chapter 12

Lazy evaluation

In this chapter we introduce lazy evaluation, the mechanism used to evaluate expressions in Haskell. We start by reviewing the notion of evaluation, then consider evaluation strategies and their properties, discuss infinite structures and modular programming, and conclude with a special form of function application that can improve the space performance of programs.

12.1 Introduction

As we have seen throughout this book, the basic method of computation in Haskell is the application of functions to arguments. For example, suppose that we define a function that increments an integer:

```
inc     :: Int → Int
inc n    = n + 1
```

Then the expression *inc* $(2 * 3)$ can be evaluated as follows:

```
     inc (2 * 3)
=        { applying * }
     inc 6
=        { applying inc }
     6 + 1
=        { applying + }
     7
```

Alternatively, the same result can also be obtained by performing the first two function applications in the opposite order:

```
     inc (2 * 3)
=        { applying inc }
     (2 * 3) + 1
=        { applying * }
     6 + 1
=        { applying + }
```

$$7$$

The fact that changing the order in which functions are applied does not affect the final result is not specific to such simple examples, but is an important general property of function application in Haskell. More formally, in Haskell any two different ways of evaluating the same expression will always produce the same final value, provided that they both terminate. We will return to the issue of termination later on in this chapter.

Note that the above property does not hold for most imperative programming languages, in which the basic method of computation is changing stored values. For example, consider the imperative expression $n + (n := 1)$ that adds the current value of the variable n to the result of changing its value to one, assuming that n initially has the value zero. This expression can be evaluated by first performing the left-hand side of the addition

$$\begin{array}{ll} & n + (n := 1) \\ = & \quad \{ \text{applying } n \} \\ & 0 + (n := 1) \\ = & \quad \{ \text{applying} := \} \\ & 0 + 1 \\ = & \quad \{ \text{applying} + \} \\ & 1 \end{array}$$

or alternatively, by first performing the right-hand side:

$$\begin{array}{ll} & n + (n := 1) \\ = & \quad \{ \text{applying} := \} \\ & n + 1 \\ = & \quad \{ \text{applying } n \} \\ & 1 + 1 \\ = & \quad \{ \text{applying} + \} \\ & 2 \end{array}$$

The final value is different in each case. The general problem illustrated by this example is that the precise time at which an assignment is performed in an imperative language may affect the value that results from a computation. In contrast, the time at which a function is applied to an argument in Haskell never affects the value that results from a computation. Nonetheless, as we shall see in the remainder of this chapter, there are important practical issues concerning the order and nature of evaluation.

12.2 Evaluation strategies

An expression that has the form of a function applied to one or more arguments that can be "reduced" by performing the application is called a reducible expression, or *redex* for short. As indicated by the use of quotations marks in the preceding sentence, such reductions do not necessarily decrease the size of an expression, although in practice this is often the case.

By way of example, suppose that we define a function *mult* that takes a pair of integers and returns their product:

```
mult         :: (Int, Int) → Int
mult (x, y)  =  x * y
```

Now consider the expression *mult* $(1 + 2, 2 + 3)$. This expression contains three redexes, namely the sub-expressions $1 + 2$ and $2 + 3$, which have the form of the function $+$ applied to two arguments, and the entire expression *mult* $(1 + 2, 2 + 3)$ itself, which has the form of the function *mult* applied to a pair of arguments. Performing the corresponding reductions gives the expressions *mult* $(3, 2 + 3)$, *mult* $(1 + 2, 5)$, and $(1 + 2) * (2 + 3)$.

When evaluating an expression, in what order should reductions be performed? One common strategy, called *innermost* evaluation, is to always choose a redex that is innermost, in the sense that it contains no other redex. If there is more than one innermost redex, by convention we choose that which begins at the leftmost position in the expression.

For example, both $1 + 2$ and $2 + 3$ contain no other redexes and are hence innermost within the expression *mult* $(1 + 2, 2 + 3)$, with the redex $1 + 2$ beginning at the leftmost position. More generally, our example expression is evaluated using innermost evaluation as follows:

```
   mult (1 + 2, 2 + 3)
=      { applying the first + }
   mult (3, 2 + 3)
=      { applying + }
   mult (3, 5)
=      { applying mult }
   3 * 5
=      { applying * }
   15
```

Innermost evaluation can also be characterised in terms of how arguments are passed to functions. In particular, using this strategy ensures that the argument of a function is always fully evaluated before the function itself is applied. That is, arguments are passed *by value*. For example, as shown above, evaluating *mult* $(1 + 2, 2 + 3)$ using innermost evaluation proceeds by first evaluating the argument expressions $1 + 2$ and $2 + 3$, and then applying the function *mult*. The fact that we always choose the leftmost innermost redex ensures that the first argument is evaluated before the second.

Another common strategy for evaluating an expression, dual to innermost evaluation, is to always choose a redex that is outermost, in the sense that it is contained in no other redex. If there is more than one such redex then as previously we choose that which begins at the leftmost position. Not surprisingly, this evaluation strategy is called *outermost* evaluation.

For example, the expression *mult* $(1 + 2, 2 + 3)$ is contained in no other redex and is hence outermost within itself. More generally, evaluating this expression using outermost evaluation proceeds as follows:

```
   mult (1 + 2, 2 + 3)
=      { applying mult }
   (1 + 2) * (2 + 3)
=      { applying the first + }
```

$$
\begin{array}{ll}
 & 3 * (2 + 3) \\
= & \quad \{ \text{applying } + \} \\
 & 3 * 5 \\
= & \quad \{ \text{applying } * \} \\
 & 15
\end{array}
$$

In terms of how arguments are passed to functions, using outermost evaluation allows functions to be applied before their arguments are evaluated. For this reason, we say that arguments are passed *by name*. For example, as shown above, evaluating $mult\ (1 + 2, 2 + 3)$ using outermost evaluation proceeds by first applying the function $mult$ to the two unevaluated arguments $1 + 2$ and $2 + 3$, and then evaluating these two expressions in turn.

Note, however, that many built-in functions require their arguments to be evaluated before being applied, even when using outermost evaluation. For example, as illustrated in the calculation above, built-in arithmetic operators such as $*$ and $+$ cannot be applied until their two arguments have been evaluated to numbers. Functions with this property are called strict, and will be discussed in further detail at the end of this chapter.

Lambda expressions

Let us now define a curried version of $mult$ that takes its arguments one at a time, using a lambda expression to make the use of currying explicit:

$$
\begin{array}{lll}
mult & :: & Int \to Int \to Int \\
mult\ x & = & \lambda y \to x * y
\end{array}
$$

Then using innermost evaluation, for example, we have:

$$
\begin{array}{ll}
 & mult\ (1 + 2)\ (2 + 3) \\
= & \quad \{ \text{applying the first } + \} \\
 & mult\ 3\ (2 + 3) \\
= & \quad \{ \text{applying } mult \} \\
 & (\lambda y \to 3 * y)\ (2 + 3) \\
= & \quad \{ \text{applying } + \} \\
 & (\lambda y \to 3 * y)\ 5 \\
= & \quad \{ \text{applying } \lambda y \to 3 * y \} \\
 & 3 * 5 \\
= & \quad \{ \text{applying } * \} \\
 & 15
\end{array}
$$

That is, the two arguments are now substituted into the body of the function $mult$ one at a time, as we would expect using currying, rather than at the same time as in the previous section. This behaviour arises because $mult\ 3$ is the leftmost innermost redex in the expression $mult\ 3\ (2 + 3)$, as opposed to $2 + 3$ in the expression $mult\ (3, 2 + 3)$. Performing a reduction on $mult\ 3$ in the second step of the calculation above gives the lambda expression $(\lambda y \to 3 * y)$, which awaits the result of evaluating the second argument.

Note that in Haskell, the selection of redexes within lambda expressions is prohibited. The rational for not "reducing under lambdas" is that functions are viewed as black boxes that we are not permitted to look inside. More formally, the only operation that can be performed on a function is that of applying it to an argument. As such, reduction within the body of a function is only permitted once the function has been applied. For example, the lambda expression $\lambda x \rightarrow 1 + 2$ is deemed to be fully evaluated, even though its body contains the redex $1 + 2$, but once this function has been applied to an argument, evaluation of this redex can then proceed:

$$\begin{array}{ll}
 & (\lambda x \rightarrow 1 + 2)\ 0 \\
= & \quad \{\text{applying } \lambda x \rightarrow 1 + 2\} \\
 & 1 + 2 \\
= & \quad \{\text{applying } +\} \\
 & 3
\end{array}$$

Using innermost and outermost evaluation, but not under lambdas, is normally referred to as *call-by-value* and *call-by-name* evaluation, respectively. In the next two sections we explore how these two evaluation strategies compare in terms of two important properties, namely their termination behaviour and the number of reduction steps that they require.

12.3 Termination

Consider the following recursive definition:

$$\begin{array}{lcl}
\mathit{inf} & :: & \mathit{Int} \\
\mathit{inf} & = & 1 + \mathit{inf}
\end{array}$$

That is, the integer *inf* (abbreviating "infinity") is defined as the successor of itself. Evaluating *inf* produces a larger and larger expression, regardless of the evaluation strategy, and hence does not terminate:

$$\begin{array}{ll}
 & \mathit{inf} \\
= & \quad \{\text{applying } \mathit{inf}\} \\
 & 1 + \mathit{inf} \\
= & \quad \{\text{applying } \mathit{inf}\} \\
 & 1 + (1 + \mathit{inf}) \\
= & \quad \{\text{applying } \mathit{inf}\} \\
 & 1 + (1 + (1 + \mathit{inf})) \\
= & \quad \{\text{applying } \mathit{inf}\} \\
 & \vdots
\end{array}$$

In practice, evaluating *inf* using Hugs will quickly exhaust the available memory and produce an error message. Now consider the expression *fst* (0, *inf*), where *fst* is the library function that selects the first component of a pair, defined by $\mathit{fst}\ (x, y) = x$. Using call-by-value evaluation with this expression also results in non-termination in a similar manner:

$$\begin{array}{ll}
 & \mathit{fst}\ (0, \mathit{inf}) \\
= & \quad \{\text{applying } \mathit{inf}\} \\
 & \mathit{fst}\ (0, 1 + \mathit{inf})
\end{array}$$

$$
\begin{array}{cl}
= & \{\text{ applying } \mathit{inf} \,\} \\
 & \mathit{fst}\ (0, 1 + (1 + \mathit{inf})) \\
= & \{\text{ applying } \mathit{inf} \,\} \\
 & \mathit{fst}\ (0, 1 + (1 + (1 + \mathit{inf}))) \\
= & \{\text{ applying } \mathit{inf} \,\} \\
 & \vdots
\end{array}
$$

In contrast, using call-by-name evaluation results in termination in just one step, by immediately applying the definition of *fst* and hence avoiding the evaluation of the non-terminating expression *inf*:

$$
\begin{array}{cl}
 & \mathit{fst}\ (0, \mathit{inf}) \\
= & \{\text{ applying } \mathit{fst} \,\} \\
 & 0
\end{array}
$$

This simple example shows that call-by-name evaluation may produce a result when call-by-value evaluation fails to terminate. More generally, we have the following important property: if there exists any evaluation sequence that terminates for a given expression, then call-by-name evaluation will also terminate for this expression, and produce the same final result.

In summary, call-by-name evaluation is preferable to call-by-value for the purpose of ensuring that evaluation terminates as often as possible.

12.4 Number of reductions

Now consider the following definition:

$$
\begin{array}{lcl}
\mathit{square} & :: & \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{square}\ n & = & n * n
\end{array}
$$

For example, using call-by-value evaluation, we have:

$$
\begin{array}{cl}
 & \mathit{square}\ (1 + 2) \\
= & \{\text{ applying } + \,\} \\
 & \mathit{square}\ 3 \\
= & \{\text{ applying } \mathit{square} \,\} \\
 & 3 * 3 \\
= & \{\text{ applying } * \,\} \\
 & 9
\end{array}
$$

In contrast, using call-by-name evaluation with the same expression requires one extra reduction step, due to the fact that $1 + 2$ is duplicated when the function *square* is applied, and hence must be evaluated twice:

$$
\begin{array}{cl}
 & \mathit{square}\ (1 + 2) \\
= & \{\text{ applying } \mathit{square} \,\} \\
 & (1 + 2) * (1 + 2) \\
= & \{\text{ applying the first } + \,\} \\
 & 3 * (1 + 2) \\
= & \{\text{ applying } + \,\}
\end{array}
$$

$$
\begin{aligned}
& 3 * 3 \\
= & \quad \{ \text{applying } * \} \\
& 9
\end{aligned}
$$

This example shows that call-by-name evaluation may require more steps than call-by-value evaluation, in particular when an argument is used more than once in the body of a function. More generally, we have the following property: arguments are evaluated precisely once using call-by-value evaluation, but may be evaluated many times using call-by-name.

Fortunately, the above efficiency problem with call-by-name evaluation can easily be solved, by using pointers to indicate sharing of expressions during evaluation. Rather than physically copying an argument if it is used many times in the body of a function, we simply keep one copy of the argument and make many pointers to it. In this manner, any reductions that are performed on the argument are automatically shared between each of the pointers to that argument. For example, using this strategy we have:

$$
\begin{aligned}
& \mathit{square}\ (1 + 2) \\
= & \quad \{ \text{applying } \mathit{square} \}
\end{aligned}
$$

$$\begin{tikzcd}[column sep=small] \bullet \arrow[rrr, bend left] & * & \bullet \arrow[r, bend left] & 1+2 \end{tikzcd}$$

$$
= \quad \{ \text{applying } + \}
$$

$$\begin{tikzcd}[column sep=small] \bullet \arrow[rrr, bend left] & * & \bullet \arrow[r, bend left] & 3 \end{tikzcd}$$

$$
\begin{aligned}
= & \quad \{ \text{applying } * \} \\
& 9
\end{aligned}
$$

That is, when applying the definition $\mathit{square}\ n = n * n$ in the first step, we keep a single copy of the argument expression $1 + 2$, and make two pointers to it. In this manner, when the expression $1 + 2$ is reduced in the second step, both pointers in the expression share the result.

The use of call-by-name evaluation in conjunction with sharing is called *lazy evaluation*. This is the evaluation strategy that is used in Haskell, as a result of which Haskell is called a lazy programming language. Being based upon call-by-name evaluation, lazy evaluation has the property that it ensures that evaluation terminates as often as possible. Moreover, using sharing ensures that lazy evaluation never requires more steps than call-by-value evaluation. The use of the term "lazy" will be explained shortly.

12.5 Infinite structures

An additional property of call-by-name evaluation, and hence lazy evaluation, is that it allows what at first sight may seem impossible: programming with infinite structures. We have already seen a simple example of this idea earlier in this chapter, in the form of the evaluation of $\mathit{fst}\ (0, \mathit{inf})$ avoiding the production of the infinite structure $1 + (1 + (1 + \cdots))$ defined by inf.

More interesting forms of behaviour occur when we consider infinite lists. For example, consider the following recursive definition:

$$\begin{aligned} ones &:: [Int] \\ ones &= 1 : ones \end{aligned}$$

That is, the list *ones* is defined as a single one followed by itself. As with *inf*, evaluating *ones* does not terminate, regardless of the strategy used:

$$\begin{array}{ll} & ones \\ = & \quad \{ \text{applying } ones \} \\ & 1 : ones \\ = & \quad \{ \text{applying } ones \} \\ & 1 : (1 : ones) \\ = & \quad \{ \text{applying } ones \} \\ & 1 : (1 : (1 : ones)) \\ = & \quad \{ \text{applying } ones \} \\ & \vdots \end{array}$$

In practice, evaluating *ones* using Hugs will produce a never-ending list of ones, until the user eventually decides to terminate this process:

```
> ones
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ···
```

Now consider the expression *head ones*, which attempts to select the first element of this infinite list of ones. Using call-by-value evaluation with this expression also results in non-termination:

$$\begin{array}{ll} & head\ ones \\ = & \quad \{ \text{applying } ones \} \\ & head\ (1 : ones) \\ = & \quad \{ \text{applying } ones \} \\ & head\ (1 : (1 : ones)) \\ = & \quad \{ \text{applying } ones \} \\ & head\ (1 : (1 : (1 : ones))) \\ = & \quad \{ \text{applying } ones \} \\ & \vdots \end{array}$$

In contrast, using lazy evaluation (or call-by-name evaluation, as sharing is not required in this example) results in termination in two steps:

$$\begin{array}{ll} & head\ ones \\ = & \quad \{ \text{applying } ones \} \\ & head\ (1 : ones) \\ = & \quad \{ \text{applying } head \} \\ & 1 \end{array}$$

This behaviour arises because lazy evaluation proceeds in a lazy manner as its name suggests, only evaluating arguments as and when strictly necessary to produce results. For example, when selecting the first element of a list, the remainder of the list is not required, and hence in *head* (1 : *ones*) the further evaluation of the infinite list *ones* is avoided. More generally, we have the following property: using lazy evaluation, expressions are only evaluated as much as required by the context in which they are used.

Using this idea, we now see that under lazy evaluation *ones* is not an infinite list as such, but rather a potentially infinite list, which is only evaluated as much as required by the context. This idea is not restricted to lists, but applies equally to any form of data structure in Haskell. For example, the last exercise for this chapter involves potentially infinite trees.

12.6 Modular programming

Lazy evaluation also allows us to separate control from data in our computations. For example, a list of three ones can be produced by selecting the first three elements (control) of the infinite list of ones (data):

```
> take 3 ones
[1, 1, 1]
```

Using the definition of *take* from the standard prelude

```
take 0 _              =  []
take (n + 1) []       =  []
take (n + 1) (x : xs) =  x : take n xs
```

this behaviour arises using lazy evaluation as follows:

```
     take 3 ones
=        { applying ones }
     take 3 (1 : ones)
=        { applying take }
     1 : take 2 ones
=        { applying ones }
     1 : take 2 (1 : ones)
=        { applying take }
     1 : 1 : take 1 ones
=        { applying ones }
     1 : 1 : take 1 (1 : ones)
=        { applying take }
     1 : 1 : 1 : take 0 ones
=        { applying take }
     1 : 1 : 1 : []
=        { list notation }
     [1, 1, 1]
```

That is, the data is only evaluated as much as required by the control, and these two parts take it in turn to perform reductions. Without lazy evaluation, the control and data parts would need to be combined in the form of a single function that produces a list of n identical elements, such as:

$$\begin{array}{lcl}
\mathit{replicate} & :: & \mathit{Int} \to a \to [a] \\
\mathit{replicate}\ 0\ \_ & = & [\,] \\
\mathit{replicate}\ (n+1)\ x & = & x : \mathit{replicate}\ n\ x
\end{array}$$

Being able to modularise programs by separating them into logically distinct parts is an important goal in programming, and being able to separate control from data is one of the most important benefits of lazy evaluation.

Note that care is still required when programming with infinite lists, to avoid non-termination. For example, the expression

$$\mathit{filter}\ (\leq 5)\ [1\,..]$$

(where $[n\,..]$ produces the infinite list of integers beginning with n) will produce the integers 1, 2, 3, 4, 5 and then loop forever, because the function *filter* (≤ 5) keeps testing elements of the infinite list in a vain attempt to find another that is less than or equal to five. In contrast, the expression

$$\mathit{takeWhile}\ (\leq 5)\ [1\,..]$$

will produce the same integers and then terminate, because $\mathit{takeWhile}\ (\leq 5)$ stops as soon as it finds an element of the list that is greater than five.

We conclude this section with an example concerning prime numbers. In chapter 5 we wrote a function to generate prime numbers up to a given limit. Here is a simple procedure for generating the infinite sequence of all prime numbers, as opposed to some finite prefix of this sequence:

- write down the infinite sequence $2, 3, 4, 5, 6, \cdots$;
- mark the first number, p, in the sequence as prime;
- delete all multiples of p from the sequence;
- return to the second step.

Note that the first and third steps require an infinite amount of work, and hence in practice the steps must be interleaved. The first few iterations of this procedure can be illustrated as follows:

$$\begin{array}{cccccccccccccccc}
\underline{\mathbf{2}} & 3 & \underline{4} & 5 & \underline{6} & 7 & \underline{8} & 9 & \underline{10} & 11 & \underline{12} & 13 & \underline{14} & 15 & \cdots \\
 & \underline{\mathbf{3}} & & 5 & \_ & 7 & & \underline{9} & & 11 & \_ & 13 & & \underline{15} & \cdots \\
 & & & \underline{\mathbf{5}} & & 7 & & & \_ & 11 & & 13 & & \_ & \cdots \\
 & & & & & \underline{\mathbf{7}} & & & & 11 & & 13 & \_ & & \cdots \\
 & & & & & & & & & \underline{\mathbf{11}} & & 13 & & & \cdots \\
 & & & & & & & & & & & \underline{\mathbf{13}} & & & \cdots
\end{array}$$

Each row corresponds to one iteration, with the first row being the initial sequence (step one), the first number in each row being written in bold to indicate its primality (step two), and all multiples of this number being underlined to indicate their deletion (step three) prior to the next iteration. In this manner, we can imagine the initial sequence of numbers falling downwards, with certain numbers being sieved out at each stage by the underlining, and the bold numbers forming the infinite sequence of primes:

$$\mathbf{2, 3, 5, 7, 11, 13, \cdots}$$

The above procedure for generating prime numbers is known as the *sieve of Eratosthenes*, after the Greek mathematician who first described it. This procedure can be translated directly into Haskell:

$$\begin{array}{lcl} primes & :: & [Int] \\ primes & = & sieve\ [2\,..] \\ sieve & :: & [Int] \to [Int] \\ sieve\ (p : xs) & = & p : sieve\ [x \mid x \leftarrow xs, x\ \text{`}mod\text{`}\ p \neq 0] \end{array}$$

That is, starting with the infinite list $[2\,..]$ (step one), we apply the function $sieve$ that retains the first number p as being prime (step two), and then calls itself recursively with a new list obtained by filtering all multiples of p from this list (steps three and four). Lazy evaluation ensures that this program does indeed produce the infinite list of all prime numbers:

```
> primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, ···
```

By freeing the generation of prime numbers from the constraint of finiteness, we have obtained a modular program on which different control parts can be used in different situations. For example, the first ten prime numbers, and the prime numbers less than ten, can be produced as follows:

```
> take 10 primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

> takeWhile (<10) primes
[2, 3, 5, 7]
```

12.7 Strict application

Haskell uses lazy evaluation by default, but also provides a special *strict* version of function application, written as \$!, which can sometimes be useful. Informally, an expression of the form f \$! x behaves in the same way as the normal application $f\ x$, except that the top-level of evaluation of the argument expression x is forced before the function f is applied.

For example, if the argument has a basic type, such as *Int* or *Bool*, then top-level evaluation is simply complete evaluation. On the other hand, for a pair type such as (*Int*, *Bool*), evaluation is performed until a pair of expressions is obtained, but no further. Similarly, for a list type, evaluation is performed until the empty list or the cons of two expressions is obtained.

More formally, an expression of the form f \$! x is only a redex once evaluation of the argument x, using lazy evaluation as normal, has reached the point where it is known that the result is not an undefined value, at which point the expression can be reduced to the normal application $f\ x$. For example, using the definition $square\ n = n * n$, evaluation of $square$ \$! $(1 + 2)$ proceeds in a