Chapter 6

Proofs

We have seen a lot of laws in the previous two chapters, though perhaps the word ‘law’ is a little inappropriate because it suggests something that is given to us from on high and which does not have to be proved. At least the word has the merit of being short. All of the laws we have encountered so far assert the equality of two functional expressions, possibly under subsidiary conditions; in other words, laws have been equations or *identities* between functions, and calculations have been point-free calculations (see Chapter 4, and the answer to Exercise K for more on the point-free style). Given suitable laws to work with, we can then use equational reasoning to prove other laws. Equational logic is a simple but powerful tool in functional programming because it can guide us to new and more efficient definitions of the functions and other values we have constructed. Efficiency is the subject of the following chapter. This one is about another aspect of equational reasoning, proof by induction. We will also show how to shorten proofs by introducing a number of *higher-order* functions that capture common patterns of computations. Instead of proving properties of similar functions over and over again, we can prove more general results about these higher-order functions, and appeal to them instead.

6.1 Induction over natural numbers

Consider the following definition of the exponential function:

```
exp :: Num a => a -> Nat -> a
exp x Zero     = 1
exp x (Succ n) = x * exp x n
```

In the old days we could have written

```
exp :: Num a => a -> Int -> a
exp x 0     = 1
exp x (n+1) = x * exp x n
```

but this precise form of definition using a `(n+1)`-pattern is no longer allowed in the current standard version of Haskell, Haskell 2010.

Anyway, we would expect that the equation

```
exp x (m+n) = exp x m * exp x n
```

is true for all `m` and `n`. After all, $x^{m+n} = x^m x^n$ is a true equation of mathematics. But how can we prove this law?

The answer, of course, is by *induction*. Every natural number is either `Zero` or of the form `Succ n` for some natural number `n`. That is exactly what the definition

```
data Nat = Zero | Succ Nat
```

of the data type `Nat` tells us. So to prove that $P(n)$ holds for all natural numbers n, we can prove

1. $P(0)$ holds;
2. For all natural numbers n, that $P(n+1)$ holds assuming that $P(n)$ does.

We have reverted to writing 0 for `Zero` and $n+1$ for `Succ n`, and we shall continue to do so. In the second proof we can assume $P(n)$ and use this assumption to prove $P(n+1)$.

As an example we prove that

```
exp x (m+n) = exp x m * exp x n
```

for all x, m and n by induction on m. We could also prove it by induction on n but that turns out to be more complicated. Here is the proof:

Case 0

```
  exp x (0 + n)                exp x 0 * exp x n
=   {since 0 + n = n}        =   {exp.1}
  exp x n                      1 * exp x n
                             =   {since 1 * x = x}
                               exp x n
```

Case `m+1`

```
    exp x ((m + 1) + n))
  =   {arithmetic}
    exp x ((m + n) + 1
  =   {exp.2}
    x * exp x (m + n)
  =   {induction}
    x * (exp x m * exp x n)
```

```
    exp x (m+1) * exp x n
  =   {exp.2}
    (x * exp x m) * exp x n
  =   {since * is associative}
    x * (exp x m * exp x n)
```

The above format will be used in all induction proofs. The proof breaks into two cases, the *base case* 0 and the *inductive case* $n+1$. Each case is laid out in two columns, one for the left-hand side of the equation, and one for the right-hand side. (When there is not enough space for two columns, we display one after the other.) Each side is simplified until one can go no further, and the proof of each case is completed by observing that each side simplifies to the same result. The hints `exp.1` and `exp.2` refer to the first and second equations defining `exp`.

Finally, observe that the proof depends on three further laws, namely that

```
(m + 1) + n = (m + n) + 1
1 * x       = x
(x * y) * z = x * (y * z)
```

If we were recreating all of arithmetic from scratch – and that would be a tedious thing to do – we would also have to prove these laws. In fact, only the first can be proved because it is entirely about natural numbers and we have defined the operation of addition on natural numbers. The second two rely on the implementation of multiplication prescribed by Haskell for the various instances of the type class `Num`.

In fact, the associative law breaks down for floating-point numbers:

```
ghci> (9.9e10 * 0.5e-10) * 0.1e-10 :: Float
4.95e-11
ghci> 9.9e10 * (0.5e-10 * 0.1e-10) :: Float
4.9499998e-11
```

Recall that in scientific notation `9.9e10` means `9.9 * 10^10`. So, although our proof was correct mathematically, one of the provisos in it wasn't, at least in Haskell.

6.2 Induction over lists

We have seen that every finite list is either the empty list `[]` or of the form `x:xs` where `xs` is a finite list. Hence, to prove that $P(xs)$ holds for all finite lists xs, we can prove:

1. $P($`[]`$)$ holds;

2. For all `x` and for all finite lists `xs`, that $P($`x:xs`$)$ holds assuming $P($`xs`$)$ does.

As an example, recall the definition of concatenation `(++)`:

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

We prove that ++ is associative:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

for all finite lists `xs` and for all lists `ys` and `zs` (note that neither of the last two is required to be a finite list), by induction on `xs`:

Case `[]`

```
  ([] ++ ys) ++ zs               [] ++ (ys ++ zs)
=   {++.1}                     =   {++.1}
  ys ++ zs                       ys ++ zs
```

Case `x:xs`

```
  ((x:xs) ++ ys) ++ zs           (x:xs) ++ (ys ++ zs)
=   {++.2}                     =   {++.2}
  (x:(xs ++ ys)) ++ zs           x:(xs ++ (ys ++ zs))
=   {++.2}                     =   {induction}
  x:((xs ++ ys) ++ zs)           x:((xs ++ ys) ++ zs)
```

As another example, given the definition

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

We prove that `reverse` is an involution:

```
reverse (reverse xs) = xs
```

for all finite lists `xs`. The base case is easy and the inductive case proceeds:

Case `x:xs`

```
   reverse (reverse (x:xs))
 =   {reverse.2}
   reverse (reverse xs ++ [x])
 =   {????}
   x:reverse (reverse xs)
 =   {induction}
   x:xs
```

The right-hand column is omitted in this example, since it consists solely of `x:xs`. But we got stuck in the proof halfway through. We need an auxiliary result, namely that

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

for all finite lists ys. This auxiliary result is also proved by induction:

Case `[]`

```
   reverse ([] ++ [x])                 x:reverse []
 =   {++.1}                          =   {reverse.1}
   reverse [x]                         [x]
 =   {reverse.2}
   reverse [] ++ [x]
 =   {reverse.1 and ++.1}
   [x]
```

Case `y:ys`

```
   reverse ((y:ys) ++ [x])           x:reverse (y:ys)
 =   {++.2}                        =   {reverse.2}
   reverse (y:(ys ++ [x]))           x:(reverse ys ++ [y])
 =   {reverse.2}
   reverse (ys ++ [x]) ++ [y]
 =   {induction}
   (x:reverse ys) ++ [y]
 =   {++.2}
   x:(reverse ys ++ [y])
```

The auxiliary result holds, and therefore so does the main result.

Induction over partial lists

Every partial list is either the undefined list or of the form `x:xs` for some `x` and some partial list `xs`. Hence, to prove that $P(xs)$ holds for all partial lists xs we can prove that

1. P(`undefined`) holds;
2. P(`x:xs`) holds assuming P(`xs`) does, for all `x` and all partial lists `xs`.

As an example, we prove that

```
xs ++ ys = xs
```

for all partial lists `xs` and all lists `ys`:

Case `undefined`

```
    undefined ++ ys
 =    {++.0}
    undefined
```

Case `x:xs`

```
    (x:xs) ++ ys
 =    {++.2}
    x:(xs ++ ys)
 =    {induction}
    x:xs
```

In each case the trivial right-hand column is omitted. The hint `(++).0` refers to the failing clause in the definition of `(++)`: since concatenation is defined by pattern matching on the left-hand argument, the result is undefined if the left-hand argument is.

Induction over infinite lists

Proving that something is true of all infinite lists requires a bit of background that we will elaborate on in a subsequent chapter. Basically an infinite list can be thought of as the limit of a sequence of partial lists. For example, `[0..]` is the limit of the sequence

```
undefined,  0:undefined,  0:1:undefined,  0:1:2:undefined,
```

and so on. A property P is called *chain complete* if whenever $xs_0, xs_1, \ldots$ is a sequence of partial lists with limit xs, and $P(xs_n)$ holds for all n, then $P(xs)$ also holds.

In other words, if P is a chain complete property that holds for all partial lists (and possibly all finite lists too), then it holds for all infinite lists.

Many properties are chain complete; for instance:

- All equations $e1 = e2$, where $e1$ and $e2$ are Haskell expressions involving universally quantified free variables, are chain complete.
- If P and Q are chain complete, then so is their conjunction $P \wedge Q$.

But inequalities $e1 \neq e2$ are not necessarily chain complete, and neither are properties involving existential quantification. For example, consider the assertion

```
drop n xs = undefined
```

for some integer `n`. This property is obviously true for all partial lists, and equally obviously not true for any infinite list.

Here is an example proof. Earlier we proved that

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

for all finite lists `xs` and for all lists `ys` and `zs`. We can extend this chain complete property to *all* lists `xs` by proving

Case `undefined`

```
   (undefined ++ ys) ++ zs          undefined ++ (ys ++ zs)
=    {++.0}                       =   {++.0}
   undefined ++ zs                  undefined
=    {++.0}
   undefined
```

Thus `++` is a truly associative operation on lists, independent of whether the lists are finite, partial or infinite.

But we have to be careful. Earlier we proved

```
reverse (reverse xs) = xs
```

for all finite lists xs. Can we extend this property to all lists by proving the following additional case?

Case undefined

```
    reverse (reverse undefined)
  =   {reverse.0}
    undefined
```

That goes through but something is wrong: as a Haskell equation we have

```
reverse (reverse xs) = undefined
```

for all partial lists xs. What did we miss?

The answer is that in proving the involution property of reverse we made use of an auxiliary result:

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

for all finite lists ys. This result is not true for all lists, indeed not true for any partial list ys.

It follows that reverse . reverse is not the identity function on lists, A functional equation f = g over lists asserts that f xs = g xs for *all* lists xs, finite, partial and infinite. If the equation is true only for finite lists, we have to say so explicitly.

6.3 The function foldr

All the following functions have a common pattern:

```
sum []     = 0
sum (x:xs) = x + sum xs

concat []       = []
concat (xs:xss) =  xs ++ concat xss

filter p []     = []
filter p (x:xs) = if p x then x:filter p xs
                  else filter p xs
```

```
map f []     = []
map f (x:xs) = f x:map f xs
```

Similarly, the proofs by induction of the following laws all have a common pattern:

```
sum (xs ++ ys)      = sum xs + sum ys
concat (xss ++ yss) = concat xss ++ concat yss
filter p (xs ++ ys) = filter p xs ++ filter p ys
map f (xs ++ ys)    = map f xs ++ map f ys
```

Can we not ensure that the functions above are defined as instances of a more general function, and the laws above as instances of a more general law? That would save a lot of repetitive effort.

The function `foldr` (fold from the right) is defined by

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

To appreciate this definition, consider

```
foldr (@) e [x,y,z] = x @ (y @ (z @ e))
            [x,y,z] = x : (y : (z : []))
```

In other words, `foldr (@) e` applied to a list replaces the empty list by `e`, and `(:)` by `(@)` and evaluates the result. The parentheses group from the right, whence the name.

It follows at once that `foldr (:) []` is the identity function on lists. Furthermore,

```
sum      = foldr (+) 0
concat   = foldr (++) []
filter p = foldr (\x xs -> if p x then x:xs else xs) []
map f    = foldr ((:) . f) []
```

The following fact captures all the identities mentioned above:

```
foldr f e (xs ++ ys) = foldr f e xs @ foldr f e ys
```

for some operation `(@)` satisfying various properties. We prove this equation by induction on `xs`. Along the way, we discover what properties of `f`, `e` and `(@)` we need.

Case `[]`

```
   foldr f e ([] ++ ys)          foldr f e [] @ foldr f e ys
 =   {++.1}                    =   {foldr.1}
   foldr f e ys                  e @ foldr f e ys
```

Hence we need `e @ x = x` for all `x`.

Case `x:xs`

```
   foldr f e ((x:xs) ++ ys)
 =   {++.2}
   foldr f e (x:(xs ++ ys)
 =   {foldr.2}
   f x (foldr f e (xs ++ ys))
 =   {induction}
   f x (foldr f e xs @ foldr f e ys)
```

The right-hand side in this case simplifies to

```
f x (foldr f e xs) @ foldr f e ys
```

So, in summary, we require that

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

for all `x`, `y` and `z`. In particular the two requirements are met if `f = (@)` and `(@)` is associative with identity `e`. That immediately proves

```
sum (xs ++ ys)      = sum xs + sum ys
concat (xss ++ yss) = concat xss ++ concat yss
```

For the `map` law, we require that

```
 [] ++ xs = xs
 f x:(xs ++ ys) = (f x:ys) ++ ys
```

Both immediately follow from the definition of concatenation.

For the law of `filter` we require that

```
if p x then x:(ys ++ zs) else ys ++ zs
  = (if p x then x:ys else ys) ++ zs
```

This is immediate from the definitions of concatenation and conditional expressions.

Fusion

The most important property of `foldr` is the *fusion law*, which asserts that

```
f . foldr g a =  foldr h b
```

provided certain properties of the ingredients hold. As two simple examples,

```
double . sum    = foldr ((+) . double) 0
length . concat = foldr ((+) . length) 0
```

In fact, many of the laws we have seen already are instances of the fusion law for `foldr`. In a word, the fusion law is a 'pre-packaged' form of induction over lists.

To find out what properties we need, we carry out an induction proof of the fusion law. The law is expressed as a functional equation, so we have to show that it holds for all finite and all partial lists:

Case `undefined`

```
   f (foldr g a undefined)          foldr h b undefined
=   {foldr.0}                    =   {foldr.0}
   f undefined                      undefined
```

So the first condition is that `f` is a strict function.

Case `[]`

```
   f (foldr g a [])                 foldr h b []
=   {foldr.1}                    =   {foldr.1}
   f a                              b
```

The second condition is that `f a = b`.

Case `x:xs`

```
   f (foldr g a (x:xs))             foldr h b (x:xs)
=   {foldr.2}                    =   {foldr.2}
   f (g x (foldr g a xs))           h x (foldr h b xs)
                                 =   {induction}
                                    h x (f (foldr g a xs))
```

The third condition is met by `f (g x y) = h x (f y)` for all `x` and `y`.

Let us apply the fusion law to show that

```
foldr f a . map g = foldr h a
```

Recall that `map g = foldr ((:) . g) []`. Looking at the conditions of the fusion law we have that

```
foldr f a undefined = undefined
foldr f a []        = a
```

So the first two fusion conditions are satisfied. The third one is

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

The left-hand side simplifies to

```
f (g x) (foldr f a xs)
```

so we can define `h x y = f (g x) y`. More briefly, `h = f . g`. Hence we have the useful rule:

```
foldr f a . map g = foldr (f . g) a
```

In particular,

```
double . sum = sum . map double
             = foldr ((+) . double) 0

length . concat = sum . map length
                = foldr ((+) . length) 0
```

Other simple consequences of the fusion law are explored in the exercises.

A variant

Sometimes having the empty list around is a pain. For example, what is the minimum element in an empty list? For this reason, Haskell provides a variant on `foldr`, called `foldr1`, restricted to nonempty lists. The Haskell definition of this function is

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

So we can define

```
minimum, maximum :: Ord a => [a] -> a
minimum = foldr1 min
maximum = foldr1 max
```

and avoid two other explicit recursions. Actually the Haskell definition of `foldr1` is not as general as it should be, but we will leave that discussion to an exercise.

6.4 The function `foldl`

Recall that

```
foldr (@) e [w,x,y,z] = w @ (x @ (y @ (z @ e)))
```

Sometimes a more convenient pattern for the right-hand side is

```
(((e @ w) @ x) @ y) @ z
```

This pattern is encapsulated by a function `foldl` (fold from the left):

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

As an example, suppose we are given a string, such as `1234.567`, representing a real number and we want to compute its integer part and fractional part. We could define

```
ipart :: String -> Integer
ipart xs = read (takeWhile (/= '.') xs) :: Integer

fpart :: String -> Float
fpart xs = read ('0':dropWhile (/= '.' xs) :: Float
```

This uses the function `read` of the type class `Read`. Note by the way that `.567` is not a well-formed literal in Haskell. It is necessary to include at least one digit before and after the decimal point to ensure that the decimal point cannot be mistaken for functional composition. For example,

```
ghci> :t 3 . 4
3 . 4 :: (Num (b -> c), Num (a -> b)) => a -> c
```

As an alternative, we can define

```
parts :: String -> (Integer,Float)
parts ds = (ipart es,fpart fs)
           where (es,d:fs) = break (== '.') ds
ipart    = foldl shiftl 0 . map toDigit
           where shiftl n d = n*10 + d
fpart    = foldr shiftr 0 . map toDigit
           where shiftr d x = (d + x)/10
toDigit d = fromIntegral (fromEnum d - fromEnum '0')
```

We have

```
1234  = 1*1000 + 2*100 + 3*10 + 4
      = (((0*10 + 1)*10 + 2)*10 + 3)*10 + 4
0.567 = 5/10 + 6/100 + 7/1000
      = (5 + (6 + (7 + 0)/10)/10)/10
```

so use of `foldl` for the integer part and `foldr` for the fractional part are both indicated.

Here is another example. The function `reverse` was defined above by the equations

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

We are wiser now and would now write

```
reverse = foldr snoc []
          where snoc x xs = xs ++ [x]
```

But a little learning is a dangerous thing: both definitions of `reverse` are terrible because they take of the order of n^2 steps to reverse a list of length n. Much better is to define

```
reverse = foldl (flip (:)) []
```

where `flip f x y = f y x`. The new version reverses a list in linear time:

```
  foldl (flip (:)) [] [1,2,3]
= foldl (flip (:)) (1:[]) [2,3]
= foldl (flip (:)) (2:1:[]) [3]
= foldl (flip (:)) (3:2:1:[]) []
= 3:2:1:[]
```

That seems a bit of a trick, but there is a sound principle at work behind this new definition that we will take up in the following chapter.

As this example suggests, there are the following relationships between `foldr` and `foldl`: for all finite lists `xs` we have

```
foldl f e xs = foldr (flip f) e (reverse xs)
foldr f e xs = foldl (flip f) e (reverse xs)
```

Proofs are left as an exercise. Note the restriction to finite lists, even though both sides reduce to $\perp$ when `xs` is $\perp$. That means the proofs have to rely on a subsidiary result that is true only for finite lists.

Here is another relationship between the two folds:

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

for all finite lists `xs`, provided that

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

Again, the proof is left as an exercise. As one instructive application of this law, suppose `(<>) = (@)` and `(@)` is associative with identity `e`. Then the two provisos are satisfied and we can conclude that

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

for all finite lists `xs` whenever `(@)` is associative with identity `e`. In particular,

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

for all finite lists `xss`. The two definitions are *not* the same if `xss` is an infinite list:

```
ghci> foldl (++) [] [[i] | i <- [1..]]
Interrupted.
ghci> foldr (++) [] [[i] | i <- [1..]]
[1,2,3,4,{Interrupted}
```

In response to the first expression, GHCi went into a long silence that was interrupted by pressing the ‘Stop program execution’ button. In response to the second, GHCi started printing an infinite list.

OK, so the definition in terms of `foldr` works on infinite lists, but the other one doesn’t. But maybe the definition of `concat` in terms of `foldl` leads to a more efficient computation when all the lists are finite? To answer this question, observe that

```
foldr (++) [] [xs,ys,us,vs]
           = xs ++ (ys ++ (us ++ (vs ++ [])))
```

```
    foldl (++) [] [xs,ys,us,vs]
              = (((([] ++ xs) ++ ys) ++ us) ++ vs)
```

Let all the component lists have length n. The first expression on the right takes $4n$ steps to perform all the concatenations, while the second takes $0+n+(n+n)+(n+n+n)=6n$ steps. Enough said, at least for now.

6.5 The function `scanl`

The function `scanl f e` applies `foldl f e` to each initial segment of a list. For example

```
ghci> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
```

The expression computes the *running sums* of the first ten positive numbers:

```
   [0, 0+1, (0+1)+2, ((0+1)+2)+3, (((0+1)+2)+3)+4, ...]
```

The specification of `scanl` is

```
    scanl :: (b -> a -> b) -> b -> [a] -> [b]
    scanl f e = map (foldl f e) . inits

    inits :: [a] -> [[a]]
    inits []     = [[]]
    inits (x:xs) = [] : map (x:) (inits xs)
```

For example

```
ghci> inits "barbara"
["","b","ba","bar","barb","barba","barbar","barbara"]
```

The function `inits` is in the library `Data.List`.

But this definition of `scanl f` involves evaluating `f` a total of

$$0+1+2+\cdots+n=n(n+1)/2$$

times on a list of length n. Can we do better?

Yes, we can calculate a better definition by doing a kind of induction proof, except that we don't know what it is we are proving!

Case `[]`

```
  scanl f e []
=   {definition}
  map (foldl f e) (inits [])
=   {inits.1}
  map (foldl f e) [[]]
=   {map.1 and map.2}
  [foldl f e []]
=   {foldl.1}
  [e]
```

Hence we have shown that `scanl f e [] = [e]`

Case `x:xs`

```
  scanl f e (x:xs)
=   {definition}
  map (foldl f e) (inits (x:xs))
=   {inits.2}
  map (foldl f e) ([]:map (x:) (inits xs))
=   {map.1 and map.2}
  foldl f e []:map (foldl f e . (x:)) (inits xs)
=   {foldl.1}
  e:map (foldl f e . (x:)) (inits xs)
=   {claim: foldl f e . (x:) = foldl f (f e x)}
  e:map (foldl f (f e x)) (inits xs)
=   {definition of scanl}
  e:scanl f (f e x)
```

The claim is an easy consequence of the definition of `foldl`. Hence, in summary, we have shown

```
scanl f e []     = [e]
scanl f e (x:xs) = e:scanl f (f e x) xs
```

This definition evaluates `f` only a linear number of times.

What we have just done is an example of optimising a function by *program calculation*. One of the exciting things about Haskell is that you can do this without fuss. There is no need to bring in a totally different logical language to reason about programs.

However, the prelude definition of `scanl` is a little different:

```
scanl f e xs = e : (case xs of
                     []   -> []
                     x:xs -> scanl f (f e x) xs)
```

Whereas for our version `scanl f e undefined = undefined`, the prelude version has

```
scanl f e undefined = e:undefined.
```

The reason is that the right-hand sides of the two clauses defining `scanl` are both lists that begin with `e`. We do not have to know anything about the left-hand sides to determine this fact, and laziness dictates that we don't ask.

The prelude version also uses a `case` expression. We won't go into details since such expressions are used rarely in this book. Haskell allows us many ways to say the same thing.

6.6 The maximum segment sum

Here is another example of program calculation. The *maximum segment sum* problem is a famous one and its history is described in J. Bentley's *Programming Pearls* (1987). Given is a sequence of integers and it is required to compute the maximum of the sums of all *segments* in the sequence. A segment is also called a *contiguous subsequence*. For example, the sequence

```
[-1,2,-3,5,-2,1,3,-2,-2,-3,6]
```

has maximum sum 7, the sum of the segment `[5,-2,1,3]`. On the other hand, the sequence `[-1,-2,-3]` has a maximum segment sum of zero, since the empty sequence is a segment of every list and its sum is zero. It follows that the maximum segment sum is always nonnegative.

Our problem is specified by

```
mss  :: [Int] -> Int
mss = maximum . map sum . segments
```

where `segments` returns a list of all segments of a list. This function can be defined in a number of ways, including

```
segments = concat . map inits . tails
```

where `tails` is dual to `inits` and returns all the tail segments of a list:

```
tails :: [a] -> [[a]]
tails []     = [[]]
tails (x:xs) = (x:xs):tails xs
```

The definition of `segments` describes the process of taking all the initial segments of all the tail segments. For example,

```
ghci> segments "abc"
["","a","ab","abc","","b","bc","","c",""]
```

The empty sequence appears four times in this list, once for every tail segment.

Direct evaluation of `mss` will take a number of steps proportional to n^3 on a list of length n. There are about n^2 segments, and summing each of them will take n steps, so in total it will take n^3 steps. It is not obvious that we can do better than cubic time for this problem.

However, let's see where some program calculation leads us. We can start by installing the definition of `segments`:

```
maximum . map sum . concat . map inits . tails
```

Searching for a law we can apply, we spot that

```
map f . concat = concat . map (map f)
```

applies to the subterm `map sum . concat`. That gives

```
maximum . concat . map (map sum) . map inits . tails
```

Now we can use the law `map f . map g = map (f . g)` to give

```
maximum . concat . map (map sum . inits) . tails
```

Oh, we can also use the law

```
maximum . concat = maximum . map maximum
```

can't we? No, not unless the argument to `concat` is a nonempty list of nonempty lists, because the maximum of the empty list is undefined. In the present example the rule is valid because both `inits` and `tails` return nonempty lists. That leads to

```
maximum . map (maximum . map sum . inits) . tails
```

The next step is to use the property of `scanl` described in the previous section, namely

```
map sum . inits =  scanl (+) 0
```

That leads to

```
maximum . map (maximum . scanl (+) 0) . tails
```

Already we have reduced a n^3 algorithm to a n^2 one, so we are making progress. But now we appear stuck since there is no law in our armoury that seems to help.

The next step obviously concerns `maximum . scanl (+) 0`. So, let's see what we can prove about

```
foldr1 max . scanl (+) 0
```

This looks like a fusion rule, but can `scanl (+) 0` be expressed as a `foldr`? Well, we do have, for instance,

```
  scanl (+) 0 [x,y,z]
= [0,0+x,(0+x)+y,((0+x)+y)+z]
= [0,x,x+y,x+y+z]
= 0:map (x+) [0,y,y+z]
= 0:map (x+) (scanl (+) 0 [y,z])
```

This little calculation exploits the associativity of `(+)` and the fact that `0` is the identity element of `(+)`. The result suggests, more generally, that

```
scanl (@) e = foldr f [e]
  where f x xs = e:map (x@) xs
```

provided that `(@)` is associative with identity `e`. Let us take this on trust and move on to the conditions under which

```
foldr1 (<>) . foldr f [e] = foldr h b
  where f x xs = e:map (x@) xs
```

It is immediate that `foldr1 (<>)` is strict and `foldr1 (<>) [e] = e`, so we have `b = e`. It remains to check the third proviso of the fusion rule: we require `h` to satisfy

```
foldr1 (<>) (e:map (x@) xs) = h x (foldr1 (<>) xs)
```

for all *x* and *xs*. The left-hand side simplifies to

```
    e <> (foldr1 (<>) (map (x@) xs))
```

Taking the singleton case `xs = [y]`, we find that

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

That gives us our definition of `h`, but we still have to check that

```
    foldr1 (<>) (e:map (x@) xs) = e <> (x @ foldr1 (<>) xs)
```

Simplifying both sides, this equation holds provided

```
    foldr1 (<>) . map (x@) = (x@) . foldr1 (<>)
```

This final equation holds provided `(@)` *distributes* over `(<>)`; that is

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

The proof is left as an exercise.

Does addition distribute over (binary) maximum? Yes:

```
    x + (y `max` z) = (x + y) `max` (x + z)
    x + (y `min` z) = (x + y) `min` (x + z)
```

Back to the maximum segment sum. We have arrived at

```
    maximum . map (foldr (@) 0) . tails
    where x @ y = 0 `max` (x + y)
```

What we have left looks very like an instance of the `scanl` rule of the previous section, except that we have a `foldr` not a `foldl` and a `tails` not an `inits`. But a similar calculation to the one about `scanl` reveals

```
   map (foldr f e) . tails = scanr f e
```

where

```
   scanr :: (a -> b -> b) -> b -> [a] -> [b]
   scanr f e []     = [e]
   scanr f e (x:xs) = f x (head ys):ys
                      where ys = scanr f e xs
```

The function `scanr` is also defined in the standard prelude. In summary,

```
    mss = maximum . scanr (@) 0
          where  x @ y = 0 `max` (x + y)
```

The result is a linear-time program for the maximum segment sum.

6.7 Exercises

Exercise A

In Chapter 3 we defined multiplication on natural numbers. The following definition is slightly different:

```
mult :: Nat -> Nat -> Nat
mult Zero y   = Zero
mult (Succ x) = mult x y + y
```

Prove that `mult (x+y) z = mult x z + mult y z`. You can use only the facts that `x+0 = x` and that `(+)` is associative. That means a long think about which variable `x`, `y` or `z` is the best one on which to do the induction.

Exercise B

Prove that

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

for all finite lists `xs` and `ys`. You may assume that `(++)` is associative.

Exercise C

Recall our friends Eager Beaver and Lazy Susan from Exercise D in Chapter 2. Susan happily used the expression `head . map f`, while Beaver would probably prefer `f . head`. Wait a moment! Are these two expressions equal? Carry out an induction proof to check.

Exercise D

Recall the cartesian product function `cp :: [[a]] -> [[a]]` from the previous chapter. Give a definition of the form `cp = foldr f e` for suitable `f` and `e`. You can use a list comprehension for the definition of `f` if you like.

The rest of this exercise concerns the proof of the identity

```
length . cp = product . map length
```

where `product` returns the result of multiplying a list of numbers.

1. Using the fusion theorem, express `length.cp` as an instance of `foldr`.
2. Express `map length` as an instance of `foldr`.

3. Using the fusion theorem again, express `product . map length` as an instance of `foldr`.
4. Check that the two results are identical. If they aren't, your definition of `cp` was wrong.

Exercise E

The first two arguments of `foldr` are replacements for the constructors

```
(:) :: a -> [a] -> [a]
[]  :: [a]
```

of lists. A fold function can be defined for any data type: just give replacements for the constructors of the data type. For example, consider

```
data Either a b = Left a | Right b
```

To define a fold for `Either` we have to give replacements for

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

That leads to

```
foldE :: (a -> c) -> (b -> c) -> Either a b -> c
foldE f g (Left x)  = f x
foldE f g (Right x) = g x
```

The type `Either` is not a recursive data type and `foldE` is not a recursive function. In fact `foldE` is a standard prelude function, except that it is called `either` not `foldE`.

Now define fold functions for

```
data Nat = Zero | Succ Nat
data NEList a = One a | Cons a (NEList a)
```

The second declaration introduces nonempty lists.

What is wrong with the Haskell definition of `foldr1`?

Exercise F

Prove that

```
foldl f e xs = foldr (flip f) e (reverse xs)
```

for all finite lists `xs`. Also prove that

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

for all finite lists `xs`, provided that

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

Exercise G

Using

```
foldl f e (xs ++ ys) = foldl f (foldl f e xs) ys
foldr f e (xs ++ ys) = foldr f (foldr f e ys) xs
```

prove that

```
foldl f e . concat = foldl (foldl f) e
foldr f e . concat = foldr (flip (foldr f)) e
```

Exercise H

Mathematically speaking, what is the value of

```
sum (scanl (/) 1 [1..])  ?
```

Exercise I

Calculate the efficient definition of `scanr` from the specification

```
scan r f e = map (foldr f e) . tails
```

Exercise J

Consider the problem of computing

```
mss :: [Int] -> Int
mss = maximum . map sum . subseqs
```

where `subseqs` returns all the subsequences of a finite list, including the list itself:

```
subseqs :: [a] -> [[a]]
subseqs []     = [[]]
subseqs (x:xs) = xss ++ map (x:) xss
                 where xss =  subseqs xs
```

Find a more efficient alternative for `mss`.

Exercise K

This question is in pieces.

1. The function `takePrefix p` applied to a list *xs* returns the longest initial segment of *xs* that satisfies *p*. Hence

   ```
   takePrefix :: ([a] -> Bool) -> [a] -> [a]
   ```

   What are the values of the following expressions?

   ```
   takePrefix nondec [1,3,7,6,8,9]
   takePrefix (all even) [2,4,7,8]
   ```

   Complete the right-hand side of

   ```
   takePrefix (all p) = ...
   ```

   Give a definition of `takePrefix` in terms of standard functions, including `inits`.

   We will return to `takePrefix` in the final part of this question.

2. The functions `one` and `none` are defined by the equations

   ```
   one x  = [x]
   none x = []
   ```

   Complete the right-hand side of the following identities:

   ```
   none . f     = ...
   map f . none = ...
   map f . one  = ...
   ```

3. Recall that `fork (f,g) x = (f x,g x)`. Complete the identities

   ```
   fst . fork (f,g) = ...
   snd . fork (f,g) = ...
   fork (f,g) . h   = ...
   ```

4. Define

   ```
   test p (f,g) x = if p x then f x else g x
   ```

   Complete the right-hand sides of

   ```
   test p (f,g) . h  = ...
   h . test p (f,g)  = ...
   ```

   The function `filter` can be defined by

```
filter p = concat . map (test p (one,none))
```

Using the identities above, together with other standard identities, prove using equational reasoning that

```
filter p = map fst . filter snd . map (fork (id,p))
```

(*Hint*: as always in calculations, start with the more complicated side.)

5. Recall the standard prelude functions `curry` and `uncurry` from the answer to Exercise K in Chapter 4:

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

Complete the right-hand side of

```
map (fork (f,g)) = uncurry zip . (??)
```

6. Returning to `takePrefix`, use equational reasoning to calculate an efficient program for the expression

```
takePrefix (p . foldl f e)
```

that requires only a linear number of applications of *f*.

6.8 Answers

Answer to Exercise A

The proof is by induction on *y*:

Case 0

```
   mult (x+0) z                   mult x z + mult 0 z
 =   {since x + 0=x}            =   {mult.1}
   mult x z                       mult x z + 0
                                =   {since x + 0 = x}
                                  mult x z
```

Case `y+1`

```
  mult (x+(y+1)) z                     mult x z + mult (y+1) z
=   {as (+) is associative}          =   {mult.2}
  mult ((x+y)+1) z                     mult x z + (mult y z + z)
=   {mult.2}                         =   {since (+) is associative}
  mult (x+y) z + z                     (mult x z + mult y z) + z
=   {induction}
  (mult x z + mult y z) + z
```

Answer to Exercise B

The proof is by induction on *xs*:

Case `[]`

```
  reverse ([]++ys)            reverse ys ++ reverse []
=   {++.1}                  =   {reverse.1}
  reverse ys                  reverse ys ++ []
                            =   {since xs ++ [] = xs}
                              reverse ys
```

Case `x:xs`

```
  reverse ((x:xs)++ys)
=   {++.2}
  reverse (x:(xs++ys))
=   {reverse.2}
  reverse (xs++ys) ++ [x]
=   {induction}
  (reverse ys ++ reverse xs) ++ [x]
```

and

```
  reverse ys ++ reverse (x:xs)
=   {reverse.2}
  reverse ys ++ (reverse xs ++ [x])
=   {since (++) is associative}
  (reverse ys ++ reverse xs) ++ [x]
```

Answer to Exercise C

We have to prove that

```
head (map f xs) = f (head xs)
```

for all lists *xs*, finite, partial or infinite. The case `undefined` and the inductive case `x:xs` are okay, but the case `[]` gives

```
head (map f []) = head [] = undefined
f (head [])     = f undefined
```

Hence the law holds only if `f` is a strict function. Eager Beaver is not bothered by this since he can only construct strict functions.

Answer to Exercise D

We have

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

1. `length . cp = foldr h b` provided `length` is strict (it is) and

   ```
   length [[]] = b
   length (op xs xss) = h xs (length xss)
   ```

   The first equation gives `b = 1` and as

   ```
   length (op xs xss) = length xs * length xss
   ```

   the second equation gives `h = (*) . length`.

2. `map length = foldr f []`, where `f xs ns = length xs:ns`. A shorter definition is `f = (:) . length`.

3. `product . map length = foldr h b` provided `product` is strict (it is) and

   ```
   product [] = b
   product (length xs:ns) = h xs (product ns)
   ```

   The first equation gives `b = 1`, and as

   ```
   product (length xs:ns) = length xs * product ns
   ```

   the second equation gives `h = (*) . length`.

4. The two definitions of `h` and `b` are identical.

Answer to Exercise E

The definition of `foldN` is straightforward:

```
foldN :: (a -> a) -> a -> Nat -> a
foldN f e Zero     = e
foldN f e (Succ n) = f (foldN f e n)
```

In particular,

```
m+n = foldN Succ m n
m*n = foldN (+m) Zero n
m^n = foldN (*m) (Succ Zero) n
```

For nonempty lists, the definition of `foldNE` is:

```
foldNE :: (a -> b -> b) -> (a -> b) -> NEList a -> b
foldNE f g (One x)     = g x
foldNE f g (Cons x xs) = f x (foldNE f g xs)
```

To be a proper fold over nonempty lists, the correct definition of `foldr1` should have been

```
foldr1 :: (a -> b -> b) -> (a -> b) -> [a] -> b
foldr1 f g [x]    = g x
foldr1 f g (x:xs) = f x (foldr1 f g xs)
```

The Haskell definition of `foldr1` restricts g to be the identity function.

Answer to Exercise F

Write `g = flip f` for brevity. We prove that

```
foldl f e xs = foldr g e (reverse xs)
```

for all finite lists `xs` by induction:

Case `[]`

```
   foldl f e []              foldl g e (reverse [])
=   {foldl.1}             =   {reverse.1}
   e                         foldl g e []
                          =   {foldl.1}
                             e
```

Case `x:xs`

```
  foldl f e (x:xs)
=   {foldl.2}
  foldl f (f e x) xs
=   {induction}
  foldr g (f e x) (reverse xs)
```

and

```
  foldr g e (reverse (x:xs))
=   {reverse.2}
  foldr g e (reverse xs ++ [x])
=   {claim: see below}
  foldr g (foldr g e [x]) (reverse xs)
=   {since foldr (flip f) e [x] = f e x}
  foldr g (f e x) (reverse xs)
```

The claim is that

```
foldr f e (xs ++ ys) = foldr f (foldr f e ys) xs
```

We leave the proof to the reader. By the way, we have the companion result that

```
foldl f e (xs ++ ys) = foldl f (foldl f e xs) ys
```

Again, the proof is left to you.

We prove

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

for all finite lists `xs` by induction. The base case is trivial. For the inductive case:

Case `x:xs`

```
  foldl (@) e (x:xs)                  foldr (<>) e (x:xs)
=   {foldl.2}                       =   {foldr.2}
  foldl (@) (e @ x) xs                x <> foldr (<>) e xs
=   {given that e @ x = x <> e}     =   {induction}
  foldl (@) (x <> e) xs               x <> foldl (@) e xs
```

The two sides have simplified to different results. We need another induction hypothesis:

```
    foldl (@) (x <> y) xs = x <> foldl (@) y xs
```

The base case is trivial. For the inductive case

Case `z:zs`

```
     foldl (@) (x <> y) (z:zs)
   =   {foldl.2}
     foldl (@) ((x <> y) @ z) zs
   =   {since (x <> y) @ z = x <> (y @ z)}
     foldl (@) (x <> (y @ z)) zs
   =   {induction}
     x <> foldl (@) (y @ z) zs
```

and

```
     x <> foldl (@) y (z:zs)
   =   {foldl.2}
     x <> foldl (@) (y @ z) zs
```

Answer to Exercise G

The proofs are by induction. The base cases are easy and the inductive cases are

```
     foldl f e (concat (xs:xss))
   =   {definition of concat}
     foldl f e (xs ++ concat xss)
   =   {given property of foldl}
     foldl f (foldl f e xs) (concat xss)
   =   {induction}
     foldl (foldl f) (foldl f e xs) xss
   =   {definition of foldl}
     foldl (foldl f) e (xs:xss)
```

and

```
      foldr f e (concat (xs:xss))
    =   {definition of concat}
      foldr f e (xs ++ concat xss)
    =   {given property of foldr}
      foldr f (foldr f e (concat xss)) xs
    =   {using flip}
      flip (foldr f) xs (foldr f e (concat xss))
    =   {induction}
      flip (foldr f) xs (foldr (flip (foldr f)) e xss)
    =   {definition of foldr}
      foldr (flip (foldr f)) e (xs:xss)
```

Answer to Exercise H

Mathematically speaking,

```
    sum (scanl (/) 1 [1..]) = e
```

since $\sum_{n=0}^{\infty} 1/n! = e$. Computationally speaking, replacing `[1..]` by a finite list `[1..n]` gives an approximation to e. For example,

```
ghci> sum (scanl (/) 1 [1..20])
2.7182818284590455
ghci> exp 1
2.718281828459045
```

The standard prelude function `exp` takes a number x and returns e^x. By the way, the prelude function `log` takes a number x and returns $\log_e x$. If you want logarithms in another base, use `logBase` whose type is

```
    logBase :: Floating a => a -> a -> a
```

Answer to Exercise I

We synthesise a more efficient definition by cases. The base case yields

```
    scanr f e [] = [e]
```

and the inductive case `x:xs` is:

```
   scanr f e (x:xs)
=    {specification}
   map (foldr f e) (tails (x:xs))
=    {tails.2}
   map (foldr f e) ((x:xs):tails xs)
=    {definition of map}
   foldr f e (x:xs):map (foldr f e) (tails xs)
=    {foldr.2 and specification}
   f x (foldr f e xs):scan f e xs
=    {claim: foldr f e xs = head (scanr f e xs)}
   f x (head ys):ys where ys = scanr f e xs
```

Answer to Exercise J

Firstly,

```
subseqs = foldr op [[]]
 where op x xss = xss ++ map (x:) xss
```

Appeal to the fusion law yields

```
map sum . subseqs = foldr op [0]
 where op x xs = xs ++ map (x+) xs
```

A second appeal to fusion yields

```
maximum . map sum . subseqs = foldr op 0
 where op x y = y `max` (x+y)
```

That will do nicely. Of course, `sum . filter (>0)` also does the job.

Answer to Exercise K

1. We have

   ```
   takePrefix nondec [1,3,7,6,8,9] = [1,3,7]
   takePrefix (all even) [2,4,7,8] = [2,4]
   ```

   The identity is

   ```
   takePrefix (all p) = takeWhile p
   ```

The specification is

```
takePrefix p = last . filter p . inits
```

2. We have

```
none . f     = none
map f . none = none
map f . one  = one . f
```

3. We have

```
fst . fork (f,g) = f
snd . fork (f,g) = g
fork (f,g) . h   = fork (f.h,g.h)
```

4. We have

```
test p (f,g) . h = test (p.h) (f . h, g . h)
h . test p (f,g) = test p (h . f, h . g)
```

The reasoning is:

```
  map fst . filter snd . map (fork (id,p))
=   {definition of filter}
  map fst . concat . map (test snd (one,none)) .
  map (fork (id,p))
=   {since map f . concat = concat . map (map f)}
  concat . map (map fst . test snd (one,none) .
  fork (id,p))
=   {second law of test; laws of one and none}
  concat . map (test snd (one . fst,none) .
  fork (id,p))
=   {first law of test; laws of fork}
  concat . map (test p (one . id, none . fork (id,p)))
=   {laws of id and none}
  concat . map (test p (one,none))
=   {definition of filter}
  filter p
```

5. We have

```
map (fork (f,g))  = uncurry zip . fork (map f,map g)
```

6. We have

```
   filter (p . foldl f e) . inits
=  {derived law of filter}
   map fst . filter snd .
   map (fork (id, p . foldl f e)) . inits
=  {law of zip}
   map fst . filter snd . uncurry zip .
   fork (id, map (p . foldl f e)) . inits
=  {law of fork}
   map fst . filter snd . uncurry zip .
   fork (inits, map (p . foldl f e) . inits)
=  {scan lemma}
   map fst . filter snd . uncurry zip .
   fork (inits, map p . scanl f e)
```

Hence

```
takePrefix (p.foldl f e)
   =  fst . last . filter snd . uncurry zip .
      fork (inits,map p . scanl f e)
```

6.9 Chapter notes

Gofer, an earlier version of Haskell designed by Mark Jones, was so named because it was GOod For Equational Reasoning. HUGS (The Haskell Users Gofer System) was an earlier alternative to GHCi, and used in the second edition of the book on which the current one is based, but is no longer maintained.

Many people have contributed to the understanding of the laws of functional programming, too many to list. The Haskellwiki page

```
haskell.org/haskellwiki/Equational_reasoning_examples
```

contains examples of equational reasoning and links to various discussions about the subject.

The fascinating history of the maximum segment sum problem is discussed in Jon Bentley's *Programming Pearls* (second edition) (Addison-Wesley, 2000).