Chapter 7

Efficiency

The question of efficiency has been an ever-present undercurrent in recent discussions, and the time has come to bring this important subject to the surface. The best way to achieve efficiency is, of course, to find a decent algorithm for the problem. That leads us into the larger topic of Algorithm Design, which is not the primary focus of this book. Nevertheless we will touch on some fundamental ideas later on. In the present chapter we concentrate on a more basic question: functional programming allows us to construct elegant expressions and definitions, but do we know what it costs to evaluate them? Alan Perlis, a US computer scientist, once inverted Oscar Wilde's definition of a cynic to assert that a functional programmer was someone who knew the value of everything and the cost of nothing.

7.1 Lazy evaluation

We said in Chapter 2 that, under lazy evaluation, an expression such as

```
sqr (sqr (3+4))
```

where `sqr x = x*x`, is reduced to its simplest possible form by applying reduction steps from the outside in. That means the definition of the function `sqr` is installed first, and its argument is evaluated only when needed. The following evaluation sequence follows this prescription, but is *not* lazy evaluation:

```
  sqr (sqr (3+4))
= sqr (3+4) * sqr (3+4)
= ((3+4)*(3+4)) * ((3+4)*(3+4))
= ...
= 2401
```

The ellipsis in the penultimate line hides no fewer than four evaluations of `3+4` and two of `7*7`. Clearly the simple policy of substituting argument expressions into function expressions is a very inefficient way of carrying out reduction.

Instead, lazy evaluation guarantees that when the value of an argument is needed, it is evaluated *only once*. Under lazy evaluation, the reduction sequence would unfold basically as follows:

```
    sqr (sqr (3+4))
= let x = sqr (3+4) in x*x
= let y = 3+4 in
  let x = y*y in x*x
= let y = 7 in
  let x = y*y in x*x
= let x = 49 in x*x
= 2401
```

The expression `3+4` is evaluated only once (and so is `7*7`). The names `x` and `y` have been *bound* to expressions using `let`, though in the implementation of Haskell these names are anonymous *pointers* to expressions. When an expression is reduced to a value, the pointer then points to the value and that value can then be *shared*.

Even then, the headline 'Under lazy evaluation arguments are evaluated only when needed and then only once!' doesn't tell the full story. Consider evaluation of `sqr (head xs)`. In order to evaluate `sqr` we have to evaluate its argument, but in order to evaluate `head xs` we do not have to evaluate `xs` all the way, but only to the point where it becomes an expression of the form `y:ys`. Then `head xs` can return `y` and `sqr (head xs)` can return `y*y`. More generally, an expression is said to be in *head normal form* if it is a function (such as `sqr`) or if it takes the form of a data constructor (such as `(:)`) applied to its arguments. Every expression in normal form (i.e. in fully reduced form) is in head normal form but not vice versa. For example, `(e1,e2)` is in head normal form (because it is equivalent to `(,) e1 e2`, where `(,)` is the data constructor for pairs), but is in normal form only if both `e1` and `e2` are. Of course, for numbers or booleans there is no distinction between the two kinds of normal form.

'Under lazy evaluation arguments are evaluated only when needed and then only once, and then maybe only to head normal form' is not as catchy a headline as before, but it does tell a better story.

Next, consider the following two definitions of the inductive case of the function `subseqs` that returns all the subsequences of a list:

```
subseqs (x:xs) = subseqs xs ++ map (x:) (subseqs xs)
subseqs (x:xs) = xss ++ map (x:) xss
                 where xss = subseqs xs
```

In the first definition the expression `subseqs xs` appears twice on the right-hand side, so it is evaluated twice when the subsequences of a given list are required. In the second definition this duplication of effort has been recognised by the programmer and a `where` clause has been used to ensure that `subseqs xs` is evaluated only once (we could also have used a `let` expression).

The important point is that you, the programmer, are in control of which definition you want. It is quite possible for Haskell to recognise the double occurrence and to *abstract* it away using the equivalent of an internal `let` expression. This is a well-known technique called *common subexpression elimination*. But Haskell doesn't do this, and for a very good reason: it can cause a *space leak*. The second definition of `subseqs (x:xs)` has the following problem: the list `subseqs xs` is constructed only once, but it is retained in its entirety in memory because its value is used again, namely in the second expression `map (x:) xss`.

Look at it this way: the first definition takes longer because computation is duplicated; the second definition is faster (though still exponential) but can rapidly run out of available space. After all, there are 2^n subsequences of a list of length n. There is a fundamental dichotomy in programming we can never get away from: to avoid doing something twice you have to use up space to store the result of doing it once.

Here is a related example. Consider the following two definitions in a script:

```
foo1 n = sum (take n primes)
 where
 primes     = [x | x <- [2..], divisors x == [x]]
 divisors x = [d | d <- [2..x], x `mod` d == 0]


foo2 n = sum (take n primes)
primes     = [x | x <- [2..], divisors x == [x]]
divisors x = [d | d <- [2..x], x `mod` d == 0]
```

The programmer who wrote `foo1` decided to structure their script by making the definitions of both `primes` and `divisors` local to the definition of `foo1`, presumably because neither definition was used elsewhere in the script. The programmer who wrote `foo2` decided to allow these two subsidiary definitions to float to the status of a global or *top-level* definition. You might think that doesn't make any

difference to the efficiency, but consider the following interaction with GHCi. (The command `:set +s` turns on some statistics which are printed after an expression is evaluated.)

```
ghci> :set +s
ghci> foo1 1000
3682913
(4.52 secs, 648420808 bytes)
ghci> foo1 1000
3682913
(4.52 secs, 648412468 bytes)
ghci> foo2 1000
3682913
(4.51 secs, 647565772 bytes)
ghci> foo2 1000
3682913
(0.02 secs, 1616096 bytes)
```

Why was the second evaluation of `foo2 1000` so much faster than the first, while the two evaluations of `foo1 1000` took the same time?

The answer is that in the definition of `foo2` the first 1000 elements of the list `primes` is demanded, so after evaluation `primes` now points to a list in which the first 1000 primes appear explicitly. The second evaluation of `foo 1000` does not require these primes to be computed again. Internally, the script has grown in size because `primes` now occupies at least 1000 units of space.

Programmer Three chooses to write `foo` in the following way:

```
foo3 = \n -> sum (take n primes)
 where
 primes     = [x | x <- [2..], divisors x == [x]]
 divisors x = [d | d <- [2..x], x `mod` d == 0]
```

This uses a lambda expression to express `foo3` at the function level, but otherwise the definition is exactly the same as that of `foo1`. The alternative

```
foo3 = sum . flip take primes
```

also works but seems a little obscure. Now we have

```
ghci> foo3 1000
3682913
(3.49 secs, 501381112 bytes)
```

```
ghci> foo3 1000
3682913
(0.02 secs, 1612136 bytes)
```

Again, the second evaluation is much faster than the first. Why is that?

To see what is going on, we can rewrite the two functions in the form

```
foo1 n = let primes = ... in
         sum (take n primes)
foo3   = let primes = ... in
         \n -> sum (take n primes)
```

Now you can appreciate that in the first definition `primes` is re-evaluated every time `foo1 1000` is called because it is bound to an application of `foo1` not to the function itself. It is theoretically possible that the local definitions in the first definition depend on `n`, so any such definitions have to be re-evaluated for each `n`. In the second definition the local definitions are bound to the function itself (and can't possibly depend on any argument to the function); consequently, they are evaluated only once. Of course, after evaluating `foo3 1000`, the local definition of `primes` will be expanded to an explicit list of 1000 elements followed by a recipe for evaluating the rest.

7.2 Controlling space

Suppose we define `sum` by `sum = foldl (+) 0`. Under lazy evaluation the expression `sum [1..1000]` is reduced as follows

```
  sum [1..1000]
= foldl (+) 0 [1..1000]
= foldl (+) (0+1) [2..1000]
= foldl (+) ((0+1)+2) [3..1000]
= ...
= foldl (+) (..((0+1)+2)+ ... +1000) []
= (..((0+1)+2)+ ... +1000)
= ...
= 500500
```

It requires 1000 units of space just to build up the arithmetic expression that sums the first 1000 numbers before it pops to the surface and is finally evaluated.

Much better is to use a mixture of lazy and eager evaluation:

```
  sum [1..1000]
= foldl (+) 0 [1..1000]
= foldl (+) (0+1) [2..1000]
= foldl (+) 1 [2..1000]
= foldl (+) (1+2) [3..1000]
= foldl (+) 3 [3..1000]
= ...
= foldl (+) 500500 []
= 500500
```

While the list expression `[1..1000]` is evaluated lazily, the second argument of `foldl`, the accumulated sum, is evaluated eagerly. The result of interleaving lazy and eager evaluation steps is a sequence that uses a constant amount of space.

This suggests that it would be useful to have some way of controlling the reduction order. Such a method is provided by a primitive function `seq` with type

```
seq :: a -> b -> b
```

Evaluation of `x `seq` y` proceeds by first evaluating `x` (to head normal form) and then returning the result of evaluating `y`. If evaluation of `x` does not terminate, then neither does `x `seq` y`. It's not possible to define `seq` in Haskell; instead Haskell provides it as a primitive function.

Now consider the following version `foldl'` of `foldl` that evaluates its second argument strictly:

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f e []     = e
foldl' f e (x:xs) = y `seq` foldl' f y xs
                    where y = f e x
```

Haskell provides the function `foldl'` in the standard prelude (yes, with just this unimaginative name). Now we can define `sum = foldl' (+) 0`, with the consequence that evaluation proceeds in constant space. In fact, `sum` is another prelude function with essentially this definition.

Is it the case that `foldl` is now redundant and can be replaced by the new improved `foldl'`? The answer is in practice yes, but in theory no. It is possible to construct `f`, `e` and `xs` such that

$$\texttt{foldl f e xs} \neq \texttt{foldl' f e xs}$$

However, when `f` is strict (recall that f is strict if $f\ \bot = \bot$) the two expressions do return the same result. The exercises go into details.

Taking the mean

Armed with the above information, let's now consider a very instructive example: how to compute the average or *mean* of a list of numbers. Surely that is an easy problem, you might think, just divide the sum of the list by the length of the list:

```
mean :: [Float] -> Float
mean xs = sum xs / length xs
```

There are lots of things wrong with this definition, not the least of which is that the expression on the right is not well-formed! The function `length` in Haskell has type `[a] -> Int` and we can't divide a `Float` by an `Int` without performing an explicit conversion.

There is a function in the standard prelude that comes to our aid:

```
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```

Recall from Chapter 3 the two conversion functions

```
toInteger   :: (Integral a) => a -> Integer
fromInteger :: (Num a) => Integer -> a
```

The first converts any integral type to an integer, and the second converts an integer to a number. Their composition converts an integral number, such as `Int`, to a more general kind of number, such as `Float`.

We can now rewrite `mean` to read

```
mean :: [Float] -> Float
mean xs = sum xs / fromIntegral (length xs)
```

The second thing wrong with this definition is that it silently ignores the case of the empty list. What is `0/0`? Either we should identify the failing case with an explicit error message, or else adopt one common convention, which is to agree that the mean of the empty list should be zero:

```
mean [] = 0
mean xs = sum xs / fromIntegral (length xs)
```

Now we are ready to see what is *really* wrong with `mean`: it has a space leak. Evaluating `mean [1..1000]` will cause the list to be expanded and retained in memory after summing because there is a *second pointer* to it, namely in the computation of its length.

We can replace the two traversals of the list by one, using a strategy of program optimisation called *tupling*. The idea is simple enough in the present example: define `sumlen` by

```
sumlen :: [Float] -> (Float,Int)
sumlem xs = (sum xs,length xs)
```

and then calculate an alternative definition that avoids the two traversals. It is easy to carry out the calculation and we just state the result:

```
sumlen []     = (0,0)
sumlen (x:xs) = (s+x,n+1)  where (s,n) = sumlen xs
```

The pattern of the definition of `sumlen` should be familiar by now. An alternative definition is

```
sumlen = foldr f (0,0)  where f x (s,n) = (s+x,n+1)
```

Even better, we can replace `foldr f` by `foldl g`, where

```
g (s,n) x = (s+x,n+1)
```

The justification of this step is the law in the previous chapter that said

```
foldr f e xs = foldl g e xs
```

for all finite lists `xs`, provided

```
f x (g y z) = g (f x y) z
f x e = g e x
```

The verification of these two conditions is left as an exercise.

And that means we can use `foldl'`:

```
sumlen = foldl' g (0,0)  where g (s,n) x = (s+x,n+1)
```

Now we can replace our heavily criticised definition of `mean` by

```
mean [] = 0
mean xs = s / fromIntegral n
          where (s,n) = sumlen xs
```

Surely we have now achieved our goal of a constant-space computation for `mean`?

Unfortunately not. The problem is with `sumlen` and it is a little tricky to spot. Expanding the definition out a little, we find

```
foldl' f (s,n) (x:xs) = y `seq` foldl' f y xs
                        where y = (s+x,n+1)
```

Ah, but `y `seq` z` reduces `y` to head normal form and the expression `(s+x,n+1)` is already in head normal form. Its two components are not evaluated until the end of the computation. That means we have to dig deeper with our `seq`s and rewrite `sumlen` in the following way:

```
sumlen = foldl' f (0,0)
  where f (s,n) x = s `seq` n `seq` (s+x,n+1)
```

Finally, everything in the garden is rosy and we have a computation that runs in constant space.

Two more application operators

Function application is the only operation not denoted by any visible sign. However, Haskell provides two more application operators, `($)` and `($!)`:

```
infixr 0 $,$!
($),($!) :: (a -> b) -> a -> b
f $ x  = f x
f $! x = x `seq` f x
```

The only difference between `f x` and `f $! x` is that in the second expression the argument `x` is evaluated *before* `f` is applied. The only difference between `f x` and `f $ x` is that `($)` (and also `($!)`) is declared to have the lowest binding power of 0 and to associate to the right in expressions. That is exactly what the *fixity* declaration in the first line provides. Why do we want that?

The answer is that we can now write, for example

```
process1 $ process2 $ process3 input
```

instead of having to write either of

```
process1 (process2 (process3 x))
(process1 . process2 . process3) x
```

It is undeniable that `($)` can be quite useful on occasions, especially when submitting expressions for evaluation with GHCi, so it's worth mentioning its existence. And the strict application operator `($!)` is useful for the reasons discussed above.

7.3 Controlling time

We have seen that having an 'eager' button on our dashboard is a very simple way of controlling the space involved in driving a computation, but what about time? Unfortunately there is no analogous button for speeding up computations; instead we have to understand some of the things that can unintentionally slow down a computation. The Haskell platform comes with documentation on GHC, which contains useful advice on how to make your program run more quickly. The documentation makes three key points:

- Make use of GHC's *profiling* tools. There is no substitute for finding out where your program's time and space is really being used up. We will not discuss profiling in this book, but it is important to mention that such tools are available.

- The best way to improve a program's performance is to use a better algorithm. We mentioned this point at the beginning of the chapter.

- It is far better to use library functions that have been Seriously Tuned by Someone Else, than to craft your own. You might be able to write a better sorting algorithm than the one provided in `Data.List`, but it will take you longer than just writing `import Data.List (sort)`. This is particularly true when you use GHCi because GHCi loads *compiled* versions of the functions in its standard libraries. Compiled functions typically run about an order of magnitude faster than interpreted ones.

Much of the detailed advice in the GHC documentation is beyond the scope of this book, but two tips can be explained here. Firstly, the management of lazy evaluation involves more overheads than eager evaluation, so that if you know that a function's value will be needed, it is better to push the eager button. As the documentation says: 'Strict functions are your dear friends'.

The second piece of advice is about types. Firstly, `Int` arithmetic is faster than `Integer` arithmetic because Haskell has to perform more work in handling potentially very large numbers. So, use `Int` rather than `Integer` whenever it is safe to do so. Secondly, there is less housekeeping work for Haskell if you tailor the type of your function to the instance you want. For example, consider the type of `foo1`, defined in Section 7.1. There we did not provide a type signature for `foo1` (or indeed for any of the other related functions) and that was a mistake. It turns out that

```
foo1 :: Integral a => Int -> a
```

If we are really interested in the sum of the first n prime numbers, it is better to declare the type of `foo1` to be (say)

```
foo1 :: Int -> Integer
```

With this more specialised definition Haskell does not have to carry around a dictionary of the methods and instances of the type class `Integral`, and that lightens the load.

These pieces of advice can help shave off constant amounts of time and do not affect *asymptotic* time complexity, the order of magnitude of the timing function. But sometimes we can write code that is inadvertently less efficient asymptotically than we intended. Here is an instructive example. Consider the cartesian product function `cp` discussed in Chapter 5:

```
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

Pretty and clear enough you would think, but compare it with

```
cp' = foldr op [[]]
 where op xs yss = [x:ys | x <- xs, ys <- yss]
```

The first version is a direct recursive definition, while the second uses `foldr` to encapsulate the pattern of the recursion. The two 'algorithms' are the same, aren't they? Well,

```
ghci> sum $ map sum $ cp [[1..10] | j <- [1..6]]
33000000
(12.11 secs, 815874256 bytes)
ghci> sum $ map sum $ cp' [[1..10] | j <- [1..6]]
33000000
(4.54 secs, 369640332 bytes)
```

The expression `sum $ map sum` is there just to force complete evaluation of the cartesian product. Why is the first computation three times slower than the second?

To answer this question, look at the translation that eliminates the list comprehension in the first definition:

```
cp [] = [[]]
cp (xs:xss) = concat (map f xs)
     where f x = [x:ys | ys <- cp xss]
```

Now we can see that `cp xss` is evaluated *each time* `f` is applied to elements of `xs`. That means, in the examples above, that `cp` is evaluated many more times in

the first example than in the second. We cannot be more precise at this point, but will be below when we develop a little calculus for estimating running times. But the issue should be clear enough: the simple recursive definition of `cp` has led us inadvertently into a situation in which more evaluations are carried out than we intended.

One other way to get a more efficient cartesian product is to just write

```
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- yss]
              where yss = cp xss
```

This definition has exactly the same efficiency as the one in terms of `foldr`. The lesson here is that innocent-looking list comprehensions can hide the fact that some expressions, though only written once, are evaluated multiple times.

7.4 Analysing time

Given the definition of a function `f` we will write $T(\texttt{f})(n)$ to denote an asymptotic estimate of the number of reduction steps required to evaluate `f` on an argument of 'size' n in the worst case. Moreover, for reasons explained in a moment, we will assume eager, not lazy, evaluation as the reduction strategy involved in defining T.

The definition of T requires some amplification. Firstly, $T(\texttt{f})$ refers to the complexity of a given *definition* of `f`. Time complexity is a property of an expression, not of the value of that expression.

Secondly, the number of reduction steps does not correspond exactly to the elapsed time between submitting an expression for evaluation and waiting for the answer. No account is taken of the time to find the next subexpression to be reduced in a possibly large and complicated expression. For this reason the statistics facility of GHCi does not count reduction steps, but produces a measure of elapsed time.

Thirdly, we do not formalise the notion of size, since different measures are appropriate in different situations. For example, the cost of evaluating `xs++ys` is best measured in terms of (m,n), a pair describing the lengths of the two lists. In the case of `concat xss` we could take the length of `concat xss` as a measure of size, but if `xss` is a list of length m consisting of lists all of length n, then (m,n) might be a more suitable measure.

The fourth and crucial remark is that $T(\texttt{f})(n)$ is determined under an *eager* evaluation model of reduction. The reason is simply that estimating the number of

reduction steps under lazy evaluation is difficult. To illustrate, consider the definition `minimum = head . sort`. Under eager evaluation, the time to evaluate the minimum on a list of length n under this definition is given by

$$T(\texttt{minimum})(n) = T(\texttt{sort})(n) + T(\texttt{head})(n).$$

In other words we first have to completely sort a list of length n and then take the head of the result (presumably a constant-time operation). This equation does not hold under lazy evaluation, since the number of reduction steps required to find the head of `sort xs` requires only that `sort xs` be reduced to head normal form. How long that takes depends on the precise algorithm used for `sort`. Timing analysis under eager reduction is simpler because it is *compositional*. Since lazy evaluation never requires more reduction steps than eager evaluation, any upper bound for $T(\texttt{f})(n)$ will also be an upper bound under lazy evaluation. Furthermore, in many cases of interest, a lower bound will also be a lower bound under lazy evaluation.

In order to give some examples of timing analyses we have to introduce a little order notation. So far, we have used the awkward phrase 'taking a number of steps proportional to' whenever efficiency is discussed. It is time to replace it by something shorter. Given two functions f and g on the natural numbers, we say that f is of order g, and write $f = \Theta(g)$ if there are positive constants C_1 and C_2 and a natural number n_0 such that $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n > n_0$. In other words, f is bounded above and below by some constant times g for all sufficiently large arguments.

The notation is abused to the extent that one conventionally writes, for example, $f(n) = \Theta(n^2)$ rather than the more correct $f = \Theta(\lambda n.n^2)$. Similarly, one writes $f(n) = \Theta(n)$ rather than $f = \Theta(id)$. The main use of Θ-notation is to hide constants; for example, we can write

$$\sum_{j=1}^{n} j = \Theta(n^2) \quad \text{and} \quad \sum_{j=1}^{n} j^2 = \Theta(n^3)$$

without bothering about the exact constants involved. When $\Theta(g)$ appears in a formula it stands for some unnamed function f satisfying $f = \Theta(g)$. In particular, $\Theta(1)$ denotes an anonymous constant.

With that behind us, we give three examples of how to analyse the running time of a computation. Consider first the following two definitions of `concat`:

```
concat xss  = foldr (++) [] xss
concat' xss = foldl (++) [] xss
```

The two definitions are equivalent provided `xss` is a finite list. Suppose `xss` is a

list of length m of lists all of length n. Then the first definition gives

$$\begin{aligned} T(\texttt{concat})(m,n) &= T(\texttt{foldr (++) []})(m,n),\\ T(\texttt{foldr (++) []})(0,n) &= \Theta(1),\\ T(\texttt{foldr (++) []})(m{+}1,n) &= T(\texttt{++})(n,mn) + \\ &\qquad T(\texttt{foldr (++) []})(m,n). \end{aligned}$$

The estimate $T(\texttt{++})(n,mn)$ arises because a list of length n is concatenated with a list of length mn. Since $T(\texttt{++})(n,m) = \Theta(n)$, we obtain

$$T(\texttt{foldr (++) []})(m,n) = \sum_{k=0}^{m} \Theta(n) = \Theta(mn).$$

For the second definition of `concat` we have

$$\begin{aligned} T(\texttt{concat'})(m,n) &= T(\texttt{foldl (++)})(0,m,n),\\ T(\texttt{foldl (++)})(k,0,n) &= O(1),\\ T(\texttt{foldl (++)})(k,m{+}1,n) &= T(\texttt{++})(k,n) + \\ &\qquad T(\texttt{foldl (++)})(k{+}n,m,n). \end{aligned}$$

The additional argument k refers to the length of the accumulated list in the second argument of `foldl`. This time we obtain

$$T(\texttt{foldl (++)})(k,m,n) = \sum_{j=0}^{m-1} \Theta(k+jn) = \Theta(k+m^2n).$$

Hence $T(\texttt{concat'})(m,n) = \Theta(m^2n)$. The conclusion, which was anticipated in the previous chapter, is that using `foldr` rather than `foldl` in the definition of `concat` leads to an asymptotically faster program.

For the second example let us time the two programs for `subseqs` discussed in Section 7.1, where we had either of the following two possibilities:

```
subseqs (x:xs)  = subseqs xs ++ map (x:) (subseqs xs)
subseqs' (x:xs) = xss ++ map (x:) xss
                  where xss = subseqs' xs
```

Bearing in mind that (i) if `xs` has length n, then `subseqs xs` has length 2^n; and (ii) the time for both the concatenation and for applying `map (x:)` is therefore $\Theta(2^n)$, the two timing analyses give

$$\begin{aligned} T(\texttt{subseqs})(n{+}1) &= 2T(\texttt{subseqs})(n) + \Theta(2^n),\\ T(\texttt{subseqs'})(n{+}1) &= T(\texttt{subseqs'})(n) + \Theta(2^n) \end{aligned}$$

together with $T(\texttt{subseqs})(0) = \Theta(1)$. We will just state the two solutions (which can be proved by a simple induction argument):

$$T(\texttt{subseqs})(n) = \Theta(n2^n),$$
$$T(\texttt{subseqs'})(n) = \Theta(2^n).$$

The latter is therefore asymptotically faster than the former by a logarithmic factor.

For the third example, let us time the two programs for `cp` discussed at the beginning of this section. The first one was

```
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- cp xss]
```

Suppose once again that `xss` is a list of length m of lists all of length n. Then the length of `cp xss` is n^m. Then we have

$$T(\texttt{cp})(0,n) = \Theta(1),$$
$$T(\texttt{cp})(m+1,n) = nT(\texttt{cp})(m,n) + \Theta(n^m).$$

because it takes $\Theta(n^m)$ steps to apply `(x:)` to every subsequence. The solution is

$$T(\texttt{cp})(m,n) = \Theta(mn^m).$$

On the other hand, the definition of `cp` in terms for `foldr` gives

$$T(\texttt{cp})(0,n) = \Theta(1),$$
$$T(\texttt{cp})(m+1,n) = T(\texttt{cp})(m,n) + \Theta(n^m).$$

with solution $T(\texttt{cp})(m,n) = \Theta(n^m)$. The second version is therefore asymptotically faster, again by a logarithmic factor.

7.5 Accumulating parameters

Sometimes we can improve the running time of a computation by adding an extra argument, called an *accumulating parameter*, to a function. The canonical example is the function `reverse`:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

With this definition we have $T(\texttt{reverse})(n) = \Theta(n^2)$. In search of a linear-time program, suppose we define

```
revcat :: [a] -> [a] -> [a]
revcat xs ys = reverse xs ++ ys
```

It is clear that `reverse xs` $=$ `revcat xs []`, so if we can obtain an efficient version of `revcat` we can obtain an efficient version of `reverse`. To this end we calculate a recursive definition of `revcat`. The base case `revcat [] ys = ys` is left as an exercise, and the inductive case is as follows:

```
  revcat (x:xs) ys
=   {definition of revcat}
  reverse (x:xs) ++ ys
=   {definition of reverse}
  (reverse xs ++ [x]) ++ ys
=   {associativity of (++)}
  reverse xs ++ ([x] ++ ys)
=   {definition of (:)}
  reverse xs ++ (x:ys)
=   {definition of revcat}
  revcat xs (x:ys)
```

Hence

```
revcat [] ys     = ys
revcat (x:xs) ys = revcat xs (x:ys)
```

As to the running time, $T(\texttt{revcat})(m,n) = \Theta(m)$. In particular,

$$T(\texttt{reverse}(n) = T(\texttt{revcat}(n,0) = \Theta(n)$$

That gives a linear-time computation for reversing a list.

Here is another example. The function `length` is defined by

```
length :: [a] -> Int
length []     = 0
length (x:xs) = length xs + 1
```

We have $T(\texttt{length})(n) = \Theta(n)$, so there is no time advantage in calculating another definition. Nevertheless, define `lenplus` by

```
lenplus :: [a] -> Int -> Int
lenplus xs n = length xs + n
```

If we go through exactly the same calculation for `lenplus` as we did for `revcat`, we arrive at

```
lenplus [] n     = n
lenplus (x:xs) n = lenplus xs (1+n)
```

The reason the calculation goes through is that `(+)`, like `(++)`, is an associative operation. The advantage of defining

```
length xs = lenplus xs 0 = foldl (\n x -> 1+n) 0 xs
```

is that, by using `foldl'` in place of `foldl`, the length of a list can be computed in constant space. That indeed is how `length` is defined in Haskell's prelude.

As the really astute reader might have spotted, there is actually no need to go through the calculations above. Both the examples are, in fact, instances of a law already described in the previous chapter, namely that

```
foldr (<>) e xs = foldl (@) e xs
```

for all finite lists `xs` provided

```
x <> (y @ z) = (x <> y) @ z
x <> e = e @ x
```

The two instances are:

```
foldr (\x n -> n+1) 0 xs = foldl (\n x -> 1+n) 0 xs
foldr (\x xs -> xs++[x]) [] xs
                         = foldl (\xs x -> [x]++xs) [] xs
```

We leave the detailed verification of these equations as an exercise.

For a final demonstration of the accumulating parameter technique we move from lists to trees. Consider the data declaration

```
data GenTree a = Node a [GenTree a]
```

An element of this type is a tree consisting of a node with a label and a list of subtrees. Such trees arise in problems that can be formulated in terms of positions and moves. The label of a node specifies the current position, and the number of subtrees corresponds to the number of possible moves in the current position. Each subtree has a label that specifies the result of making the move, and its subtrees describe the moves that can be made from the new position. And so on.

Here is a function for computing the list of labels in a tree:

```
labels :: GenTree a -> [a]
labels (Node x ts) = x:concat (map labels ts)
```

The method is simple enough: compute the labels of each subtree, concatenate the results, and stick the label of the tree at the front of the final list.

Let us analyse the running time of this program on a tree t. To keep things simple, suppose that t is a *perfect* k-ary tree of height h. What that means is that if $h = 1$ then t has no subtrees, while if $h > 1$ then t has exactly k subtrees, each with height $h-1$. The number $s(h,k)$ of labels in such a tree satisfies

$$\begin{aligned} s(1,t) &= 1, \\ s(h+1,k) &= 1 + ks(h,k), \end{aligned}$$

with solution $s(h,k) = \Theta(k^h)$. Now we have

$$\begin{aligned} T(\texttt{labels})(1,k) &= \Theta(1), \\ T(\texttt{labels})(h+1,k) &= \Theta(1) + T(\texttt{concat})(k,s) + T(\texttt{map labels})(h,k), \end{aligned}$$

where $s = s(h,k)$. The term $T(\texttt{map labels})(h,k)$ estimates the running time of applying `map labels` to a list of length k of trees all of height h. In general, given a list of length k consisting of elements each of size n, we have

$$T(\texttt{map f})(k,n) = kT(\texttt{f})(n) + \Theta(k).$$

Furthermore $T(\texttt{concat})(k,s) = \Theta(ks) = \Theta(k^{h+1})$. Hence

$$T(\texttt{labels})(h+1,k) = \Theta(k^{h+1}) + kT(\texttt{labels})(h,k)$$

since $\Theta(1) + \Theta(k) = \Theta(k)$. The solution is given by

$$T(\texttt{labels})(h,k) = \Theta(hk^h) = \Theta(s \log s).$$

In words, computing the labels of a tree using the definition above takes time that is asymptotically greater than the size of the tree by a logarithmic factor.

Let us now see what an accumulating parameter can do. Define `labcat` by

```
labcat :: [GenTree a] -> [a] -> [a]
labcat ts xs = concat (map labels ts) ++ xs
```

As well as adding in a list `xs` we have also generalised the first argument from a tree to a list of trees. We have `labels t = labcat [t] []`, so any improvement on `labcat` leads to a corresponding improvement on `labels`.

We now synthesise an alternative definition for `labcat`. For the base case we obtain

```
labcat [] xs = xs
```

For the inductive case we reason:

```
  labcat (Node x us:vs) xs
=   {definition}
  concat (map labels (Node x us:vs)) ++ xs
=   {definitions}
  labels (Node x us) ++ concat (map labels vs) ++ xs
=   {definition}
  x:concat (map labels us) ++ concat (map labels vs) ++ xs
=   {definition of labcat}
  x:concat (map labels us) ++ labcat vs xs
=   {definition of labcat (again)}
  labcat us (labcat vs xs)
```

The result of this calculation is the following program for `labels`:

```
labels t = labcat [t] []
labcat [] xs          = xs
labcat (Node x us:vs) = x:labcat us (labcat vs xs)
```

For the timing analysis, let $T(\texttt{labcat})(h,k,n)$ estimate the running time of

```
labcat ts xs
```

when `ts` is a list of length n of trees, each of which is a perfect k-ary tree of height h (the size of `xs` is ignored since it doesn't affect the estimate). Then

$$\begin{aligned} T(\texttt{labcat})(h,k,0) &= \Theta(1), \\ T(\texttt{labcat})(1,k,n+1) &= \Theta(1) + T(\texttt{labcat})(1,k,n)), \\ T(\texttt{labcat})(h+1,k,n+1) &= \Theta(1) + T(\texttt{labcat})(h,k,k) + \\ &\qquad T(\texttt{labcat})(h+1,k,n). \end{aligned}$$

Solving the first two equations gives $T(\texttt{labcat})(1,k,n) = \Theta(n)$. An induction argument now shows $T(\texttt{labcat})(h,k,n) = \Theta(k^h n)$. Hence

$$T(\texttt{labels})(h,k) = T(\texttt{labcat})(h,k,1) = \Theta(k^h) = \Theta(s).$$

That means we can compute the labels of a tree in time proportional to the size of the tree, a logarithmic improvement over our first version.

7.6 Tupling

We met the idea of tupling two functions in the discussion of the function `mean`. Tupling is sort of dual to the method of accumulating parameters: we generalise a function not by including an extra argument but by including an extra result.

The canonical example of the power of tupling is the Fibonacci function:

```
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

The time to evaluate `fib` by these three equations is given by

$$\begin{aligned} T(\texttt{fib})(0) &= \Theta(1), \\ T(\texttt{fib})(1) &= \Theta(1), \\ T(\texttt{fib})(n) &= T(\texttt{fib})(n-1) + T(\texttt{fib})(n-2) + \Theta(1). \end{aligned}$$

The timing function therefore satisfies equations very like that of `fib` itself. In fact $T(\texttt{fib})(n) = \Theta(\phi^n)$, where ϕ is the golden ratio $\phi = (1+\sqrt{5})/2$. That means that the running time to compute `fib` on an input n is exponential in n.

Now consider the function `fib2` defined by

```
fib2 n = (fib n,fib (n+1))
```

Clearly `fib n = fst (fib2 n)`. Synthesis of a direct recursive definition of `fib2` yields

```
fib2 0 = (0,1)
fib2 n = (b,a+b)  where (a,b) = fib2 (n-1)
```

This program takes linear time. In this example the tupling strategy leads to a dramatic increase in efficiency, from exponential time to linear time.

It's great fun to formulate general laws that encapsulate gains in efficiency. One such law concerns the computation of

```
(foldr f a xs, foldr g b xs)
```

As expressed above, the two applications of `foldr` involve two traversals of the list `xs`. There is a modest time advantage, and possibly a greater space advantage, in formulating a version that traverses the list only once. In fact

```
(foldr f a xs, foldr g b xs) = foldr h (a,b) xs
```

where

```
h x (y,z) = (f x y,g x z)
```

The result can be proved by induction and we leave details as an easy exercise.

As one more example, we again move from lists to trees. But this time we have a different kind of tree, a *leaf-labelled binary tree*:

```
data BinTree a = Leaf a | Fork (BinTree a) (BinTree a)
```

In contrast to a `GenTree` discussed above, a `BinTree` is either a leaf, with an associated label, or a fork of two subtrees.

Suppose we wanted to build such a tree with a given list as the labels. More precisely, we want to define a function `build` satisfying

```
labels (build xs) = xs
```

for all finite nonempty lists `xs`, where `labels` returns the labels of a binary tree:

```
labels :: BinTree a -> [a]
labels (Leaf x)   = [x]
labels (Fork u v) = labels u ++ labels v
```

We are attuned now to possible optimisations, and the definition of `labels` suggest that it could be improved with an accumulating parameter. So it can, but that is not our primary interest here, and we leave the optimisation as an exercise.

One way to build a tree is to arrange that half the list goes into the left subtree, and the other half into the right subtree:

```
build :: [a] -> BinTree a
build [x] = Leaf x
build xs  = Fork (build ys) (build zs)
            where (ys,zs) = halve xs
```

The function `halve` made an appearance in Section 4.8:

```
halve xs = (take m xs,drop m xs)
           where m = length xs `div` 2
```

Thus `halve` splits a list into two approximately equal halves. The definition of `halve` involves a traversal of the list to find its length, and two further (partial) traversals to compute the two components. It is therefore a prime candidate for applying the tupling strategy to get something better. But as with `labels` we are going to ignore that particular optimisation for now. And we are also going to

ignore the proof that this definition of `build` meets its specification. That's three calculations we are leaving as exercises in order to concentrate on a fourth.

Let's time `build`:

$$\begin{aligned} T(\texttt{build})(1) &= \Theta(1), \\ T(\texttt{build})(n) &= T(\texttt{build})(m) + T(\texttt{build})(n-m) + \Theta(n) \\ &\qquad \textbf{where } m = n \operatorname{div} 2 \end{aligned}$$

It takes $\Theta(n)$ steps to halve a list of length n, and then we recursively build two subtrees from lists of length m and $n-m$, respectively. The solution is

$$T(\texttt{build})(n) = \Theta(n \log n).$$

In words, building a tree by the above method takes longer than the length of the list by a logarithmic factor.

Having established this fact, let us define `build2` by

```
build2 :: Int -> [a] -> (BinTree a,[a])
build2 n xs = (build (take n xs),drop n xs)
```

This builds a tree from the first n elements, but also returns the list that is left. We have

```
build xs = fst (build2 (length xs) xs)
```

so our original function can be determined from the tupled version.

Our aim now is to construct a direct recursive definition of `build2`. First of all, it is clear that

```
build2 1 xs = (Leaf (head xs),tail xs)
```

For the recursive case we start with

```
build2 n xs = (Fork (build (take m (take n xs)))
                    (build (drop m (take n xs))),
               drop n xs)  where m = n `div` 2
```

This equation is obtained by substituting in the recursive case of `build`. It suggests that the next step is to use some properties of `take` and `drop`. Here they are: if `m <= n` then

```
take m . take n = take m
drop m . take n = take (n-m) . drop m
```

That leads to

```
build2 n xs = (Fork (build (take m xs))
                    (build (take (n-m) (drop m xs))),
                drop n xs)  where m = n `div` 2
```

Using the definition of `build2` we can rewrite the above as follows:

```
build2 n xs = (Fork u v, drop n xs)
              where (u,xs')  = build2 m xs
                    (v,xs'') = build2 (n-m) xs'
                    m        = n `div` 2
```

But as a final step, observe that

```
xs'' = drop (n-m) xs'
     = drop (n-m) (drop m xs)
     = drop n xs
```

Hence we can rewrite `build2` once again to read

```
build2 1 xs = (Leaf (head xs),tail xs)
build2 n xs = (Fork u v, xs'')
              where (u,xs')  = build2 m xs
                    (v,xs'') = build2 (n-m) xs'
                    m        = n `div` 2
```

Timing this program yields

$$T(\texttt{build2})(1) = \Theta(1),$$
$$T(\texttt{build2})(n) = T(\texttt{build2})(m) + T(\texttt{build2})(n-m) + \Theta(1).$$

with solution $T(\texttt{build2})(n) = \Theta(n)$. Using `build2` as a subsidiary function has therefore improved the running time of `build` by a logarithmic factor.

7.7 Sorting

Sorting is a big topic and one can spend many happy hours tinkering with different algorithms. Knuth devotes about 400 pages to the subject in Volume 3 of his series *The Art of Computer Programming*. Even then some of his conclusions have to be reformulated when sorting is considered in a purely functional setting. Here we briefly consider two sorting algorithms, keeping an eye out for possible optimisations.

Mergesort

The sorting method called *Mergesort* made an appearance in Section 4.8:

```
sort :: (Ord a) => [a] -> [a]
sort []  = []
sort [x] = [x]
sort xs  = merge (sort ys) (sort zs)
           where (ys,zs) = halve xs
halve xs = (take m xs,drop m xs)
           where m = length xs `div` 2
```

In fact there are a number of variants for sorting by merging, and the standard prelude function `sort` uses a different variant than the one above.

As we said above, the definition of `halve` looks fairly inefficient in that it involves multiple traversals of its argument. One way to improve matters is to make use of the standard prelude function `splitAt`, whose specification is

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs,drop n xs)
```

The prelude version of this function is the result of a tupling transformation:

```
splitAt 0 xs     = ([],xs)
splitAt n []     = ([],[])
splitAt n (x:xs) = (x:ys,zs)
                   where (ys,zs) = splitAt (n-1) xs
```

It is easy enough to calculate this definition using the two facts that

```
take n (x:xs) = x:take (n-1) xs
drop n (x:xs) = drop (n-1) xs
```

provided `0 < n`. Now we have

```
halve xs = splitAt (length xs `div` 2) xs
```

There are still two traversals here of course.

Another way to improve `sort` is to define

```
sort2 n xs = (sort (take n xs),drop n xs)
```

We have `sort xs = fst (sort2 (length xs) xs)`, so our original sorting function can be retrieved from the general one. An almost exactly similar calculation to the one in the previous section leads to

```
sort2 0 xs = ([],xs)
sort2 1 xs = ([head xs],tail xs)
sort2 n xs = (merge ys zs, xs'')
             where (ys,xs')  = sort2 m xs
                   (zs,xs'') = sort2 (n-m) xs'
                   m         = n `div` 2
```

With this definition there are no length calculations and no multiple traversals of `xs`.

Another way to optimise `halve` is to realise that no human would split up a list in this way if forced to do so by hand. If asked to divide a list into two, you and I would surely just deal out the elements into two piles:

```
halve []       = ([],[])
halve [x]      = ([x],[])
halve (x:y:xs) = (x:ys,y:zs)
                 where (ys,zs) = halve xs
```

Of course, this definition returns a different result than the previous one, but the order of the elements in the two lists does not matter if the result is to be sorted; what is important is that the elements are all there.

That is a total of three ways to improve the performance of `sort`. However, it turns out that none of them makes that much difference to the total running time. A few per cent perhaps, but nothing substantial. Furthermore, if we are using GHCi as our functional evaluator, none of the versions compares in performance to the library function `sort` because that function is given to us in a compiled form, and compiled versions of functions are usually about ten times faster. We can always compile our functions using GHC of course.

Quicksort

Our second sorting algorithm is a famous one called *Quicksort*. It can be expressed in just two lines of Haskell:

```
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort [y | y <- xs, y < x] ++ [x] ++
              sort [y | y <- xs, x <= y]
```

That's very pretty and a testament to the expressive power of Haskell. But the prettiness comes at a cost: the program can be very inefficient in its use of space. The situation is the same as with the program for `mean` seen earlier.

Before plunging into ways the code can be optimised, let's compute $T(\texttt{sort})$. Suppose we want to sort a list of length $n{+}1$. The first list comprehension can return a list of any length k from 0 to n. The length of the result of the second list comprehension is therefore $n{-}k$. Since our timing function is an estimate of the worst-case running time, we have to take the maximum of these possibilities:

$$\begin{aligned} &T(\texttt{sort})(n{+}1) \\ &= \max\,[T(\texttt{sort})(k) + T(\texttt{sort})(n{-}k) \;\mid\; k \leftarrow [0\,..\,n]] + \Theta(n). \end{aligned}$$

The $\Theta(n)$ term accounts for both the time to evaluate the two list comprehensions and the time to perform the concatenations. Note, by the way, the use of a list comprehension in a mathematical expression rather than a Haskell one. If list comprehensions are useful notations in programming, they are useful in mathematics too.

Although not immediately obvious, the worst case occurs when $k = 0$ or $k = n$. Hence

$$\begin{aligned} T(\texttt{sort})(0) &= \Theta(1), \\ T(\texttt{sort})(n{+}1) &= T(\texttt{sort})(n) + \Theta(n), \end{aligned}$$

with solution $T(\texttt{sort})(n) = \Theta(n^2)$. Thus Quicksort is a quadratic algorithm in the worst case. This fact is intrinsic to the algorithm and has nothing to do with the Haskell expression of it. Quicksort achieved its fame for two other reasons, neither of which hold in a purely functional setting. Firstly, when Quicksort is implemented in terms of arrays rather than lists, the partitioning phase can be performed *in place* without using any additional space. Secondly, the *average case* performance of Quicksort, under reasonable assumptions about the input, is $\Theta(n \log n)$ with a smallish constant of proportionality. In a functional setting this constant is not so small and there are better ways to sort than Quicksort.

With this warning, let us now see what we can do to optimise the algorithm without changing it in any essential way (i.e. to a completely different sorting algorithm). To avoid the two traversals of the list in the partitioning process, define

```
partition p xs = (filter p xs, filter (not . p) xs)
```

This is another example of tupling two definitions to save on a traversal. Since `filter p` can be expressed as an instance of `foldr` we can appeal to the tupling law of `foldr` to arrive at

```
partition p = foldr op ([],[])
 where op x (ys,zs) | p x       = (x:ys,zs)
                    | otherwise = (ys,x:zs)
```

Now we can write

```
sort []     = []
sort (x:xs) = sort ys ++ [x] ++ sort zs
              where (ys,zs) = partition (<x) xs
```

But this program still contains a space leak. To see why, let us write the recursive case in the equivalent form

```
sort (x:xs) = sort (fst p) ++ [x] ++ sort (snd p)
              where p = partition (<x) xs
```

Suppose `x:xs` has length $n+1$ and is in strictly decreasing order, so `x` is the largest element in the list and `p` is a pair of lists of length n and 0, respectively. Evaluation of `p` is triggered by displaying the results of the first recursive call, but the n units of space occupied by the first component of p cannot be reclaimed because there is another reference to `p` in the second recursive call. Between these two calls further pairs of lists are generated and retained. All in all, the total space required to evaluate `sort` on a strictly decreasing list of length $n+1$ is $\Theta(n^2)$ units. In practice this means that evaluation of `sort` on some large inputs can abort owing to lack of sufficient space.

The solution is to force evaluation of `partition` and, equally importantly, to bind `ys` and `zs` to the components of the pair, not to `p` itself.

One way of bringing about a happy outcome is to introduce two accumulating parameters. Define `sortp` by

```
sortp x xs us vs = sort (us ++ ys) ++ [x] ++
                   sort (vs ++ zs)
                   where (ys,zs) = partition (<x) xs
```

Then we have

```
sort (x:xs) = sortp x xs [] []
```

We now synthesise a direct recursive definition of `sortp`. The base case is

```
sortp x [] us vs = sort us ++ [x] ++ sort vs
```

For the recursive case `y:xs` let us assume that `y < x`. Then

```
  sortp x (y:xs) us vs
=   {definition of sortp with (ys,zs) = partition (<x) xs}
  sort (us ++ y:ys) ++ [x] ++ sort (vs ++ zs)
=   {claim (see below)}
  sort (y:us ++ ys) ++ [x] ++ sort (vs ++ zs)
=   {definition of sortp}
  sortp x (y:us) vs
```

The claim is that if `as` is any permutation of `bs` then `sort as` and `sort bs` return the same result. The claim is intuitively obvious: sorting a list depends only on the elements in the input not on their order. A formal proof is omitted.

Carrying out a similar calculation in the case that `x <= y` and making `sortp` local to the definition of `sort`, we arrive at the final program

```
sort []     = []
sort (x:xs) = sortp xs [] []
  where
  sortp [] us vs     = sort us ++ [x] ++ sort vs
  sortp (y:xs) us vs = if y < x
                       then sortp xs (y:us) vs
                       else sortp xs us (y:vs)
```

Not quite as pretty as before, but at least the result has $\Theta(n)$ space complexity.

7.8 Exercises

Exercise A

One simple definition of `sort` is

```
sort []     = []
sort (x:xs) = insert x (sort xs)
insert x [] = [x]
insert x (y:ys)
     = if x <= y then x:y:ys else y:insert x ys
```

This method is called *insertion sort*. Reduce `sort [3,4,2,1]` to head normal

form under lazy evaluation. Now answer the following questions: (i) How long, as a function of n, does it take to compute `head . sort` when applied to a list of length n? (ii) How long does it take under eager evaluation? (iii) Does insertion sort, evaluated lazily, carry out exactly the same sequence of comparisons as the following *selection sort* algorithm?

```
sort [] = []
sort xs = y:sort ys  where (y,ys) = select xs

select [x]    = (x,[])
select (x:xs) | x <= y    = (x,y:ys)
              | otherwise = (y,x:ys)
              where (y,ys) = select xs
```

Exercise B

Write down a definition of `length` that evaluates in constant space. Write a second definition of `length` that evaluates in constant space but does not make use of the primitive `seq` (either directly or indirectly).

Exercise C

Construct `f`, `e` and `xs` so that

```
foldl f e xs ≠ foldl' f e xs
```

Exercise D

Would

```
cp []       = [[]]
cp (xs:xss) = [x:ys | ys <- cp xss, x <- xs]
```

be an alternative way of defining the function `cp` that is as efficient as the definition in terms of `foldr`? Yes, No or Maybe?

Time for a calculation. Use the fusion law of `foldr` to calculate an efficient alternative to

```
fcp = filter nondec . cp
```

See Section 4.7 for a definition of `nondec`.

Exercise E

Suppose

$$T(1) = \Theta(1),$$
$$T(n) = T(n \operatorname{div} 2) + T(n - n \operatorname{div} 2) + \Theta(n)$$

for $2 \leq n$. Prove that $T(2^k) = \Theta(k2^k)$. Hence prove $T(n) = \Theta(n \log n)$.

Exercise F

Prove that

```
foldr (\x n -> n+1) 0 xs = foldl (\n x -> 1+n) 0 xs
foldr (\x xs -> xs++[x]) [] xs
                         = foldl (\xs x -> [x]++xs) [] xs
```

Exercise G

Prove that if `h x (y,z) = (f x y,g x z)`, then

```
(foldr f a xs,foldr g b xs) = foldr h (a,b) xs
```

for all finite lists `xs`. A tricky question: does the result hold for *all* lists `xs`?

Now find a definition of `h` such that

```
(foldl f a xs,foldl g b xs) = foldl h (a,b) xs
```

Exercise H

Recall that

```
partition p xs = (filter p xs, filter (not . p) xs)
```

Express the two components of the result as instances of `foldr`. Hence use the result of the previous exercise to calculate another definition of `partition`.

Define

```
part p xs us vs = (filter p xs ++ us,
                   filter (not . p) xs ++ vs)
```

Calculate another definition of `partition` that uses `part` as a local definition.

Exercise I

Recall that

```
labels :: BinTree a -> [a]
labels (Leaf x)   = [x]
labels (Fork u v) = labels u ++ labels v
```

Compute $T(\texttt{labels})(n)$, where n is the number of leaves in the tree. Now use the accumulating parameter technique to find a faster way of computing `labels`.

Prove that `labels (build xs) = xs` for all finite nonempty lists `xs`.

Exercise J

Define `select k = (!!k) . sort`, where `sort` is the original Quicksort. Thus `select k` selects the kth smallest element of a nonempty finite list of elements, the 0th smallest being the smallest element, the 1st smallest being the next smallest element, and so on. Calculate a more efficient definition of `select` and estimate its running time.

7.9 Answers

Answer to Exercise A

```
  sort [3,4,1,2]
= insert 3 (sort [4,1,2])
= ...
= insert 3 (insert 4 (insert 1 (insert 2 [])))
= insert 3 (insert 4 (insert 1 (2:[])))
= insert 3 (insert 4 (1:2:[]))
= insert 3 (1:insert 4 (2:[]))
= 1:insert 3 (insert 4 (2:[]))
```

It takes $\Theta(n)$ steps to compute `head . sort` on a list of length n. Under eager evaluation it takes about n^2 steps. As to part (iii), the answer is yes. You may think we have defined sorting by insertion, but under lazy evaluation it turns out to be selection sort. The lesson here is that, under lazy evaluation, you don't always get what you think you are getting.

Answer to Exercise B

For the first part, the following does the job:

```
length = foldl' (\n x -> n+1) 0
```

For the second part, one solution is

```
length           = length2 0
length2 n []     = n
length2 n (x:xs) = if n==0 then length2 1 xs
                   else length2 (n+1) xs
```

The test `n==0` forces evaluation of the first argument.

Answer to Exercise C

Take `f n x = if x==0 then undefined else 0`. Then

```
foldl f 0 [0,2]  = 0
foldl' f 0 [0,2] = undefined
```

Answer to Exercise D

The answer is: maybe! Although the given version of `cp` is efficient, it returns the component lists in a different order than any of the definitions in the text. That probably doesn't matter if we are only interested in the *set* of results, but it might affect the running time and result of any program that searched `cp` to find some list satisfying a given property.

According to the fusion rule we have to find a function `g` so that

```
filter nondec (f xs yss) = g xs (filter nondec yss)
```

where `f xs yss = [x:ys | x <- xs, ys <- yss]`. Then we would have

```
filter nondec . cp
 = filter nondec . foldr f [[]]
 = foldr g [[]]
```

Now

```
nondec (x:ys) = null ys || (x <= head ys && nondec ys)
```

That leads to

```
g xs [[]] = [[x] | x <- xs]
g xs yss  = [x:ys | x <- xs, ys <- yss, x <= head ys]
```

Answer to Exercise E

For the first part, we have

$$T(2^k) = 2T(2^{k-1}) + \Theta(2^k).$$

By induction we can show $T(2^k) = \sum_{i=0} k\Theta(2^k)$. The induction step is

$$\begin{aligned} T(2^k) &= 2\sum_{i=0}^{k-1} \Theta(2^{k-1}) + \Theta(2^k) \\ &= \sum_{i=0}^{k-1} \Theta(2^k) + \Theta(2^k) \\ &= \sum_{i=0}^{k} \Theta(2^k). \end{aligned}$$

Hence $T(2^k) = \Theta(k2^k)$. Now suppose $2^k \leq n < 2^{k+1}$, so

$$\Theta(k2^k) = T(2^k) \leq T(n) \leq T(2^{k+1}) = \Theta((k+1)2^{k+1}) = \Theta(k2^k).$$

Hence $T(n) = \Theta(k2^k) = \Theta(n \log n)$.

Answer to Exercise F

Define `x <> n = n+1` and `n @ x = 1+n`. We have

```
(x <> n) @ y = 1+(n+1) = (1+n)+1 = x <> (n @ y)
```

The second proof is similar.

Answer to Exercise G

The induction step is

```
  (foldr f a (x:xs),foldr g b (x:xs)
= (f x (foldr f a xs),g x (foldr g b xs))
= h x (foldr f a xs,foldr g b xs)
= h x (foldr h (a,b) xs
= foldr h (a,b) (x:xs)
```

The answer to the tricky question is No. The values $(\bot, \bot)$ and $\bot$ are different in Haskell. For example, suppose we define `foo (x,y) = 1`. Then

```
foo undefined = undefined
foo (undefined,undefined) = 1
```

For the last part, the definition of `h` is that

```
h (y,z) x = (f y x,g z x)
```

Answer to Exercise H

We have `filter p = foldr (op p) []`, where

```
op p x xs = if p x then x:xs else xs
```

Now

```
 (op p x ys,op (not . p) x zs)
= if p x then (x:ys,zs) else (ys,x:zs)
```

Hence

```
partition p xs = foldr f ([],[]) xs
  where f x (ys,zs) = if p x
                      then (x:ys,zs)
                      else (ys,x:zs)
```

For the last part we obtain

```
partition p xs = part p xs [] []
part p [] ys zs = (ys,zs)
part p (x:xs) ys zs = if p x
                      then part p xs (x:ys) zs
                      else part p xs ys (z:zs)
```

Answer to Exercise I

Remember that T estimates the *worst case* running time. The worst case for `labels` arises when every right subtree of the tree is a leaf. Then we have

$$T(\texttt{labels})(n) = T(\texttt{labels})(n-1) + \Theta(n),$$

where $\Theta(n)$ accounts for the time to concatenate a list of length $n-1$ with a list of length 1. Hence

$$T(\texttt{labels})(n) = \sigma_{j=0}^{n}\Theta(n) = \Theta(n^2).$$

The accumulating parameter method yields

```
labels t              = labels2 t []
labels2 (Leaf x) xs   = x:xs
labels2 (Fork u v) xs = labels2 u (labels2 v xs)
```

and $T(\texttt{labels2})(n) = \Theta(n)$. This improves the running time of `labels` from quadratic to linear time.

The induction step in the proof that `labels (build xs) = xs` is to assume the

hypothesis for all lists strictly shorter than `xs`:

```
  labels (build xs)
=   {assume xs has length at least two
  and let (ys,zs) = halve xs}
  labels (Fork (build ys) (build zs))
=   {definition of labels}
  labels (build ys) ++ labels (build zs)
=   {induction, since ys and zs are strictly shorter than xs}
  ys ++ zs
=   {definition of halve xs}
  xs
```

The induction here is *general induction*: in order to prove $P(\texttt{xs})$ for all finite lists `xs` it is sufficient to prove that: (i) $P(\texttt{[]})$; and (ii) $P(\texttt{xs})$ holds under the assumption that P holds for all lists of length strictly less than `xs`.

Answer to Exercise J

One key property is that

```
(xs ++ [x] ++ ys)!!k | k < n   = xs!!k
                     | k==n    = x
                     | k > n   = ys!!(n-k)
                     where n = length xs
```

The other key property is that sorting a list does not change the length of the list. Hence

```
select k []     = error "list too short"
select k (x:xs) | k < n     = select k ys
                | k==n      = x
                | otherwise = select (n-k) zs
                where ys = [y | y <- xs, y < x]
                      zs = [z | z <- xs, x <= z]
                      n  = length ys
```

The worst-case running time for a list of length n occurs when $k = 0$ and the length of `ys` is $n-1$, i.e. when `x:xs` is in strictly decreasing order. Thus

$$T(\texttt{select})(0,n) = T(\texttt{select})(0,n-1) + \Theta(n),$$

with solution $T(\texttt{select})(0,n) = \Theta(n^2)$. But, assuming a reasonable distribution

in which each permutation of the sorted result is equally likely as input, we have $T(\texttt{select})(k,n) = \Theta(n)$.

7.10 Chapter notes

There are many books on algorithm design, but two that concentrate on functional programming are *Algorithms: A Functional Programming Approach* (second edition) (Addison-Wesley, 1999) by Fethi Rabbi and Guy Lapalme, and my own *Pearls of Functional Algorithm Design* (Cambridge, 2010).

Information about profiling tools comes with the documentation on the Haskell Platform. The source book on sorting is Don Knuth's *The Art of Computer Programming, Volume 3: Sorting and Searching* (second edition) (Addison-Wesley, 1998).