Lester, Iain MacCullum, Ursula
Miller, Oege de Moor, Chris Oka-
rk Ramaer, Hamilton Richards,
eepak D'Souza, John Spanonda-
sato Takeichi, Peter Thiemann,
n. In particular, Jeremy Gibbons
ıscript and suggested a number

etting my LaTeX into shape, and
upport. The text was prepared
BEdit as an editor, and OzTeX as
elight to use.

Richard Bird

Chapter 1

Fundamental concepts

Programming in a functional language consists of building definitions and using the computer to evaluate expressions. The primary role of the programmer is to construct a function to solve a given problem. This function, which may involve a number of subsidiary functions, is described in a notation that obeys normal mathematical principles. The primary role of the computer is to act as an evaluator or calculator; its job is to evaluate expressions and print the results. In this respect the computer acts much like an ordinary pocket calculator. What distinguishes a functional calculator from the humbler variety is the programmer's ability to make definitions to increase its powers of calculation. Expressions that contain occurrences of the names of functions defined by the programmer are evaluated using the given definitions as simplification rules for converting expressions to printable form.

1.1 Sessions and scripts

To illustrate the idea of using a computer as a calculator, imagine we are sitting in front of a terminal screen displaying a prompt sign

```
?
```

in a window. We can now type an expression, followed by a newline character, and the computer will respond by displaying the result of evaluating the expression, followed by a new prompt on a new line, indicating that the process can begin again with another expression.

One kind of expression we might type is a number:

```
? 42
42
```

Here, the computer's response is simply to redisplay the number we typed. The decimal numeral 42 is an expression in its simplest possible form and evaluating it results in no further simplification.

We might type a slightly more interesting kind of expression:

? 6×7
42

Here, the computer can simplify the expression by performing the multiplication. In this book we will use common mathematical notations for writing expressions; in particular, the multiplication operator will be denoted by the sign $\times$, rather than the asterisk * used in Haskell.

We will not elaborate for the moment on the possible forms of numerical and other kinds of expression that can be submitted for evaluation; the important point to absorb now is that one can just type expressions and have them evaluated. This sequence of interactions between user and computer is called a *session*.

The second and intellectually more challenging aspect of functional programming consists of building definitions. A list of definitions is called a *script*. Here is an example of a simple script:

$$
\begin{array}{lcl}
\mathit{square} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{square}\ x & = & x \times x \\
\\
\mathit{smaller} & :: & (\mathit{Integer}, \mathit{Integer}) \rightarrow \mathit{Integer} \\
\mathit{smaller}\ (x, y) & = & \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y
\end{array}
$$

In this script, two functions named *square* and *smaller* have been defined. The function *square* takes an integer as argument and returns its square; the function *smaller* takes a pair of integers as argument and returns the smaller value. The syntax for making definitions follows that of Haskell, the programming language adopted in this book, and will be explained in due course. Notice, however, that definitions are written as equations between certain kinds of expression; these expressions can contain *variables*, here denoted by the symbols x and y. Furthermore, each function is accompanied by a description of its *type*; for example, $(\mathit{Integer}, \mathit{Integer}) \rightarrow \mathit{Integer}$ describes the type of functions that take a pair of integers as argument, and deliver an integer as result. Such type descriptions are also called *type assignments* or *type signatures*.

Having created a script, we can submit it to the computer and enter a session. For example, the following session is now possible:

? *square* 3768
14197824

? *square* 14198724
201578206334976

? *square* (*smaller* (5, 3 + 4))
25

Notice, in passing, that *Integer* arithmetic is exact: there is no restriction on the sizes of integers that can be computed.

The purpose of a definition is to introduce a *binding* associating a given name with a given definition. A set of bindings is called an *environment* or *context*. Expressions are always evaluated in some context and can contain occurrences of the names found in that context. The Haskell evaluator will use the definitions associated with these names as rules for simplifying expressions.

Some expressions can be evaluated without having to provide a context. In Haskell a number of operations are given as primitive in the sense that the rules of simplification are built into the evaluator. For example, the basic operations of arithmetic are provided as primitive. Other commonly useful operations are predefined in special scripts, called *preludes* or *libraries*, that can be loaded when we start the computer.

At any stage a programmer can return to the script in order to add or modify definitions. The new script can then be resubmitted to the evaluator to provide a new context and another session started. For example, suppose we return to the above script and change it to read:

$$
\begin{array}{lcl}
\mathit{square} & :: & \mathit{Float} \rightarrow \mathit{Float} \\
\mathit{square}\ x & = & x \times x \\
 & & \\
\mathit{delta} & :: & (\mathit{Float}, \mathit{Float}, \mathit{Float}) \rightarrow \mathit{Float} \\
\mathit{delta}\ (a, b, c) & = & \mathit{sqrt}\ (\mathit{square}\ b - 4 \times a \times c)
\end{array}
$$

The type assigned to *square* has been changed to *Float* → *Float*. In Haskell the type *Float* consists of the single-precision floating-point numbers. The function *delta* depends on a predefined function *sqrt* for taking square roots.

Having resubmitted the script, we can enter a new session and type, for example:

? *delta* (4.2, 7, 2.3)
3.2187

To summarise the important points made so far:

- Scripts are collections of definitions supplied by the programmer.

- Definitions are expressed as equations between certain kinds of expression and describe mathematical functions. Definitions are accompanied by type signatures.
- During a session, expressions are submitted for evaluation; these expressions can contain references to the functions defined in the script, as well as references to other functions defined in preludes or libraries.
- In Haskell, at least two different kinds of number can be used in computations: arbitrary-precision integers (elements of *Integer*), and single-precision floating-point numbers (elements of *Float*).

Exercises

1.1.1 Using the function *square*, design a function *quad* that raises its argument to the fourth power.

1.1.2 Define a function *greater* that returns the greater of its two arguments.

1.1.3 Define a function for computing the area of a circle with given radius r (use $22/7$ as an approximation to π).

1.2 Evaluation

The computer evaluates an expression by reducing it to its simplest equivalent form and displaying the result. The terms *evaluation*, *simplification*, and *reduction* will be used interchangeably to describe this process. To give a brief flavour, consider the expression *square* $(3 + 4)$; one possible sequence is

$$\begin{array}{ll} & \mathit{square}\,(3+4) \\ = & \quad \{\text{definition of } +\} \\ & \mathit{square}\,7 \\ = & \quad \{\text{definition of } \mathit{square}\} \\ & 7 \times 7 \\ = & \quad \{\text{definition of } \times\} \\ & 49 \end{array}$$

The first and third steps refer to use of the built-in rules for addition and multiplication, while the second step refers to the use of the rule defining *square* supplied by the programmer. That is to say, the definition $\mathit{square}\,x = x \times x$ is interpreted by the computer simply as a left-to-right rewrite rule for reducing

s between certain kinds of expres-
ions. Definitions are accompanied

iitted for evaluation; these expres-
ctions defined in the script, as well
d in preludes or libraries.

s of number can be used in com-
(elements of *Integer*), and single-
ents of *Float*).

inction *quad* that raises its argu-

the greater of its two arguments.

rea of a circle with given radius r

lucing it to its simplest equivalent
evaluation, *simplification*, and *re-*
cribe this process. To give a brief
4); one possible sequence is

ilt-in rules for addition and multi-
e use of the rule defining *square*
the definition $square\ x = x \times x$ is
to-right rewrite rule for reducing

expressions involving *square*. The expression '49' cannot be further reduced, so that is the result displayed by the computer. An expression is said to be *canonical*, or in *normal form*, if it cannot be further reduced. Hence '49' is in normal form.

Another reduction sequence for $square\ (3+4)$ is

$$
\begin{aligned}
& square\ (3+4) \\
= \quad & \{\text{definition of } square\} \\
& (3+4) \times (3+4) \\
= \quad & \{\text{definition of } + \text{ (applied to first term)}\} \\
& 7 \times (3+4) \\
= \quad & \{\text{definition of } +\} \\
& 7 \times 7 \\
= \quad & \{\text{definition of } \times\} \\
& 49
\end{aligned}
$$

In this reduction sequence the rule for *square* is applied first, but the final result is the same. A characteristic feature of functional programming is that if two different reduction sequences both terminate, then they lead to the same result. In other words, the meaning of an expression is its value and the task of the computer is simply to obtain it.

Let us give another example. Consider the script

$$
\begin{array}{lcl}
three & :: & Integer \rightarrow Integer \\
three\ x & = & 3 \\
 & & \\
infinity & :: & Integer \\
infinity & = & infinity + 1
\end{array}
$$

It is not clear what integer, if any, is defined by the second equation but the computer can nevertheless use the equation as a rewrite rule. Now consider simplification of *three infinity*. If we try to simplify *infinity* first, then we get the reduction sequence

$$
\begin{aligned}
& three\ infinity \\
= \quad & \{\text{definition of } infinity\} \\
& three\ (infinity + 1) \\
= \quad & \{\text{definition of } infinity\}
\end{aligned}
$$

$$\begin{aligned} & \textit{three}\ ((\textit{infinity} + 1) + 1) \\ = \quad & \{\text{and so on} \ldots\} \\ & \ldots \end{aligned}$$

This reduction sequence does not terminate. If, on the other hand, we try to simplify *three* first, then we get the sequence

$$\begin{aligned} & \textit{three infinity} \\ = \quad & \{\text{definition of } \textit{three}\} \\ & 3 \end{aligned}$$

This sequence terminates in one step. So, some ways of simplifying an expression may terminate while others do not. In Chapter 7 we will describe a reduction strategy, called *lazy evaluation*, that guarantees termination whenever termination is possible, and is also reasonably efficient. Haskell is a lazy functional language, and we will explore what consequences such a strategy has in the rest of the book. However, whichever strategy is in force, the essential point is that expressions are evaluated by a conceptually simple process of substitution and simplification, using both primitive rules and rules supplied by the programmer in the form of definitions.

Exercises

1.2.1 In order to evaluate $x \times y$, the expressions x and y are reduced to normal form and then multiplication is performed. Does evaluation of *square infinity* terminate?

1.2.2 How many terminating reduction sequences are there for the expression *square* $(3 + 4)$?

1.2.3 Imagine a language of expressions for representing integers defined by the syntax rules: (i) *zero* is an expression; (ii) if e is an expression, then so are *succ* (e) and *pred* (e). An evaluator reduces expressions in this language by applying the following rules repeatedly until no longer possible:

$$\begin{aligned} \textit{succ}\ (\textit{pred}\ (e)) &= e \\ \textit{pred}\ (\textit{succ}\ (e)) &= e \end{aligned}$$

Simplify the expression *succ* (*pred* (*succ* (*pred* (*pred* (*zero*))))).

In how many ways can the reduction rules be applied to this expression? Do they all lead to the same final result? Prove that the process of reduction must

If, on the other hand, we try to

e ways of simplifying an expres-
Chapter 7 we will describe a re-
uarantees termination whenever
efficient. Haskell is a lazy func-
equences such a strategy has in
gy is in force, the essential point
tually simple process of substi-
rules and rules supplied by the

is x and y are reduced to normal
oes evaluation of *square infinity*

ices are there for the expression

epresenting integers defined by
f e is an expression, then so are
expressions in this language by
o longer possible:

$(pred\ (zero)))))$.

applied to this expression? Do
t the process of reduction must

terminate for all given expressions. (*Hint:* Define an appropriate notion of expression size, and show that each reduction step does indeed reduce size.)

1.2.4 Carrying on from the previous question, suppose an extra syntactic rule is added to the language: (iii) if e_1 and e_2 are expressions, then so is $add\,(e_1, e_2)$. The corresponding reduction rules are

$$\begin{array}{lcl} add\,(zero, e_2) & = & e_2 \\ add\,(succ\,(e_1), e_2) & = & succ\,(add\,(e_1, e_2)) \\ add\,(pred\,(e_1), e_2) & = & pred\,(add\,(e_1, e_2) \end{array}$$

Simplify the expression $add\,(succ\,(pred\,(zero)), zero)$.

Count the number of different ways the reduction rules can be applied to the above expression. Do they always lead to the same final result?

1.2.5 Now suppose we define the size of an expression by the following rules:

$$\begin{array}{lcl} \text{size}\,(zero) & = & 1 \\ \text{size}\,(succ\,(e)) & = & 1 + \text{size}\,(e) \\ \text{size}\,(pred\,(e)) & = & 1 + \text{size}\,(e) \\ \text{size}\,(add\,(e_1, e_2)) & = & 1 + 2 \times (\text{size}\,(e_1) + \text{size}\,(e_2)) \end{array}$$

Show that application of any of the five reduction rules given above reduces expression size. Why does this prove that the process of reduction must always terminate for any given initial expression?

1.3 Values

In functional programming, as in mathematics, an expression is used solely to describe (or *denote*) a *value*. Among the kinds of value an expression may denote are included: numbers of various kinds, truth values, characters, tuples, functions, and lists. All of these will be described in due course. As we will also see, it is possible to introduce new kinds of value and define operations for generating and manipulating them.

It is important to distinguish between values and their representations by expressions. The simplest equivalent form of an expression, whatever that may be, is *not* a value but a representation of it. Somewhere, in outer space perhaps, one can imagine a universe of abstract values, but on earth they can only be recognised and manipulated by concrete representations. There may be many representations for one and the same value. For example, the abstract number forty-nine can be represented by the decimal numeral 49, the roman numeral XLIX, or the expression 7×7. Computers usually operate with the binary representation of numbers in which forty-nine is represented by a certain bit-pattern consisting of a number of 0s followed by 110001.

The evaluator for a functional language prints a value by printing its canonical representation; this representation is dependent both on the syntax given for forming expressions, and the precise definition of the reduction rules.

Some values have no canonical representations, for example function values. It is difficult to imagine a canonical representation for the function $sqrt :: Float \rightarrow Float$; one can describe this function in various ways but none of the descriptions can be regarded as canonical. Other values may have reasonable representations, but no finite ones. For example, the number π has no finite decimal representation. It is possible to get a computer to print out the decimal expansion of π digit by digit, but the process will never terminate.

For some expressions the process of reduction never stops and never produces any result. For example, the expression *infinity* defined in the previous section leads to an infinite reduction sequence. Recall that the definition was

$$\begin{array}{lcl} infinity & :: & Integer \\ infinity & = & infinity + 1 \end{array}$$

Such expressions do not denote well-defined values in the normal mathematical sense. As another example, assuming the operator / denotes numerical division, returning a number of type *Float*, the expression $1/0$ does not denote a well-defined floating-point number. A request to evaluate $1/0$ may cause the evaluator to respond with an error message, such as 'attempt to divide by zero', or go into an infinitely long sequence of calculations without producing any result.

In order that we can say that, without exception, every syntactically well-formed expression denotes a value, it is convenient to introduce a special symbol $\perp$, pronounced 'bottom', to stand for the undefined value of a particular type. In particular, the value of *infinity* is the undefined value $\perp$ of type *Integer*, and $1/0$ is the undefined value $\perp$ of type *Float*. Hence we can assert that $1/0 = \perp$.

The computer is not expected to be able to produce the value $\perp$. Confronted with an expression whose value is $\perp$, the computer may give an error message, or it may remain perpetually silent. The former situation is detectable, but the second one is not (after all, evaluation might have terminated normally the moment after the programmer decided to abort it). Thus, $\perp$ is a special kind of value, rather like the special value ∞ in mathematical calculus. Like special values in other branches of mathematics, $\perp$ can be admitted to the universe of values only if we state precisely the properties it is required to have and its relationship with other values.

It is possible, conceptually at least, to apply functions to $\perp$. For example, with the definitions *three* $x = 3$ and *square* $x = x \times x$, we have

? *three infinity*
3

? *square infinity*
{*Interrupted!*}

In the first evaluation the value of *infinity* was not needed to complete the calculation, so it was never calculated. This is a consequence of the lazy evaluation reduction strategy mentioned earlier. On the other hand, in the second evaluation the value of *infinity* is needed to complete the computation: one cannot compute $x \times x$ without knowing the value of x. Consequently, the evaluator goes into an infinite reduction sequence in an attempt to simplify *infinity* to normal form. Bored by waiting for an answer that we know will never come, we hit the the interrupt key.

If $f \perp = \perp$, then f is said to be a *strict* function; otherwise it is *nonstrict*. Thus, *square* is a strict function, while *three* is nonstrict. Lazy evaluation allows nonstrict functions to be defined, some other strategies do not.

Exercises

1.3.1 Suppose we define *multiply* by

$$\begin{array}{lcl} \mathit{multiply} & :: & (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\ \mathit{multiply}\,(x, y) & = & \textbf{if } x == 0 \textbf{ then } 0 \textbf{ else } x \times y \end{array}$$

The symbol == is used for an equality test between two integers. Assume that evaluation of $e_1 == e_2$ proceeds by reducing e_1 and e_2 to normal form and testing whether the two results are identical. Under lazy evaluation, what would be the value of *multiply* (0, *infinity*)? What would be the value of *multiply* (*infinity*, 0)?

1.3.2 Suppose we define the function h by the equation $h\,x = f\,(g\,x)$. Show that if f and g are both strict, then so is h.

1.4 Functions

Naturally enough, the most important kind of value in functional programming is a function value. Although we cannot display a function value, we can apply functions to arguments and display the results (provided, of course, that the result can be displayed). Mathematically speaking, a function f is a rule of correspondence that associates each element of a given type A with a unique element of a second type B. The type A is called the *source* type, and B the *target* type of the function. We express this information by writing $f :: A \to B$. This formula asserts that the type of f is $A \to B$. In other words, the type expression $A \to B$ denotes a type whenever A and B do, and describes the type of functions from A to B. For example, we have already met the functions

$$
\begin{array}{lcl}
\mathit{three} & :: & \mathit{Integer} \to \mathit{Integer} \\
\mathit{square} & :: & \mathit{Integer} \to \mathit{Integer} \\
\mathit{delta} & :: & (\mathit{Float}, \mathit{Float}, \mathit{Float}) \to \mathit{Float}
\end{array}
$$

The definition of *three* describes a rule of correspondence that associates every integer, including the special integer $\perp$, with the single number 3. The definition of *square* associates every well-defined integer with its square, and associates $\perp$ with the undefined integer $\perp$.

A function $f :: A \to B$ is said to take *arguments* in A and return *results* in B. If x denotes an element of A, then we write $f(x)$, or just $f\,x$, to denote the result of *applying* the function f to x. This value is the unique element of B associated with x by the rule of correspondence for f. The former notation, $f(x)$, is the one normally employed in mathematics to denote functional application, but the parentheses are not really necessary and we will use the second form, $f\,x$, instead. On the other hand, parentheses are necessary when the argument is not a simple constant or variable. For example, we have to write $\mathit{square}\,(3+4)$ (if that is what we mean) because $\mathit{square}\,3+4$ means $(\mathit{square}\,3)+4$. The reason why this is so is because application has a higher *precedence* than + (see below). Similarly, we have to write $\mathit{square}\,(\mathit{square}\,3)$ and not $\mathit{square}\;\mathit{square}\,3$.

We will be careful never to confuse a function with its application to an argument. In some mathematics texts one often finds the phrase 'the function $f(x)$', when what is really meant is 'the function f'. In such texts, functions are rarely considered as values which may themselves be used as arguments to other functions and the traditional way of speaking causes no confusion. In functional programming, however, functions are values with exactly the same status as all other values; in particular, they can be passed as arguments to other functions and returned as results. Accordingly, we cannot afford to be casual about the difference between a function and the result of applying it to an argument.

1.4.1 Extensionality

Two functions are equal if they give equal results for equal arguments. Thus, $f = g$ if and only if $f\,x = g\,x$ for all x. This principle is called the principle of *extensionality*. It says that the important thing about a function is the cor-

alled the *source* type, and B the
nformation by writing $f :: A \rightarrow B$.
$\rightarrow B$. In other words, the type
and B do, and describes the type
ve already met the functions

espondence that associates every
ie single number 3. The definition
r with its square, and associates

ents in A and return *results* in B.
), or just $f\,x$, to denote the result
ie unique element of B associated
The former notation, $f(x)$, is the
note functional application, but
we will use the second form, $f\,x$,
necessary when the argument is
e, we have to write *square* $(3+4)$
means $(square\,3)+4$. The reason
er *precedence* than $+$ (see below).
and not *square square* 3.
iction with its application to an
en finds the phrase 'the function
tion f'. In such texts, functions
iemselves be used as arguments
speaking causes no confusion. In
are values with exactly the same
can be passed as arguments to
ordingly, we cannot afford to be
n and the result of applying it to

ults for equal arguments. Thus,
principle is called the principle
hing about a function is the cor-

respondence between arguments and results, not how this correspondence is described.

For instance, we can define the function which doubles its argument in the following two ways:

$$
\begin{array}{lcl}
\mathit{double}, \mathit{double}' & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{double}\; x & = & x + x \\
\mathit{double}'\; x & = & 2 \times x
\end{array}
$$

The two definitions describe different *procedures* for obtaining the correspondence, one involving addition and the other involving multiplication, but *double* and *double′* define the same function value and we can assert *double* = *double′* as a mathematical truth. Regarded as procedures for evaluation, one definition may be more or less 'efficient' than the other, but the notion of efficiency is not one that can be attached to function values themselves. This is not to say, of course, that efficiency is not important; after all, we want expressions to be evaluated in a reasonable amount of time. The point is that efficiency is an *intensional* property of definitions, not an extensional one.

Extensionality means that we can prove $f = g$ by proving that $f\,x = g\,x$ for all x. Depending on the definitions of f and g, we may also be able to prove $f = g$ directly. The former kind of proof is called an *applicative* or *point-wise* style of proof, while the latter is called a *point-free* style. We will see examples of both styles during the course of the book.

1.4.2 Currying

A useful device for reducing the number of parentheses in an expression is the idea of replacing a structured argument by a sequence of simpler ones. To illustrate, consider again the function *smaller* defined earlier:

$$
\begin{array}{lcl}
\mathit{smaller} & :: & (\mathit{Integer}, \mathit{Integer}) \rightarrow \mathit{Integer} \\
\mathit{smaller}\; (x, y) & = & \textbf{if}\; x \leq y \;\textbf{then}\; x \;\textbf{else}\; y
\end{array}
$$

The function *smaller* takes a single argument consisting of a pair of integers, and returns an integer. Another way of defining essentially the same function is to write

$$
\begin{array}{lcl}
\mathit{smallerc} & :: & \mathit{Integer} \rightarrow (\mathit{Integer} \rightarrow \mathit{Integer}) \\
\mathit{smallerc}\; x\; y & = & \textbf{if}\; x \leq y \;\textbf{then}\; x \;\textbf{else}\; y
\end{array}
$$

The function *smallerc* takes two arguments, one after the other. More precisely, *smallerc* is a function that takes an integer x as argument and returns a function *smallerc* x; the function *smallerc* x takes an integer y as argument and returns an integer, namely the smaller of x and y.

Here is another example:

$$\begin{array}{lcl} \mathit{plus} & :: & (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\ \mathit{plus}\,(x, y) & = & x + y \end{array}$$

$$\begin{array}{lcl} \mathit{plusc} & :: & \mathit{Integer} \to (\mathit{Integer} \to \mathit{Integer}) \\ \mathit{plusc}\; x\; y & = & x + y \end{array}$$

For each integer x the function *plusc* x adds x to an integer. In particular, *plusc* 1 is the successor function that increments its argument by 1, and *plusc* 0 is the identity function on integers.

This simple device for replacing structured arguments by a sequence of simple ones is known as *currying*, after the American logician Haskell B. Curry (after whom the programming language Haskell is also named). For currying to work properly in a consistent manner, we require that the operation of functional application associates to the left in expressions. Thus,

$$\begin{array}{lll} \mathit{smallerc}\; 3\; 4 & \text{means} & (\mathit{smallerc}\; 3)\; 4 \\ \mathit{plusc}\; x\; y & \text{means} & (\mathit{plusc}\; x)\; y \\ \mathit{square}\; \mathit{square}\; 3 & \text{means} & (\mathit{square}\; \mathit{square})\; 3 \end{array}$$

Although *square square* 3 is a syntactically legal expression, it makes no sense because *square* takes a single integer argument, not a function followed by an integer. In fact, the expression will be rejected by the computer because it cannot be assigned a sensible type. We will return to this point in a later section.

There are two advantages of currying functions. Firstly, currying can help to reduce the number of parentheses that have to be written in expressions. Secondly, curried functions can be applied to one argument only, giving another function that may be useful in its own right. For instance, consider the function *twice* that applies a function twice in succession:

$$\begin{array}{lcl} \mathit{twice} & :: & (\mathit{Integer} \to \mathit{Integer}) \to (\mathit{Integer} \to \mathit{Integer}) \\ \mathit{twice}\; f\; x & = & f\; (f\; x) \end{array}$$

This is a perfectly legitimate definition in Haskell. The first argument to *twice* is a function (of type *Integer* $\to$ *Integer*), and the second argument is an integer. Applying *twice* to the first argument f, we get a function *twice* f that applies f twice. We can now define, for instance,

$$\begin{array}{lcl} \mathit{quad} & :: & \mathit{Integer} \to \mathit{Integer} \\ \mathit{quad} & = & \mathit{twice}\; \mathit{square} \end{array}$$

The function *quad* raises its argument to the fourth power. Suppose, on the other hand, that we had defined *twice* by

$$\begin{array}{lcl} \mathit{twice} & :: & (\mathit{Integer} \rightarrow \mathit{Integer}, \mathit{Integer}) \rightarrow \mathit{Integer} \\ \mathit{twice}\,(f, x) & = & f\,(f\,x) \end{array}$$

Now there is no way we can name the function that applies a function twice without also mentioning the argument to which the second function is applied. Instead of saying '*quad*, where $\mathit{quad} = \mathit{twice}\ \mathit{square}$', we would have to say '*quad*, where $\mathit{quad}\ x = \mathit{twice}\,(\mathit{square}, x)$ for all x'. The second style is clumsier.

If we want to, we can always convert an uncurried function into a curried one. The function *curry* takes an uncurried function and returns a curried version of the same function; its definition is

$$\begin{array}{lcl} \mathit{curry} & :: & ((\alpha, \beta) \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma) \\ \mathit{curry}\ f\ x\ y & = & f\,(x, y) \end{array}$$

The type signature of *curry* will be explained in Section 1.6. Note that *curry* is itself an example of a curried function: *curry* takes three arguments, one after the other. We can now refer to *curry f* as the curried version of f. For example, $\mathit{plusc} = \mathit{curry}\ \mathit{plus}$.

It is left as an exercise to define a function *uncurry* that goes the other way and converts a curried function into a noncurried one.

1.4.3 Operators

Some functions are written between their (two) arguments rather than preceding them. For example, we write

$$\begin{array}{lll} 3 + 4 & \text{rather than} & \mathit{plusc}\ 3\ 4 \\ 3 \leq 4 & \text{rather than} & \mathit{leq}\ 3\ 4 \end{array}$$

A function written using infix notation is called an *operator*. To remove ambiguity, special symbols are used to denote operators. Occasionally, we will use names rather than symbols for operators, but write them in bold font. For example, we write (15 **div** 4) and (15 **mod** 4), using names in bold font for the operators associated with integer division and remainder. The Haskell convention is to enclose the name in back quotes; for example, in Haskell one would write `(15 'div' 4)` and `(15 'mod' 4)`.

Enclosing an operator in parentheses converts it to a curried prefix function that can be applied to its arguments like any other function. For example,

$$\begin{array}{lcl} (+)\ 3\ 4 & = & 3 + 4 \\ (\leq)\ 3\ 4 & = & 3 \leq 4 \end{array}$$

In particular, *plusc* = (+). Like any other name, an operator enclosed in parentheses can be used in expressions and passed as an argument to functions. For example,

$$plus \quad = \quad uncurry\,(+)$$

introduces *plus* as another name for the uncurried version of addition.

1.4.4 Sections

The notational device of enclosing a binary operator in parentheses to convert it into a normal prefix function can be extended: an argument can also be enclosed along with the operator. If $\oplus$ denotes an arbitrary binary operator, then $(x\oplus)$ and $(\oplus x)$ are functions with the definitions

$$\begin{aligned}(x\oplus)\,y &= x \oplus y\\ (\oplus y)\,x &= x \oplus y\end{aligned}$$

These two forms are called *sections*. For example:

$(\times 2)$	is the 'doubling' function
(> 0)	is the 'positive number' test
$(1/)$	is the 'reciprocal' function
$(/2)$	is the 'halving' function
$(+1)$	is the 'successor' function

There is one exception to the rule for forming sections: $(-x)$ is interpreted as the unary operation of negation applied to the number x. Sections are not used heavily in what follows, but on occasion they provide a simple means for describing expressions conveniently and without fuss.

1.4.5 Precedence

When several operators appear together in an expression, certain rules of *precedence* are provided to resolve possible ambiguity. The precedence rules for the common arithmetic operators are absorbed in childhood without ever being stated formally. Their sole purpose in life is to allow one to reduce the number of parentheses in an expression.

In particular, exponentiation, which we will denote by $\uparrow$, takes precedence over multiplication, which in turn takes precedence over addition: for example,

$? \; 1 + 3 \uparrow 4 \times 2$
163

me, an operator enclosed in paren-
d as an argument to functions. For

:urried version of addition.

operator in parentheses to convert
tended: an argument can also be
otes an arbitrary binary operator,
lefinitions

umple:

ning sections: $(-x)$ is interpreted
to the number x. Sections are not
n they provide a simple means for
:hout fuss.

in expression, certain rules of *pre-*
nbiguity. The precedence rules for
ed in childhood without ever being
to allow one to reduce the number

vill denote by $\uparrow$, takes precedence
edence over addition: for example,

Thus, $1 + 3 \uparrow 4 \times 2 = 1 + ((3 \uparrow 4) \times 2)$. Furthermore, functional application, the operator denoted by a space, takes precedence over every other operator. For example, *square* $3 + 4$ means (*square* $3) + 4$.

1.4.6 Association

Another device for reducing parentheses is to provide an order of *association* for an operator. It is clear that when the same operator occurs twice in succession, the rule of precedence is not sufficient to resolve ambiguity. Operators can associate either to the *left* or to the *right*. We have already encountered one example of declaring such a preference: functional application associates to the left in expressions. In arithmetic, operators on the same level of precedence are usually declared to associate to the left as well. Thus $5 - 4 - 2$ means $(5 - 4) - 2$ and not $5 - (4 - 2)$. One operator that associates to the right is the function type operator $(\rightarrow)$; thus,

$$A \rightarrow B \rightarrow C \quad \text{means} \quad A \rightarrow (B \rightarrow C)$$

It is not necessary to insist that an order of association be prescribed for every operator. If no preference is indicated, then parentheses must be used to avoid ambiguity. In fact, to avoid complicating a basically simple idea, we will always use parentheses to disambiguate sequences of different operators with the same precedence.

Any declaration of a specific order of association should not be confused with a different, though related, property of operators known as *associativity*. An operator $\oplus$ is said to be associative if

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

for all values x, y, and z of the appropriate type. For example, $+$ and $\times$ are associative operators. For such operators, the choice of an order of association has no effect on meaning.

1.4.7 Functional composition

The composition of two functions f and g is denoted by $f \cdot g$ and is defined by the equation

$$\begin{array}{lcl} (\cdot) & :: & (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \\ (f \cdot g)\, x & = & f\,(g\, x) \end{array}$$

The type signature will be explained in Section 1.6. In words, $f \cdot g$ applied to x is defined to be the outcome of first applying g to x, and then applying f to the result. Not every pair of functions can be composed since the types have to match up: we require that g has type $g :: \alpha \to \beta$ for some types α and β, and that f has type $f :: \beta \to \gamma$ for some type γ. Then we obtain $f \cdot g :: \alpha \to \gamma$. For example, given $square :: Integer \to Integer$, we can define

$$\begin{array}{lcl} quad & :: & Integer \to Integer \\ quad & = & square \cdot square \end{array}$$

By the definition of composition, this gives exactly the same function $quad$ as

$$\begin{array}{lcl} quad & :: & Integer \to Integer \\ quad\ x & = & square\ (square\ x) \end{array}$$

This example illustrates the main advantage of using functional composition in programs: definitions can be written more concisely. Whether to use a point-free style or a point-wise style is partly a question of taste, and we will see functions defined in both styles in the remainder of the book. However, whatever the style of expression, it is good programming practice to construct complicated functions as the composition of simpler ones.

Functional composition is an associative operation. We have

$$(f \cdot g) \cdot h \;=\; f \cdot (g \cdot h)$$

for all functions f, g and h of the appropriate types. Accordingly, there is no need to put in parentheses when writing sequences of compositions.

Exercises

1.4.1 Suppose f and g have the following types:

$$\begin{array}{lcl} f & :: & Integer \to Integer \\ g & :: & Integer \to (Integer \to Integer) \end{array}$$

Let h be defined by

$$\begin{array}{lcl} h & :: & \cdots \\ h\ x\ y & = & f\ (g\ x\ y) \end{array}$$

Fill in the correct type assignment for h.

Now determine which, if any, of the following statements is true:

$$\begin{array}{lcl} h & = & f \cdot g \\ h\ x & = & f \cdot (g\ x) \\ h\ x\ y & = & (f \cdot g)\ x\ y \end{array}$$

1.4.2 Suppose we curry the arguments of the function *delta*, so that we can write *delta a b c* rather than *delta* (a, b, c). What is the type of the curried version?

1.4.3 In mathematics one often uses logarithms to various bases; for example, $\log_2$, $\log_e$, and $\log_{10}$. Give an appropriate type of a function *log* that takes a base and returns the logarithm function for that base.

1.4.4 Describe one appropriate type for the definite integral function of mathematical analysis, as used in the phrase 'the integral of f from a to b'.

1.4.5 Give examples of functions with the following types:

$$\begin{array}{l} (\mathit{Integer} \to \mathit{Integer}) \to \mathit{Integer} \\ (\mathit{Integer} \to \mathit{Integer}) \to (\mathit{Integer} \to \mathit{Integer}) \end{array}$$

1.4.6 Which, if any, of the following statements is true?

$$\begin{array}{lcl} (\times)\, x & = & (x\times) \\ (+)\, x & = & (x+) \\ (-)\, x & = & (-x) \end{array}$$

1.4.7 Define a function *uncurry* that converts a curried function into a noncurried version. Show that

$$\begin{array}{lcl} \mathit{curry}\ (\mathit{uncurry}\ f)\ x\ y & = & f\ x\ y \\ \mathit{uncurry}\ (\mathit{curry}\ f)\ (x, y) & = & f\ (x, y) \end{array}$$

for all x and y.

1.5 Definitions

So far, we have seen one or two simple definitions of functions, but definitions of other kinds of value are possible. For example,

$$\begin{array}{lcl} \mathit{pi} & :: & \mathit{Float} \\ \mathit{pi} & = & 3.14159 \end{array}$$

declares a single-precision approximation to π.

The definition of *smaller* seen earlier made use of a *conditional expression*. Recall that the definition was

$$\begin{array}{lcl} \mathit{smaller} & :: & (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\ \mathit{smaller}\ (x, y) & = & \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y \end{array}$$

The condition $x \leq y$ evaluates to a *boolean* or truth value, *True* or *False*. Boolean values will be considered in detail in the following chapter.

Another way to express essentially the same definition of *smaller* is to write

$$
\begin{array}{l}
\mathit{smaller} :: (\mathit{Integer}, \mathit{Integer}) \rightarrow \mathit{Integer} \\
\mathit{smaller}\ (x, y) \\
\qquad \left| \begin{array}{lcl} x \leq y & = & x \\ x > y & = & y \end{array} \right.
\end{array}
$$

This form of definition uses *guarded equations*. The syntax of guarded equations follows that of Haskell and consists of a sequence of *clauses* delimited by a vertical bar. Each clause consists of a condition, or *guard*, and an expression, which is separated from the guard by an $=$ sign. In the definition of *smaller* (x, y) the guards are $(x \leq y)$ and $(x > y)$, while the associated expressions are x and y.

We will use guarded equations only sparingly in what follows, preferring conditional expressions instead. The main advantage of guarded equations is when there are three or more clauses in a definition. To illustrate, consider the function *signum* that takes an integer argument x and returns -1 if x is negative, 0 if x is zero, and 1 if x is positive. Using guarded equations, we would write

$$
\begin{array}{l}
\mathit{signum} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{signum}\ x \\
\qquad \left| \begin{array}{lcl} x < 0 & = & -1 \\ x == 0 & = & 0 \\ x > 0 & = & 1 \end{array} \right.
\end{array}
$$

Using conditional expressions, we would have to write something like

$$
\begin{array}{lcl}
\mathit{signum} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{signum}\ x & = & \textbf{if}\ x < 0\ \textbf{then}\ -1\ \textbf{else} \\
 & & \textbf{if}\ x == 0\ \textbf{then}\ 0\ \textbf{else}\ 1
\end{array}
$$

The definition using guarded equations is clearer because the three conditions are made explicit. Note that an equality test is written in the form $x == y$. This is to distinguish it from a definition, which is written in the form $x = y$. Equality and comparison tests are considered further in the following chapter.

1.5.1 Recursive definitions

Definitions can also be *recursive*. Here is a well-known example:

$$\begin{array}{lcl} \mathit{fact} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\ \mathit{fact}\ n & = & \textbf{if}\ n == 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathit{fact}\ (n-1) \end{array}$$

The function *fact* is the factorial function. Recursive definitions are evaluated like any other definition. For example, one reduction sequence for evaluating *fact* 1 is

$$\begin{array}{ll} & \mathit{fact}\ 1 \\ = & \quad \{\text{definition of } \mathit{fact}\} \\ & \textbf{if}\ 1 == 0\ \textbf{then}\ 1\ \textbf{else}\ 1 \times \mathit{fact}\ (1-1) \\ = & \quad \{\text{since } (1 == 0) \text{ evaluates to } \mathit{False}\} \\ & 1 \times \mathit{fact}\ (1-1) \\ = & \quad \{\text{definition of } \mathit{fact}\} \\ & 1 \times (\textbf{if}\ (1-1) == 0\ \textbf{then}\ 1\ \textbf{else}\ (1-1) \times \mathit{fact}\ ((1-1)-1)) \\ = & \quad \{\text{definition of } (-)\} \\ & 1 \times (\textbf{if}\ 0 == 0\ \textbf{then}\ 1\ \textbf{else}\ (1-1) \times \mathit{fact}\ ((1-1)-1)) \\ = & \quad \{\text{since } (0 == 0) \text{ evaluates to } \mathit{True}\} \\ & 1 \times 1 \\ = & \quad \{\text{definition of } \times\} \\ & 1 \end{array}$$

There are, of course, other reduction sequences of *fact* 1, but all lead to the same result.

The above definition of *fact* is not completely satisfactory: if we apply *fact* to a negative integer, then the computation never terminates. For example,

$$\begin{array}{ll} & \mathit{fact}\ (-1) \\ = & \quad \{\text{definition of } \mathit{fact}\} \\ & \textbf{if}\ {-1} == 0\ \textbf{then}\ 1\ \textbf{else}\ (-1) \times \mathit{fact}(-1-1) \\ = & \quad \{\text{since } (-1 == 0) \text{ evaluates to } \mathit{False}\} \\ & (-1) \times \mathit{fact}\ (-1-1) \\ = & \quad \{\text{as before}\} \end{array}$$

$$(-1) \times ((-2) \times fact\ (-1 - 1 - 1))$$

and so on. Although it is the case that $fact\ x = \perp$ for negative x, we would prefer that the computation terminated with a suitable error message rather than proceeding indefinitely with a futile computation. One way of achieving this is to rewrite the definition as

$$\begin{array}{l} fact :: Integer \rightarrow Integer \\ fact\ n \\ \quad \begin{array}{|lcl} n < 0 & = & error \text{ “negative argument to fact”} \\ n == 0 & = & 1 \\ n > 0 & = & n \times fact\ (n-1) \end{array} \end{array}$$

The predefined function *error* takes a string as argument; when evaluated it causes immediate termination of the evaluator and displays the given error message:

? *fact* (−1)
Program error: negative argument to fact

There are other ways to define *fact* and we will discuss them later in the book.

1.5.2 Local definitions

The final piece of notation we will introduce here is called a *local* definition. In mathematical descriptions one often finds an expression qualified by a phrase of the form ‘where ...’. For instance, one might find ‘$f(x, y) = (a + 1)(a + 2)$, where $a = (x + y)/2$’. The same device can be used in a formal definition:

$$\begin{array}{lcl} f & :: & (Float, Float) \rightarrow Float \\ f\ (x, y) & = & (a+1) \times (a+2) \quad \textbf{where } a = (x+y)/2 \end{array}$$

The special word **where** is used to introduce a local definition whose context (or *scope*) is the expression on the right-hand side of the definition of f.

When there are two or more local definitions we can lay them out in one of two styles. For example, one can write

$$\begin{array}{lcl} f & :: & (Float, Float) \rightarrow Float \\ f\ (x, y) & = & (a+1) \times (b+2) \\ & & \quad \textbf{where } a = (x+y)/2 \\ & & \quad \phantom{\textbf{where }} b = (x+y)/3 \end{array}$$

One can also writ

f ::
f (x, y) =

In the second for
definition can b
equations. Cons

f :: Intege
f x y
x
x
wh

In this definitic

Exercises

1.5.1 The Fibo
and $f_{n+2} = f_n +$
an integer n a

1.5.2 Define a
of an integer.

1.6 Types

In functional
ised collectio
but there are
booleans (ele
on. Moreover
ite variety of
so on. In the
and how to c
Each type
for other typ
or multiply
gramming la
Moreover, t

One can also write

$$
\begin{array}{lcl}
f & :: & (\mathit{Float}, \mathit{Float}) \to \mathit{Float} \\
f\ (x, y) & = & (a + 1) \times (b + 2) \\
 & & \quad \textbf{where } a = (x + y)/2; \quad b = (x + y)/3
\end{array}
$$

In the second form, a semi-colon is used to separate the two definitions. A local definition can be used in conjunction with a definition that relies on guarded equations. Consider the following definition:

$$
\begin{array}{l}
f :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
f\ x\ y \\
\quad \begin{array}{|lcl}
x \le 10 & = & x + a \\
x > 10 & = & x - a
\end{array} \\
\quad \textbf{where } a = \mathit{square}\ (y + 1)
\end{array}
$$

In this definition, the **where** clause qualifies *both* guarded equations.

Exercises

1.5.1 The Fibonacci numbers $f_0, f_1, \ldots$ are defined by the rule that $f_0 = 0$, $f_1 = 1$ and $f_{n+2} = f_n + f_{n+1}$ for all $n \geq 0$. Give a definition of the function *fib* that takes an integer n and returns f_n.

1.5.2 Define a function $abs :: Integer \to Integer$ that returns the absolute value of an integer.

1.6 Types

In functional programming the universe of values is partitioned into organised collections, called *types*. So far, we have mentioned *Integer* and *Float*, but there are also other kinds of number, including *Int* and *Double*, as well as booleans (elements of *Bool*), characters (elements of *Char*), lists, trees, and so on. Moreover, we have already seen how to put types together to make an infinite variety of other types; for example $Integer \to Float$, and $(Float, Float)$, and so on. In the next chapter we will see how some of these types can be defined, and how to define new types.

Each type has associated with it certain operations which are not meaningful for other types. For instance, one cannot sensibly add a number to a character or multiply two functions together. It is an important principle of many programming languages that every well-formed expression can be assigned a type. Moreover, this type can be deduced from the constituents of the expression alone. In other words, just as the value of an expression depends only on the values of its component expressions, so does its type. This principle is called *strong typing*.

The major consequence of the discipline imposed by strong typing is that any expression which cannot be assigned a sensible type is regarded as not being well formed and is rejected by the computer before evaluation. Such expressions are simply regarded as illegal. We saw one example earlier: the expression *squaresquare*3 is rejected by the computer as not being well formed. Similarly, the script

$$\begin{array}{lcl} quad & :: & Integer \rightarrow Integer \\ quad\ x & = & square\ square\ x \end{array}$$

is rejected by the computer since the expression *square square x* is not well formed.

One great advantage of strong typing is that it enables a range of errors, from simple typographical mistakes to muddled definitions, to be detected before evaluation. The other great advantage is that it steers the programmer into a certain discipline of thought, namely to consider appropriate types for the values being defined before considering the definitions themselves. In other words, adherence to the discipline of strong typing can help significantly in the design of clear and well-structured programs.

There are two stages of analysis when an expression is submitted for evaluation. The expression is first checked to see whether it conforms to the correct syntax laid down for constructing expressions. If it does not, the computer signals a *syntax error*. If it does, then the expression is analysed to see if it possesses a sensible type. If the expression fails to pass this stage, the computer signals a *type error*. Only if the expression passes both stages can the process of evaluation begin. Similar remarks apply to definitions created in a script.

1.6.1 Polymorphic types

Some functions and operations work with many types. For example, suppose

$$\begin{array}{lcl} square & :: & Integer \rightarrow Integer \\ sqrt & :: & Integer \rightarrow Float \end{array}$$

Then the expressions $square \cdot square$ and $sqrt \cdot square$ are both meaningful and they have the following types:

$$\begin{array}{lcl} square \cdot square & :: & Integer \rightarrow Integer \\ sqrt \cdot square & :: & Integer \rightarrow Float \end{array}$$

expression depends only on the
its type. This principle is called

imposed by strong typing is that
sensible type is regarded as not
mputer before evaluation. Such
Ne saw one example earlier: the
mputer as not being well formed.

sion *square square x* is not well

t it enables a range of errors, from
efinitions, to be detected before
it it steers the programmer into
nsider appropriate types for the
definitions themselves. In other
ping can help significantly in the

xpression is submitted for evalu-
hether it conforms to the correct
ns. If it does not, the computer
ession is analysed to see if it pos-
to pass this stage, the computer
sses both stages can the process
definitions created in a script.

ny types. For example, suppose

· *square* are both meaningful and

However, the two uses of functional composition in these expressions have different types, namely

$$(\cdot) :: (Integer \rightarrow Integer) \rightarrow (Integer \rightarrow Integer) \rightarrow (Integer \rightarrow Integer)$$
$$(\cdot) :: (Integer \rightarrow Float) \rightarrow (Integer \rightarrow Integer) \rightarrow (Integer \rightarrow Float)$$

Thus, the operation of functional composition is assigned different types in different expressions.

The problem of assigning a single type to $(\cdot)$ is solved by introducing *type variables*. The type assigned to $(\cdot)$ is

$$(\cdot) \quad :: \quad (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$

Here, α, β, and γ denote type variables. We will use greek letters to denote type variables. Like other kinds of variable, a type variable can be instantiated to different types in different circumstances. A type containing type variables is called a *polymorphic type*.

Here is another example. Look again the previous definition of *fact*:

$$
\begin{array}{l}
fact :: Integer \rightarrow Integer \\
fact\ n \\
\quad \left|\begin{array}{lll} n < 0 & = & error \text{ “negative argument to fact”} \\ n == 0 & = & 1 \\ n > 0 & = & n \times fact\ (n-1) \end{array}\right.
\end{array}
$$

Consider the function *error*. It takes a string as argument, so its type is $String \rightarrow A$ for some type A. In the program above it is clear that $A = Integer$; only with this type assignment is the program for *fact* well formed. After all, the second and third clauses deliver integers, so the first one should do too. It doesn't matter what integer is delivered, because the sole purpose in evaluating *error* is to abort the computation with an error message. If, however, the general error function had type $error :: String \rightarrow Integer$, it would have limited usefulness. Instead, the type assigned to *error* is $String \rightarrow \alpha$. Once again, the problem is resolved by making use of type variables.

As a final example for now, consider the function *curry* defined in Section 1.4.2 by the equation

$$curry\ f\ x\ y \quad = \quad f\ (x, y)$$

This function is used to convert functions with type $(A, B) \rightarrow C$ into functions with type $A \rightarrow B \rightarrow C$. No properties of any specific types A, B, and C are required in the definition of *curry*, so it is assigned the polymorphic type

$$curry \quad :: \quad ((\alpha, \beta) \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma)$$

The rightmost pair of parentheses could have been omitted, since the type operator $(\rightarrow)$ associates to the right, but it is clearer in this case to put them in.

We now have the beginnings of a language of expressions that denote types. This language contains constant expressions, such as *Integer* or *Float*, variables, such as α and β, and functions that take types to other types, such as the function type operator $(\rightarrow)$.

1.6.2 Type classes

A careful reading of the first part of this chapter reveals that we have used (curried) multiplication with two different type signatures:

$$
\begin{aligned}
(\times) &:: \mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}\\
(\times) &:: \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float}
\end{aligned}
$$

Like $(\cdot)$ and *error*, it seems that $(\times)$ should be assigned a polymorphic type, namely

$$(\times) :: \alpha \rightarrow \alpha \rightarrow \alpha$$

But one can argue that this type is too general. For instance, we cannot sensibly multiply two characters or two booleans.

In Haskell the resolution is to group together kindred types into *type classes*. In particular, *Integer* and *Float* belong to the same class, the class of numbers. The type assigned to $(\times)$ is

$$(\times) :: \mathit{Num}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

The right-hand side should be read as the type $\alpha \rightarrow \alpha \rightarrow \alpha$ restricted to those α that are instances of the type class *Num*.

The same device is used for the numeric constants. For example, 3 can be used to describe a certain floating-point number as well as an integer; accordingly, the type assigned to 3 is $\mathit{Num}\ \alpha \Rightarrow \alpha$. In words, any type, provided it is a number type.

There are other kindred types apart from numbers. For example, there are the types whose values can be displayed, the types whose values can be compared for equality, the types whose values can be enumerated, and so on. A type that is an instance of one type class may also be an instance of another. For example, we can compare numbers for equality and we can also display them. We will explain in the following chapter how type classes can be created and how specific types can be declared to be instances of these classes.

ave been omitted, since the type
is clearer in this case to put them

e of expressions that denote types.
such as *Integer* or *Float*, variables,
types to other types, such as the

hapter reveals that we have used
pe signatures:

l be assigned a polymorphic type,

l. For instance, we cannot sensibly

ier kindred types into *type classes*.
same class, the class of numbers.

ype $\alpha \rightarrow \alpha \rightarrow \alpha$ restricted to those

constants. For example, 3 can be
nber as well as an integer; accord-
n words, any type, provided it is a

numbers. For example, there are
e types whose values can be com-
can be enumerated, and so on. A
ay also be an instance of another
equality and we can also display
er how type classes can be created
instances of these classes.

Exercises

1.6.1 Give suitable polymorphic type assignments for the following functions:

$$\begin{aligned} \mathit{const}\ x\ y &= x \\ \mathit{subst}\ f\ g\ x &= f\ x\ (g\ x) \\ \mathit{apply}\ f\ x &= f\ x \\ \mathit{flip}\ f\ x\ y &= f\ y\ x \end{aligned}$$

1.6.2 Define a function *swap* so that

$$\mathit{flip}\ (\mathit{curry}\ f) = \mathit{curry}\ (f \cdot \mathit{swap})$$

for all $f :: (\alpha, \beta) \rightarrow \gamma$.

1.6.3 Can you find polymorphic type assignments for the following functions?

$$\begin{aligned} \mathit{strange}\ f\ g &= g\ (f\ g) \\ \mathit{stranger}\ f &= f\ f \end{aligned}$$

1.6.4 Find a polymorphic type assignment for

$$\mathit{square}\ x = x \times x$$

1.7 Specifications

In computing, a *specification* is a description of what task a program is to perform, while an *implementation* is a program that satisfies the specification. Specifications and implementations serve different purposes: specifications are expressions of the programmer's intent (or client's expectations) and their purpose is to be clear as possible; implementations are expressions for execution by computer and their purpose is to be efficient enough to execute within the time or space available. The link between the two is the requirement that the implementations satisfies, or *meets*, the specification, and the programmer may be obliged to provide a *proof* that this is indeed the case.

A specification for a function is some statement of the intended relationship between arguments and results. A simple example is given by the following specification of a function *increase*:

$$\begin{aligned} \mathit{increase} &:: \mathit{Integer} \rightarrow \mathit{Integer} \\ \mathit{increase}\ x &> \mathit{square}\ x, \quad \text{whenever } x \geq 0 \end{aligned}$$

This specification says that the result of *increase* should be strictly greater than the square of its argument, whenever the argument is nonnegative. The specification does not say how *increase* should be computed, but gives only a property that any implementation should have. The specification is *not* part of our programming language, even though it is expressed in a similar style.

One possible implementation is to take $\mathit{increase}\ x = \mathit{square}\ (x + 1)$. The proof that this definition satisfies its specification is as follows:

$$
\begin{array}{ll}
 & \mathit{increase}\ x \\
= & \quad \{\text{definition of } \mathit{increase}\} \\
 & \mathit{square}\ (x + 1) \\
= & \quad \{\text{definition of } \mathit{square}\} \\
 & (x + 1) \times (x + 1) \\
= & \quad \{\text{algebra}\} \\
 & x \times x + 2 \times x + 1 \\
> & \quad \{\text{since } x \geq 0 \text{ implies } 2 \times x + 1 > 0\} \\
 & x \times x \\
= & \quad \{\text{definition of } \mathit{square}\} \\
 & \mathit{square}\ x
\end{array}
$$

The proof format used above will be followed in the rest of the book. Indeed, we have used it already in the discussion of reduction sequences. A reduction sequence is also a kind of proof, albeit one conducted with a very restricted set of reasoning rules. So restricted, in fact, that a computer can be instructed to carry out all the steps in a purely mechanical fashion.

Above, we invented a definition of *increase* first, and then verified that it met its specification. There are many other functions that will satisfy the specification and, since that is the only requirement, all are equally good.

One way of specifying a function is to state the rule of correspondence explicitly. The notation of functional programming can be very expressive, and sometimes the most sensible specification of a function is a legitimate program. The specification can then be executed directly. However, it may prove to be so grossly inefficient that the possibility of execution will be mostly of theoretical interest. Having written an executable specification, the programmer is not necessarily relieved of the burden (or pleasure) of producing an equivalent but acceptably efficient alternative.

As one very simple example of the idea, consider the specification

$$
\begin{array}{lll}
\mathit{quad} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{quad}\ x & = & x \times x \times x \times x
\end{array}
$$

uld be computed, but gives only a
ve. The specification is *not* part of
s expressed in a similar style.
increase x = *square* $(x + 1)$. The
cation is as follows:

$\cdot$ 0}

ed in the rest of the book. Indeed,
reduction sequences. A reduction
onducted with a very restricted set
at a computer can be instructed to
al fashion.
ase first, and then verified that it
functions that will satisfy the spe-
ment, all are equally good.
ate the rule of correspondence ex-
nming can be very expressive, and
f a function is a legitimate program.
ctly. However, it may prove to be
execution will be mostly of theor-
e specification, the programmer is
easure) of producing an equivalent

consider the specification

In search of better things, we may calculate:

$$
\begin{array}{ll}
& \mathit{quad}\ x \\
= & \quad \{\text{specification}\} \\
& x \times x \times x \times x \\
= & \quad \{\text{since } \times \text{ is associative}\} \\
& (x \times x) \times (x \times x) \\
= & \quad \{\text{definition of } \mathit{square}\} \\
& \mathit{square}\ x \times \mathit{square}\ x \\
= & \quad \{\text{definition of } \mathit{square}\} \\
& \mathit{square}\ (\mathit{square}\ x)
\end{array}
$$

The result is that we can implement *quad* with two multiplications instead of three, a significant saving with arbitrary-precision arithmetic.

In this case, we didn't invent the implementation of *quad* first, but developed it from the specification. The derivation was not entirely mechanical: the creative step was to employ the associativity of multiplication to put in brackets in a clever way. Admittedly, this example is absurdly simple, but we will see other, more convincing, examples of systematic program development, or *program synthesis*, later on in the book.

This paradigm of software development – first write a clear specification, then develop an acceptably efficient implementation – has been the focus of active research over the past twenty years, and should not be taken as a cut-and-dried method applicable in all circumstances. Two potential sources of difficulty are that the formal specification may not match the client's informal requirements, and the proof that the implementation meets the specification may be so large that it cannot be guaranteed to be free of error. Nevertheless, by trying to follow the approach whenever we can, the reliability of programs can be greatly increased.

Exercises

1.7.1 Give another definition of *increase* that meets its specification.

1.8 Chapter notes

The interactive use of a functional language, as described in the text, is provided by the HUGS (Haskell Users Gofer System) environment developed by Mark

Jones of Nottingham University. HUGS is available by FTP from

```
ftp://ftp.cs.nott.ac.uk/haskell/hugs
```

Haskell proper is a non-interactive language. Haskell compilers are available from Chalmers, Glasgow, and Yale Universities, by FTP from

```
ftp://ftp.cs.chalmers.se/pub/haskell
ftp://ftp.dcs.glasgow.ac.uk/pub/haskell
ftp://nebula.cs.yale.edu/pub/haskell
```

The language used in this book follows Haskell 1.3, although not all features of Haskell 1.3 will be covered. Furthermore, normal mathematical symbols are preferred over Haskell ones, which use a restricted character set. For example, Haskell uses * for multiplication. The keywords **if**, **then**, **else**, and **where** are reserved words in Haskell.

Web pages for Haskell, which include an on-line version of the Haskell 1.3 report, extensions to Haskell, and information about Haskell implementations, can be found at the following site:

```
http://www.haskell.org/
```

A tutorial introduction to Haskell is given in Hudak, Fasel, and Peterson (1996). Another elementary text on lazy functional programming that uses Haskell 1.3 is Thompson (1996).

While this text was being prepared there has been another release, Haskell 1.4. Currently the Haskell committee are aiming to move towards a standardisation, Standard Haskell, of the language. None of the changes under discussion is likely to affect the details described in the text.

Other nonstrict functional languages include Gofer and Miranda (Miranda is a trade-mark of Research Software Ltd.). Miranda, which is fairly close to Haskell, is described in Thompson (1995) and Clack, Myers, and Poon (1995). Another popular functional language is ML, which differs from Haskell in that it is strict rather than lazy. ML is described in Paulson (1996).

For further information about the denotational aspects of programming languages, consult Stoy (1977) or Gordon (1979). The implementation of lazy functional languages is covered in Peyton Jones (1987) and Peyton Jones and Lester (1991). The formal derivation of programs from their specifications is the subject of Morgan (1996) and Kaldewaij (1990), although the target programming language is procedural, not functional. For a functional and relational treatment of program derivation in a categorical setting, consult Bird and de Moor (1997). This is an advanced text, suitable for those particularly interested in the mathematics of programming, and can be studied after the present one.