing.

Exercise 47
Find $E$, $E'$ such that $E = E'$ but it is not the case that $E \rightarrow E'$. $\square$

Exercise 48
[very hard!] Show that if $E_1 = E_2$, then there exists $E$ such that $E_1 \rightarrow E$ and $E_2 \rightarrow E$. (This property is called the Church-Rosser theorem. Some of its consequences are discussed in Chapter 7.) $\square$

4.7 Extensionality

Suppose $V$ does not occur free in $E_1$ or $E_2$ and

$$E_1 \ V = E_2 \ V$$

Then by Leibnitz's law (see page 69):

$$\lambda V. E_1 \ V = \lambda V. E_2 \ V$$

so by $\eta$-reduction applied to both sides

$$E_1 = E_2$$

It is often convenient to prove that two $\lambda$-expressions are equal using this property, i.e. to prove $E_1 = E_2$ by proving $E_1 \ V = E_2 \ V$ for some $V$ not occurring free in $E_1$ or $E_2$. We will refer to such proofs as being by extensionality.
Exercise 49
Show that

\[(\lambda f \, g \, x. \, f \, x \, (g \, x)) \, (\lambda x \, y. \, x) \, (\lambda x \, y. \, x) = \lambda x. \, x\]

4.8 Substitution

At the beginning of Section 4.4 \(E'[E'/V]\) was defined to mean the result of substituting \(E'\) for each free occurrence of \(V\) in \(E\). The substitution was said to be valid if no free variable in \(E'\) became bound in \(E[E'/V]\). In the definitions of \(\alpha\)- and \(\beta\)-conversion, it was stipulated that the substitutions involved must be valid. Thus, for example, it was only the case that

\[\left(\lambda V. \, E_1\right) \xrightarrow{\beta} E_1[E_2/V]\]

as long as the substitution \(E_1[E_2/V]\) was valid.

It is very convenient to extend the meaning of \(E[E'/V]\) so that we don't have to worry about validity. This is achieved by the definition below which has the property that for all expressions \(E\), \(E_1\) and \(E_2\) and all variables \(V\) and \(V'\):

\[\left(\lambda V. \, E_1\right) \, E_2 \rightarrow E_1[E_2/V] \quad \text{and} \quad \lambda V. \, E \rightarrow \lambda V'. \, E[V'/V]\]

To ensure this property holds, \(E[E'/V]\) is defined recursively on the structure of \(E\) as follows:
<table>
<thead>
<tr>
<th>$E$</th>
<th>$E[E'/V]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>$E'$</td>
</tr>
<tr>
<td>$V'$ (where $V \neq V'$)</td>
<td>$V'$</td>
</tr>
<tr>
<td>$E_1 ; E_2$</td>
<td>$E_1[E'/V] ; E_2[E'/V]$</td>
</tr>
<tr>
<td>$\lambda V. ; E_1$</td>
<td>$\lambda V. ; E_1$</td>
</tr>
<tr>
<td>$\lambda V'. ; E_1$ (where $V \neq V'$ and $V'$ is not free in $E'$)</td>
<td>$\lambda V'. ; E_1[E'/V]$</td>
</tr>
<tr>
<td>$\lambda V'. ; E_1$ (where $V \neq V'$ and $V'$ is free in $E'$)</td>
<td>$\lambda V''. ; E_1[V''/V'][E'/V]$ where $V''$ is a variable not free in $E'$ or $E_1$</td>
</tr>
</tbody>
</table>

This particular definition of $E[E'/V]$ is based on (but not identical to) the one in Appendix C of [4]. A LISP implementation of it is given in Chapter 12 on page 228.

To illustrate how this works consider $(\lambda y. \; y \; x)[y/x]$. Since $y$ is free in $y \; x$ we must use the last case of the table above. Since $x$ does not occur in $y \; x$ or $y$,

$$(\lambda y. \; y \; x)[y/x] \equiv \lambda z. \; (y \; x)[z/y][y/x] \equiv \lambda z. \; (z \; x)[y/x] \equiv \lambda z. \; z \; y$$

In the last line of the table above, the particular choice of $V''$ is not specified. Any variable not occurring in $E'$ or $E_1$ will do. In Chapter 12 an implementation of substitution in LISP is given.

A good discussion of substitution can be found in the book by Hindley and Seldin [31] where various technical properties are stated and proved. The following exercise is taken from that book.

**Exercise 59**

Use the table above to work out

(i) $(\lambda y. \; z \; (\lambda x. \; z))[((\lambda y. \; y \; x)/x]$.

(ii) $(y \; (\lambda z. \; x \; z))[((\lambda y. \; y \; z)/x]$.

$\Box$
It is straightforward, but rather tedious, to prove from the definition of $E[E'/V]$ just given that indeed

$$(\lambda V. E_1) E_2 \rightarrow E_1[E_2/V] \quad \text{and} \quad \lambda V. E \rightarrow \lambda V'. E[V'/V]$$

for all expressions $E$, $E_1$ and $E_2$ and all variables $V$ and $V'$.

In Chapter 8 it will be shown how the theory of combinators can be used to decompose the complexities of substitution into simpler operations. Instead of combinators it is possible to use the so-called nameless terms of De Bruijn [8]. De Bruijn's idea is that variables can be thought of as 'pointers' to the $\lambda$s that bind them. Instead of 'labelling' $\lambda$s with names (i.e. bound variables) and then pointing to them via these names, one can point to the appropriate $\lambda$ by giving the number of levels 'upwards' needed to reach it. For example, $\lambda x. \lambda y. x \ y \ y$ would be represented by $\lambda \lambda 2 \ 1$. As a more complicated example, consider the expression below in which we indicate the number of levels separating a variable from the $\lambda$ that binds it.

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

In De Bruijn's notation this is $\lambda \lambda 2 \ 1 \ 3 \ 1 \ 1$.

A free variable in an expression is represented by a number bigger than the depth of $\lambda$s above it; different free variables being assigned different numbers. For example,

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

would be represented by

$$\lambda (\lambda 1 \ 2 \ 3) \ 1 \ 2 \ 4$$

Since there are only two $\lambda$s above the occurrence of $3$, this number must denote a free variable; similarly there is only one $\lambda$ above the second occurrence of $2$ and the occurrence of $4$, so these too must be free variables. Note that $2$ could not be used to represent $w$ since this had already been used to represent the free $y$; we thus chose the first available number bigger than $2$ ($3$ was already in use representing $z$).

Care must be taken to assign big enough numbers to free variables. For example, the first occurrence of $z$ in $\lambda x. z \ (\lambda y. x)$ could be represented by $2$, but the second occurrence requires $3$; since they are the same variable we must use $3$. 
Example: With De Bruijn’s scheme $\lambda x. x (\lambda y. x y)$ would be represented by $\lambda 1(\lambda 2 1 1)$. □

Exercise 51
What $\lambda$-expression is represented by $\lambda 2(\lambda 2)$? □

Exercise 52
Describe an algorithm for computing the De Bruijn representation of the expression $E[E'/V]$ from the representations of $E$ and $E'$. □
Chapter 5

Representing Things in the \( \lambda \)-calculus

The representation in the \( \lambda \)-calculus of various data objects (e.g. numbers), data-structures (e.g. pairs) and useful functions (e.g. addition) is described. Definition by recursion using the fixed-point operator \( Y \) is explained. It is shown that all the recursive functions can be represented by suitable \( \lambda \)-expressions.

The \( \lambda \)-calculus appears at first sight to be a very primitive language. However, it can be used to represent most of the objects and structures needed for modern programming. The idea is to code these objects and structures in such a way that they have the required properties. For example, to represent the truth values \textit{true} and \textit{false} and the Boolean function \( \neg \) (\textit{not}), \( \lambda \)-expressions \textit{true}, \textit{false} and \textit{not} are devised with the properties that:

\[
\text{not true} = \text{false} \\
\text{not false} = \text{true}
\]

To represent the Boolean function \( \land \) (\textit{and}) a \( \lambda \)-expression and is devised such that:

\[
\begin{align*}
\text{and true true} &= \text{true} \\
\text{and true false} &= \text{false} \\
\text{and false true} &= \text{false} \\
\text{and false false} &= \text{false}
\end{align*}
\]

and to represent \( \lor \) (\textit{or}) an expression or such that:

\[
\begin{align*}
\text{or true true} &= \text{true} \\
\text{or true false} &= \text{true} \\
\text{or false true} &= \text{true} \\
\text{or false false} &= \text{false}
\end{align*}
\]
The $\lambda$-expressions used to represent things may appear completely unmotivated at first. However, the definitions are chosen so that they work together in unison.

We will write

$$\text{LET } \sim = \lambda\text{-expression}$$

to introduce $\sim$ as a new notation. Usually $\sim$ will just be a name such as true or and. Such names are written in bold face, or underlined, to distinguish them from variables. Thus, for example, true is a variable but true is the $\lambda$-expression $\lambda x. \lambda y. x$ (see Section 5.1 below) and 2 is a number but 2 is the $\lambda$-expression $\lambda f. \lambda x. f(f x)$ (see Section 5.3).

Sometimes $\sim$ will be a more complicated form like the conditional notation ($E \rightarrow E_1 \mid E_2$).

### 5.1 Truth-values and the conditional

This section defines $\lambda$-expressions true, false, not and ($E \rightarrow E_1 \mid E_2$) with the following properties:

- $\text{not } \text{true} = \text{false}$
- $\text{not } \text{false} = \text{true}$
- $(\text{true} \rightarrow E_1 \mid E_2) = E_1$
- $(\text{false} \rightarrow E_1 \mid E_2) = E_2$

The $\lambda$-expressions true and false represent the truth-values true and false, not represents the negation function $\neg$ and $(E \rightarrow E_1 \mid E_2)$ represents the conditional 'if $E$ then $E_1$ else $E_2$'.

There are infinitely many different ways of representing the truth-values and negation that work; the ones used here are traditional and have been developed over the years by logicians.

\begin{verbatim}
LET true = $\lambda x. \lambda y. x$
LET false = $\lambda x. \lambda y. y$
LET not = $\lambda t. t \text{false true}$
\end{verbatim}

It is easy to use the rules of $\lambda$-conversion to show that these definitions have the desired properties. For example:

- $\text{not true} = (\lambda t. t \text{false true}) \text{true}$ (definition of not)
5.1 Truth-values and the conditional

\[ = \text{true, false, true} \quad (\beta\text{-conversion}) \]
\[ = (\lambda x. \lambda y. x) \text{false, true, (definition of true)} \]
\[ = (\lambda y. \text{false}) \text{true, (\beta\text{-conversion})} \]
\[ = \text{false} \quad (\beta\text{-conversion}) \]

Similarly not \( \text{false} = \text{true} \).

Conditional expressions \((E \rightarrow E_1 \mid E_2)\) can be defined as follows:

\[
\text{LET} \quad (E \rightarrow E_1 \mid E_2) = (E \ E_1 \ E_2)
\]

This means that for any \(\lambda\)-expressions \(E, E_1\) and \(E_2\), \((E \rightarrow E_1 \mid E_2)\) stands for \((E \ E_1 \ E_2)\).

The conditional notation behaves as it should:

\[
(\text{true} \rightarrow E_1 \mid E_2) = \text{true} \ E_1 \ E_2
\]
\[ = (\lambda x. y. x) \ E_1 \ E_2 \]
\[ = E_1 \]

and

\[
(\text{false} \rightarrow E_1 \mid E_2) = \text{false} \ E_1 \ E_2
\]
\[ = (\lambda x. y. y) \ E_1 \ E_2 \]
\[ = E_2 \]

Exercise 53
Let \(\text{and}\) be the \(\lambda\)-expression \(\lambda x. y. (x \rightarrow y \mid \text{false})\). Show that:

\[ \text{and true true} = \text{true} \]
\[ \text{and true false} = \text{false} \]
\[ \text{and false true} = \text{false} \]
\[ \text{and false false} = \text{false} \]

Exercise 54
Devise a \(\lambda\)-expression \(\text{or}\) such that:

\[ \text{or true true} = \text{true} \]
\[ \text{or true false} = \text{true} \]
\[ \text{or false true} = \text{true} \]
\[ \text{or false false} = \text{false} \]
5.2 Pairs and tuples

The following abbreviations represent pairs and \( n \)-tuples in the \( \lambda \)-calculus.

\[
\begin{align*}
\text{LET } & \text{fst} = \lambda p. \ p \ \text{true} \\
\text{LET } & \text{snd} = \lambda p. \ p \ \text{false} \\
\text{LET } & (E_1, E_2) = \lambda f. \ f \ E_1 E_2
\end{align*}
\]

\((E_1, E_2)\) is a \( \lambda \)-expression representing an ordered pair whose first component (i.e. \( E_1 \)) is accessed with the function \( \text{fst} \) and whose second component (i.e. \( E_2 \)) is accessed with \( \text{snd} \). The following calculation shows how the various definitions co-operate together to give the right results.

\[
\begin{align*}
\text{fst} \ (E_1, E_2) & = (\lambda p. \ p \ \text{true}) \ (E_1, E_2) \\
& = (E_1, E_2) \ \text{true} \\
& = (\lambda f. \ f \ E_1 E_2) \ \text{true} \\
& = \text{true} \ E_1 E_2 \\
& = (\lambda x \ y \ z) \ E_1 E_2 \\
& = E_1
\end{align*}
\]

Exercise 55
Show that \( \text{snd}(E_1, E_2) = E_2 \).

A pair is a data-structure with two components. The generalization to \( n \) components is called an \( n \)-tuple and is easily defined in terms of pairs.

\[
\text{LET } (E_1, E_2, \ldots, E_n) = (E_1, (E_2, \ldots, (E_{n-1}, E_n), \ldots))
\]

\((E_1, \ldots, E_n)\) is an \( n \)-tuple with components \( E_1, \ldots, E_n \) and length \( n \). Pairs are 2-tuples. The abbreviations defined next provide a way of extracting the components of \( n \)-tuples.
\[
\begin{align*}
\text{LET } E^n_1 &= \text{fst } E \\
\text{LET } E^n_2 &= \text{fst(snd } E) \\
& \vdots \\
\text{LET } E^n_i &= \text{fst(snd(snd(\cdots(snd } E)\cdots)))} \quad (\text{if } i < n) \\
& \vdots \\
\text{LET } E^n_n &= \text{snd(\cdots(snd } E)\cdots))) \\
\end{align*}
\]

It is easy to see that these definitions work, for example:

\[
(E_1, E_2, \ldots, E_n)^n_1 = (E_1, (E_2, \ldots))^n_1 = \text{fst } E_1 = E_1
\]

\[
(E_1, E_2, \ldots, E_n)^n_2 = (E_1, (E_2, \ldots))^n_2 = \text{fst( snd } E_1, (E_2, \ldots)) = \text{fst } E_1 = E_2
\]

In general \((E_1, E_2, \ldots, E_n)^n_i = E_i\) for all \(i\) such that \(1 \leq i \leq n\).

**Convention**

We will usually just write \(E \downarrow i\) instead of \(E^n \downarrow i\) when it is clear from the context what \(n\) should be. For example,

\[
(E_1, \ldots, E_n) \downarrow i = E_i \quad (\text{where } 1 \leq i \leq n)
\]

### 5.3 Numbers

There are many ways to represent numbers by \(\lambda\)-expressions, each with their own advantages and disadvantages [72,40]. The goal is to define for each number \(n\) a \(\lambda\)-expression \(\text{n}\) that represents it. We also want to define
\(\lambda\)-expressions to represent the primitive arithmetical operations. For example, we will need \(\lambda\)-expressions \text{suc}, \text{pre}, \text{add} and \text{iszero} representing the successor function \((n \mapsto n + 1)\), the predecessor function \((n \mapsto n - 1)\), addition and a test for zero, respectively. These \(\lambda\)-expressions will represent the numbers correctly if they have the following properties:

\[
\text{suc } n = n + 1 \quad (\text{for all numbers } n)
\]

\[
\text{pre } n = n - 1 \quad (\text{for all numbers } n)
\]

\[
\text{add } m \ n = m + n \quad (\text{for all numbers } m \text{ and } n)
\]

\[
\text{iszero } 0 = \text{true}
\]

\[
\text{iszero } (\text{suc } n) = \text{false}
\]

The representation of numbers described here is the original one due to Church. In order to explain this it is convenient to define \(f^n x\) to mean \(n\) applications of \(f\) to \(x\). For example,

\[
f^5 x = f(f(f(f(f x))))
\]

By convention \(f^0 x\) is defined to mean \(x\). More generally:

\[
\begin{align*}
\text{LET } E^0 E' &= E' \\
\text{LET } E^n E' &= E(E(\cdots(E \ E')\cdots)) \\
&= \underbrace{E E_1 E_2 \cdots E_n}_{n \ E_x}
\end{align*}
\]

Note that \(E^n(E E') = E^{n+1} E' = E(E^n E')\); we will use the fact later.

Example

\[
f^4 x = f(f(f(f x))) = f(f^3 x) = f^3(f x)
\]

Using the notation just introduced we can now define Church's numerals. Notice how the definition of the \(\lambda\)-expression \(n\) below encodes a unary representation of \(n\).
\[
\begin{align*}
\text{LET } \lambda = \lambda f \cdot x \cdot x \\
\text{LET } 1 = \lambda f \cdot x \cdot f x \\
\text{LET } 2 = \lambda f \cdot x \cdot f(f x) \\
\vdots \\
\text{LET } n = \lambda f \cdot x \cdot f^n x \\
\vdots 
\end{align*}
\]

The representations of \text{suc}, \text{add} and \text{iszero} are now magically pulled out of a hat. The best way to see how they work is to think of them as operating on unary representations of numbers. The exercises that follow should help.

\[
\begin{align*}
\text{LET } \text{suc} = \lambda n \cdot f \cdot x \cdot n \cdot f(f x) \\
\text{LET } \text{add} = \lambda m \cdot n \cdot f \cdot x \cdot m \cdot f(n \cdot f x) \\
\text{LET } \text{iszero} = \lambda n \cdot (\lambda x. \text{false}) \cdot \text{true}
\end{align*}
\]

Exercise 56
Show:

(i) \text{suc } \lambda = 1 \\
(ii) \text{suc } 5 = 6 \\
(iii) \text{iszero } \lambda = \text{true} \\
(iv) \text{iszero } 5 = \text{false} \\
(v) \text{add } \lambda \lambda = 1 \\
(vi) \text{add } 2 \lambda = 5

\]

Exercise 57
Show for all numbers \( m \) and \( n \):

(i) \text{suc } \lambda = \lambda + 1 \\
(ii) \text{iszero } (\text{suc } \lambda) = \text{false} \\
(iii) \text{add } \lambda \lambda = \lambda

(iv) \( \text{add } 0 = m \)
(v) \( \text{add } n = m + n \)

\[ \square \]

The predecessor function is harder to define than the other primitive functions. The idea is that the predecessor of \( n \) is defined by using \( \lambda f \, x. \, f^n \, x \) (i.e. \( n \)) to obtain a function that applies \( f \) only \( n-1 \) times. The trick is to 'throw away' the first application of \( f \) in \( f^n \). To achieve this, we first define a function \( \text{prefn} \) that operates on pairs and has the property that:

(i) \( \text{prefn } f \, (\text{true}, x) = (\text{false}, x) \)
(ii) \( \text{prefn } f \, (\text{false}, x) = (\text{false}, f \, x) \)

From this it follows that:

(iii) \( (\text{prefn } f)^n \, (\text{false}, x) = (\text{false}, f^n \, x) \)
(iv) \( (\text{prefn } f)^n \, (\text{true}, x) = (\text{false}, f^{n-1} \, x) \quad \text{(if } n > 0 \text{)} \)

Thus \( n \) applications of \( \text{prefn} \) to \( (\text{true}, x) \) result in \( n-1 \) applications of \( f \) to \( x \). With this idea, the definition of the predecessor function \( \text{pre} \) is straightforward. Before giving it, here is the definition of \( \text{prefn} \):

\[
\text{LET } \text{prefn} = \lambda f \, p. \, (\text{false}, (\text{fst } p \rightarrow \text{snd } p \mid (f(\text{snd } p))))
\]

**Exercise 58**
Show \( \text{prefn } f \, (b, x) = (\text{false}, (b \rightarrow x \mid f \, x)) \) and hence:

(i) \( \text{prefn } f \, (\text{true}, x) = (\text{false}, x) \)
(ii) \( \text{prefn } f \, (\text{false}, x) = (\text{false}, f \, x) \)
(iii) \( (\text{prefn } f)^n \, (\text{false}, x) = (\text{false}, f^n \, x) \)
(iv) \( (\text{prefn } f)^n \, (\text{true}, x) = (\text{false}, f^{n-1} \, x) \quad \text{(if } n > 0 \text{)} \)

\[ \square \]

The predecessor function \( \text{pre} \) can now be defined.

\[
\text{LET } \text{pre} = \lambda n \, x. \, \text{snd } (n \, (\text{prefn } f) \, (\text{true}, x))
\]
It follows that if \( n > 0 \)
\[
\text{pre } n \ f x = \ \text{snd} \ (n \ (\text{prefn } f) \ (\text{true}, x)) \quad (\text{definition of pre})
\]
\[
= \ \text{snd} \ ((\text{prefn } f)^n \ (\text{true}, x)) \quad (\text{definition of } n)
\]
\[
= \ \text{snd} (\text{false}, f^{n-1} x) \quad (\text{by (v) above})
\]
\[
= f^{n-1} x
\]

hence by extensionality (Section 4.7 on page 71)
\[
\text{pre } n = \lambda f \ x. f^{n-1} x
\]
\[
= \frac{n-1}{\text{definition of } n-1}
\]

Exercise 59
Using the results of the previous exercise (or otherwise) show that

(i) \( \text{pre } (\text{suc } n) = n \)

(ii) \( \text{pre } 0 = 0 \)

\[\square\]

The numeral system in the next exercise is the one used in [4] and has some advantages over Church's (e.g. the predecessor function is easier to define).

Exercise 60

\[
\text{LET } \widehat{0} = \lambda z. z
\]
\[
\text{LET } \widehat{1} = \text{false, } \widehat{0}
\]
\[
\text{LET } \widehat{2} = \text{false, } \widehat{1}
\]
\[
\vdots
\]
\[
\text{LET } \widehat{n+1} = \text{false, } \widehat{n}
\]
\[
\vdots
\]

Devise \( \lambda \)-expressions \( \text{suc}, \text{iszero}, \text{pre} \) such that for all \( n \):

(i) \( \text{suc } \widehat{n} = \widehat{n+1} \)

(ii) \( \text{iszero } \widehat{0} = \text{true} \)

(iii) \( \text{iszero } (\text{suc } \widehat{n}) = \text{false} \)

(iv) \( \text{pre } (\text{suc } \widehat{n}) = \widehat{n} \)

\[\square\]
5.4 Definition by recursion

To represent the multiplication function in the λ-calculus we would like to define a λ-expression, \text{mult} say, such that:

\[
\text{mult} \; m \; n = \underbrace{\text{add} \; n \; (\cdots (\text{add} \; n \; 0 \cdots))}_{m \; \text{adds}}
\]

This would be achieved if \text{mult} could be defined to satisfy the equation:

\[
\text{mult} \; m \; n = (\text{iszero} \; m \rightarrow 0 \; | \; \text{add} \; n \; (\text{mult} \; (\text{pre} \; m) \; n))
\]

If this held then, for example,

\[
\text{mult} \; 2 \; 3 = (\text{iszero} \; 2 \rightarrow 0 \; | \; \text{add} \; 3 \; (\text{mult} \; (\text{pre} \; 2) \; 3))
\]

(by the equation)

\[
= \text{add} \; 3 \; (\text{mult} \; 1 \; 3)
\]

(by properties of \text{iszero}, the conditional and \text{pre})

\[
= \text{add} \; 3 \; (\text{iszero} \; 1 \rightarrow 0 \; | \; \text{add} \; 3 \; (\text{mult} \; (\text{pre} \; 1) \; 3))
\]

(by the equation)

\[
= \text{add} \; 3 \; (\text{add} \; 3 \; (\text{mult} \; 0 \; 3))
\]

(by properties of \text{iszero}, the conditional and \text{pre})

\[
= \text{add} \; 3 \; (\text{add} \; 3 \; (\text{iszero} \; 0 \rightarrow 0 \; | \; \text{add} \; 3 \; (\text{mult} \; (\text{pre} \; 0) \; 3)))
\]

(by the equation)

\[
= \text{add} \; 3 \; (\text{add} \; 3 \; 0)
\]

(by properties of \text{iszero} and the conditional)

The equation above suggests that \text{mult} be defined by:

\[
\text{mult} = \lambda m \; n. \; (\text{iszero} \; m \rightarrow 0 \; | \; \text{add} \; n \; (\text{mult} \; (\text{pre} \; m) \; n))
\]

N.B.

Unfortunately, this cannot be used to define \text{mult} because, as indicated by the arrow, \text{mult} must already be defined for the λ-expression to the right of the equals to make sense.

Fortunately, there is a technique for constructing λ-expressions that satisfy arbitrary equations. When this technique is applied to the equation above it gives the desired definition of \text{mult}. First define a λ-expression \text{Y} that, for any expression \text{E}, has the following odd property:

\[
\text{Y} \; E = E \; (\text{Y} \; E)
\]
This says that $Y\ E$ is unchanged when the function $E$ is applied to it. In general, if $E\ E' = E'$ then $E'$ is called a fixed point of $E$. A $\lambda$-expression $\text{Fix}$ with the property that $\text{Fix} E = E(\text{Fix} E)$ for any $E$ is called a fixed-point operator. There are known to be infinitely many different fixed-point operators [57]; $Y$ is the most famous one, and its definition is:

\[
\text{LET } Y = \lambda f. (\lambda x. f(x\ x)) (\lambda x. f(x\ x))
\]

It is straightforward to show that $Y$ is indeed a fixed-point operator:

\[
Y\ E = (\lambda f. (\lambda x. f(x\ x)) (\lambda x. f(x\ x)))\ E \quad \text{(definition of } Y) \\
= (\lambda x. \ E(x\ x)) (\lambda x. E(x\ x)) \quad \text{()\-conversion) } \\
= E ((\lambda x. E(x\ x)) (\lambda x. E(x\ x))) \quad \text{()\-conversion) } \\
= E (Y\ E) \quad \text{(the line before last)}
\]

This calculation shows that every $\lambda$-expression $E$ has a fixed point (namely $Y\ E$); this is sometimes referred to as the first fixed-point theorem. The second fixed-point theorem is introduced in Section 7.1.

Armed with $Y$, we can now return to the problem of solving the equation for $\text{mult}$. Suppose $\text{multfn}$ is defined by

\[
\text{LET } \text{multfn} = \lambda f\ n. (\text{iszero}\ m\rightarrow 0 | \text{add}\ n\ (f\ (\text{pre}\ m\ n))
\]

and then define $\text{mult}$ by:

\[
\text{LET } \text{mult} = Y\ \text{multfn}
\]

Then:

\[
\text{mult}\ m\ n = (Y\ \text{multfn})\ m\ n \quad \text{(definition of } \text{mult}) \\
= \text{multfn}\ (Y\ \text{multfn})\ m\ n \quad \text{(fixed-point property of } Y) \\
= \text{multfn}\ \text{mult}\ m\ n \quad \text{(definition of } \text{mult}) \\
= (\lambda f\ n. (\text{iszero}\ m\rightarrow 0 | \text{add}\ n\ (f\ (\text{pre}\ m\ n))))\ \text{mult}\ m\ n \quad \text{(definition of } \text{multfn}) \\
= (\text{iszero}\ m\rightarrow 0 | \text{add}\ n\ (\text{mult}\ (\text{pre}\ m\ n))) \quad (\beta\text{-conversion})
\]

An equation of the form $f\ x_1 \cdots x_n = E$ is called recursive if $f$ occurs free in $E$. $Y$ provides a general way of solving such equations. Start with an equation of the form:

\[
f\ x_1 \cdots x_n = \__f\__
\]
where \( \f \) is some \( \lambda \)-expression containing \( f \). To obtain \( f \) so that this equation holds define:

\[
\text{LET } f = Y (\lambda x_1 \ldots x_n. \f)
\]

The fact that the equation is satisfied can be shown as follows:

\[
f x_1 \ldots x_n = Y (\lambda x_1 \ldots x_n. \f) x_1 \ldots x_n
\]

(definition of \( f \))

\[
= (\lambda x_1 \ldots x_n. \f) (Y (\lambda x_1 \ldots x_n. \f)) x_1 \ldots x_n
\]

(fixed-point property)

\[
= (\lambda x_1 \ldots x_n. \f) f x_1 \ldots x_n
\]

(definition of \( f \))

\[
= f
\]

(\( \beta \)-conversion)

Exercise 61
Construct a \( \lambda \)-expression \( \text{eq} \) such that

\[
\text{eq } m \ n = (\text{iszero } m \rightarrow \text{iszero } n \mid \text{iszero } n \rightarrow \text{false} \mid \text{eq } \text{pre } m \text{ pre } n)
\]

\( \square \)

Exercise 62
Show that if \( Y_1 \) is defined by:

\[
\text{LET } Y_1 = Y (\lambda f. f f)
\]

then \( Y_1 \) is a fixed-point operator, i.e. for any \( E \):

\[
Y_1 E = E (Y_1 E)
\]

\( \square \)

The fixed-point operator in the next exercise is due to Turing (Barendregt [4], page 132).

Exercise 63
Show that \((\lambda x. y. y (x \ y)) \ (\lambda x. y. y (x \ y))\) is a fixed-point operator. \( \square \)

The next exercise also comes from Barendregt's book, where it is attributed to Klopf.

Exercise 64
Show that \( Y_2 \) is a fixed-point operator, where:

\[
\text{LET } L = \lambda abcdefghijklmnopqrstuvwxyz.
\]

\[
r(\text{this is a fixedpoint combinator})
\]

\[
\text{LET } Y_2 = L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L L
5.5 Functions with several arguments

Exercise 65
Is it the case that \( \forall f \rightarrow f(\forall f) \)? If so prove it; if not find a \( \lambda \)-expression \( \forall f \rightarrow f(\forall f) \). □

In the pure \( \lambda \)-calculus as defined on page 60, \( \lambda \)-expressions could only be applied to a single argument; however, this argument could be a tuple (see page 80). Thus one can write:

\[ E(E_1, \ldots, E_n) \]

which actually abbreviates:

\[ E(E_1, (E_2, \cdots (E_{n-1}, E_n) \cdots)) \]

For example, \( E(E_1, E_2) \) abbreviates \( E(\lambda f. f E_1 E_2) \).

5.5 Functions with several arguments

In conventional mathematical usage, the application of an \( n \)-argument function \( f \) to arguments \( x_1, \ldots, x_n \) would be written as \( f(x_1, \ldots, x_n) \). There are two ways of representing such applications in the \( \lambda \)-calculus:

(i) as \( (f \ x_1 \ldots x_n) \), or

(ii) as the application of \( f \) to an \( n \)-tuple \( (x_1, \ldots, x_n) \).

In case (i), \( f \) expects its arguments 'one at a time' and is said to be curried after a logician called Curried (the idea of currying was actually invented by Schönfinkel [65]). The functions and, or and add defined earlier were all curried. One advantage of curried functions is that they can be 'partially applied'; for example, \( \text{add 1} \) is the result of partially applying \( \text{add} \) to \( 1 \) and denotes the function \( n \mapsto n + 1 \).

Although it is often convenient to represent \( n \)-argument functions as curried, it is also useful to be able to represent them, as in case (ii) above, by \( \lambda \)-expressions expecting a single tuple argument. For example, instead of representing \( + \) and \( \times \) by \( \lambda \)-expressions \( \text{add} \) and \( \text{mult} \) such that

\[
\begin{align*}
\text{add } m \ n &= m + n \\
\text{mult } m \ n &= m \times n
\end{align*}
\]

it might be more convenient to represent them by functions, \( \text{sum} \) and \( \text{prod} \) say, such that

\[
\begin{align*}
\text{sum } (m, n) &= m + n \\
\text{prod } (m, n) &= m \times n
\end{align*}
\]
This is nearer to conventional mathematical usage and has applications that will be encountered later. One might say that sum and prod are uncurried versions of add and mult respectively.

Define:

\[
\text{LET curry } = \lambda f \ z_1 \ z_2 . \ f \ (z_1, z_2)
\]

\[
\text{LET uncurry } = \lambda f \ p . \ f \ (\text{fst } p) \ (\text{snd } p)
\]

then defining

\[
\begin{align*}
\text{sum} & = \text{uncurry add} = \lambda p . \ \bar{\langle \} \ \bar{\langle x \ y \rangle} \ (\text{add } p) \\
\text{prod} & = \text{uncurry mult}
\end{align*}
\]

results in sum and prod having the desired properties; for example:

\[
\begin{align*}
\text{sum } (m, n) & = \text{uncurry add } (m, n) \\
& = (\lambda f \ p . \ f \ (\text{fst } p) \ (\text{snd } p)) \text{add } (m, n) \\
& = \text{add } (\text{fst } (m, n)) \ (\text{snd } (m, n)) \\
& = \text{add } m \ n \\
& = m+n
\end{align*}
\]

Exercise 66
Show that for any \( E \):

\[
\begin{align*}
\text{curry } (\text{uncurry } E) & = E \\
\text{uncurry } (\text{curry } E) & = E
\end{align*}
\]

hence show that:

\[
\begin{align*}
\text{add} & = \text{curry sum} \\
\text{mult} & = \text{curry prod}
\end{align*}
\]

We can define \( n \)-ary functions for currying and uncurrying. For \( n > 0 \) define:

\[
\begin{align*}
\text{LET curry}_n & = \lambda f \ z_1 \ldots z_n . \ f \ (z_1, \ldots, z_n) \\
\text{LET un curry}_n & = \lambda f \ p . \ f \ (p \downarrow 1) \ldots (p \downarrow n)
\end{align*}
\]