chapter 13

Reasoning about programs

In this final chapter we introduce the idea of reasoning about Haskell programs. We start by reviewing the notion of equational reasoning, then consider how it can be applied in Haskell, introduce the important technique of induction, show how induction can be used to eliminate uses of the append operator, and conclude by proving the correctness of a simple compiler.

13.1 Equational reasoning

At school we learn basic algebraic properties of numbers, such as the fact that multiplication is commutative, addition is associative, and multiplication distributes over addition on both the left- and right-hand sides:

$$\begin{array}{lcl} x\,y & = & y\,x \\ x + (y + z) & = & (x + y) + z \\ x\,(y + z) & = & x\,y + x\,z \\ (x + y)\,z & = & x\,z + y\,z \end{array}$$

For example, using these properties we can show that a product of the form $(x + a)\,(x + b)$ can be expanded to a summation $x^2 + (a + b)\,x + a\,b$:

$$\begin{array}{ll} & (x + a)\,(x + b) \\ = & \quad \{ \text{ left distributivity } \} \\ & (x + a)\,x + (x + a)\,b \\ = & \quad \{ \text{ right distributivity } \} \\ & x\,x + a\,x + x\,b + a\,b \\ = & \quad \{ \text{ squaring } \} \\ & x^2 + a\,x + x\,b + a\,b \\ = & \quad \{ \text{ commutativity } \} \\ & x^2 + a\,x + b\,x + a\,b \\ = & \quad \{ \text{ right distributivity } \} \\ & x^2 + (a + b)\,x + a\,b \end{array}$$

Note that in this calculation we follow the common practice of implicitly exploiting associativity properties, in this case the associativity of addition by omitting parentheses when more than one addition is used in sequence.

As well as being interesting in their their own right, algebraic properties can also have a computational significance. For example, the expression $x\,(y+z)$ requires two operations (one multiplication and one addition), whereas the equivalent expression $x\,y + x\,z$ requires three (two multiplications and one addition). Hence even though these two expressions are algebraically equal, in terms of efficiency the former is preferable to the latter.

13.2 Reasoning about Haskell

The same style of equational reasoning can also be used in Haskell. For example, in this context the equation $x * y = y * x$ means that for any expressions x and y of the same numeric types, evaluation of $x * y$ and $y * x$ will always produce the same numeric value. Note that the equality operator $==$ provided within Haskell itself is not used when stating such properties, as in $x * y == y * x$, because we are aiming to use mathematics as a language to reason about Haskell, rather than using Haskell as a language to reason about itself, which would be somewhat circular.

When reasoning about Haskell, we do not just use properties of built-in operations of the language such as addition and multiplication, but also use the equations from which user-defined functions are constructed. For example, consider the following function that doubles an integer:

$$
\begin{array}{lcl}
\mathit{double} & :: & \mathit{Int} \to \mathit{Int} \\
\mathit{double}\ x & = & x + x
\end{array}
$$

As well as being viewed as the definition of a function, this equation can also be viewed as a property that can be used when reasoning about this function. In particular, as a logical property the above equation states that for any integer expression x, the expression $\mathit{double}\ x$ can freely be replaced by $x + x$, and, conversely, that the expression $x + x$ can freely be replaced by $\mathit{double}\ x$. In this manner, when reasoning about programs, function definitions can be both applied from left-to-right and unapplied from right-to-left.

However, some care is required when reasoning about functions that are defined using multiple equations. For example, consider a function that decides if an integer is zero, defined using two equations:

$$
\begin{array}{lcl}
\mathit{isZero} & :: & \mathit{Int} \to \mathit{Bool} \\
\mathit{isZero}\ 0 & = & \mathit{True} \\
\mathit{isZero}\ n & = & \mathit{False}
\end{array}
$$

The first equation, $\mathit{isZero}\ 0 = \mathit{True}$, can freely be viewed as a logical property that can be applied in both directions. However, this is not the case for the second equation, $\mathit{isZero}\ n = \mathit{False}$. In particular, because the order in which the equations are written is significant, an expression of the form $\mathit{isZero}\ n$ can only be replaced by False provided that $n \neq 0$, as in the case when $n = 0$ the first equation applies. Dually, it is only valid to unapply the equation

$isZero\ n = False$ and replace $False$ by an expression of the form $isZero\ n$ in the case when $n \neq 0$, for the same reason.

More generally, when a function is defined using multiple equations, the equations cannot be viewed as logical properties in isolation from one another, but need to be interpreted in light of the order in which patterns are matched within the equations. For this reason, it is preferable to define functions in a manner that does not rely on the order in which their equations are written. For example, if we rewrite the above definition using a guard

$$\begin{array}{lcl} isZero\ 0 & = & True \\ isZero\ n \mid n \neq 0 & = & False \end{array}$$

then it is now explicitly clear that $isZero\ n$ can only be replaced by $False$, and conversely that $False$ can only be replaced by $isZero\ n$, when the guard $n \neq 0$ is satisfied. Patterns that do not rely on the order in which they are matched are called *disjoint* or *non-overlapping*. In order to simplify the process of reasoning about programs, it is good practice to use non-overlapping patterns whenever possible. For example, most of the functions in the standard library given in appendix A are defined in this manner.

13.3 Simple examples

As a simple example of equational reasoning in Haskell, recall the following definition of the library function that reverses a list:

$$\begin{array}{lcl} reverse & :: & [a] \rightarrow [a] \\ reverse\ [\,] & = & [\,] \\ reverse\ (x : xs) & = & reverse\ xs +\!\!+ [x] \end{array}$$

Using this definition, we can show that $reverse$ has no effect on singleton lists, in the sense that $reverse\ [x] = [x]$ for any element x:

$$\begin{array}{ll} & reverse\ [x] \\ = & \quad \{ \text{list notation} \} \\ & reverse\ (x : [\,]) \\ = & \quad \{ \text{applying } reverse \} \\ & reverse\ [\,] +\!\!+ [x] \\ = & \quad \{ \text{applying } reverse \} \\ & [\,] +\!\!+ [x] \\ = & \quad \{ \text{applying } +\!\!+ \} \\ & [x] \end{array}$$

Hence any expression of the form $reverse\ [x]$ in a program can freely be replaced by $[x]$ without change in meaning, but with a change in efficiency by avoiding the need to apply the $reverse$ function.

Equational reasoning is often combined with some form of case analysis. For example, consider the logical negation function:

$$\begin{array}{lcl} \neg & :: & Bool \rightarrow Bool \\ \neg\ False & = & True \\ \neg\ True & = & False \end{array}$$

Because this function is defined by pattern matching, properties of $\neg$ are normally proved by case analysis on its argument. For example, the fact that $\neg$ is its own inverse, $\neg\ (\neg\ b) = b$ for all logical values b, can be shown by case analysis on the two possible values for b. For example, the case when $b = \mathit{False}$ is verified below, and $b = \mathit{True}$ follows similarly:

$$
\begin{array}{ll}
& \neg\ (\neg\ \mathit{False}) \\
= & \quad \{\text{ applying the inner } \neg\ \} \\
& \neg\ \mathit{True} \\
= & \quad \{\text{ applying } \neg\ \} \\
& \mathit{False}
\end{array}
$$

13.4 Induction on numbers

Most interesting functional programs involve some form of recursion. Reasoning about such programs normally proceeds using the simple but powerful technique of *induction*. Let us begin by recalling the simplest example of a recursive type, namely the type of natural numbers:

$$\mathbf{data}\ \mathit{Nat} \ = \ \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

This declaration states that Zero is a value of type Nat (the base case), and that if n is a value of type Nat, then so is $\mathit{Succ}\ n$ (the recursive case). Implicit in the declaration is the fact that Zero and Succ are the only constructors for the type Nat. Hence, the values of Nat can be enumerated as follows:

$$
\begin{array}{l}
\mathit{Zero} \\
\mathit{Succ}\ \mathit{Zero} \\
\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}) \\
\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})) \\
\vdots
\end{array}
$$

For simplicity, we only consider the finite natural numbers, obtained by starting with Zero and applying Succ a finite number of times. In particular, we do not consider infinity, defined by $\mathit{inf} = \mathit{Succ}\ \mathit{inf}$. A similar comment applies to all the other recursive types that we consider in this chapter.

Now suppose we want to prove that some property, p say, holds for all (finite) natural numbers. Then the principle of induction states that it is sufficient to show that p holds for Zero, called the *base case*, and that p is preserved by Succ, called the *inductive case*. More precisely, in the inductive case one is required to show that if the property p holds for any natural number n, called the *induction hypothesis*, then it also holds for $\mathit{Succ}\ n$.

Why is induction sufficient to show that p holds for all natural numbers? For example, how does it then follow that p holds for $\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})$. Starting from the base case that p holds for Zero, we can apply the inductive case once to conclude that p holds for $\mathit{Succ}\ \mathit{Zero}$, by taking $n = \mathit{Zero}$, and then apply the inductive case a second time to conclude that p holds for $\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})$, by taking $n = \mathit{Succ}\ \mathit{Zero}$. In a similar manner, it can be established that p holds for any natural number.

It is useful to draw an analogy with the "domino effect". Suppose there is a line of dominoes standing on end and you know that the first domino will fall, and that whenever a domino falls then its next neighbour will also fall. Then it is clear that all the dominoes will fall, by applying the first fact to get the process started, and repeatedly applying the second to keep it going. The same pattern of reasoning occurs with induction: we first verify the required property for $Zero$ (the first domino falls), then that the property is preserved by $Succ$ (if any domino falls, then so will its neighbour), and conclude that the property holds for all natural numbers (all dominoes fall).

As a concrete example, consider the definition of a recursive function that takes two natural numbers and adds them together:

$$\begin{array}{lcl} add & :: & Nat \rightarrow Nat \rightarrow Nat \\ add\ Zero\ m & = & m \\ add\ (Succ\ n)\ m & = & Succ\ (add\ n\ m) \end{array}$$

From the first equation it is immediate that $add\ Zero\ m = m$ holds for any natural number m. Now let us show that the dual property, $add\ n\ Zero = n$, which we abbreviate by p, also holds for all natural numbers n. We proceed by induction on n. The base case, showing that $p\ Zero$ holds, amounts to showing that $add\ Zero\ Zero = Zero$, which is immediate:

$$\begin{array}{ll} & add\ Zero\ Zero \\ = & \quad \{\text{ applying } add \text{ }\} \\ & Zero \end{array}$$

For the inductive case, we must show that if p holds for any natural number n, then $p\ (Succ\ n)$ also holds. That is, using the induction hypothesis $add\ n\ Zero = n$ as an assumption, we must show that $add\ (Succ\ n)\ Zero = Succ\ n$, which can be verified by the following calculation:

$$\begin{array}{ll} & add\ (Succ\ n)\ Zero \\ = & \quad \{\text{ applying } add \text{ }\} \\ & Succ\ (add\ n\ Zero) \\ = & \quad \{\text{ induction hypothesis }\} \\ & Succ\ n \end{array}$$

□

Because proofs by induction normally involve more than one calculation, it is useful to explicitly indicate the end of the proof. For this purpose, we use a square box □ in the right-hand margin, as illustrated above.

As another example, let us now show that addition of natural numbers is associative. That is, $add\ x\ (add\ y\ z) = add\ (add\ x\ y)\ z$ for all natural numbers x, y and z. There are three arguments to choose from, so which should induction be performed over? Note that the add function is defined by pattern matching on its first argument, so it is natural to try induction on x, which appears twice as the first argument to add in the associativity equation, whereas y only appears once as such and z never. Using induction on x, the proof of the associativity of add proceeds as follows.

Base case:

$$
\begin{array}{ll}
& add\ Zero\ (add\ y\ z) \\
= & \quad \{\text{ applying the outer } add \text{ }\} \\
& add\ y\ z \\
= & \quad \{\text{ unapplying } add \text{ }\} \\
& add\ (add\ Zero\ y)\ z
\end{array}
$$

Inductive case:

$$
\begin{array}{ll}
& add\ (Succ\ x)\ (add\ y\ z) \\
= & \quad \{\text{ applying the outer } add \text{ }\} \\
& Succ\ (add\ x\ (add\ y\ z)) \\
= & \quad \{\text{ induction hypothesis }\} \\
& Succ\ (add\ (add\ x\ y)\ z) \\
= & \quad \{\text{ unapplying the outer } add \text{ }\} \\
& add\ (Succ\ (add\ x\ y)\ z) \\
= & \quad \{\text{ unapplying the inner } add \text{ }\} \\
& add\ (add\ (Succ\ x)\ y)\ z
\end{array}
$$

□

Note that both cases start by applying definitions, and conclude by unapplying definitions. This pattern is typical in proofs by induction, but the latter part may seem somewhat mysterious at first. In particular, knowing which definitions to unapply seems to require a degree of foresight. In practice, however, if one becomes stuck at a certain point during such a calculation, progress can often be made by focusing on the desired end result and trying to work backwards to the point where one became stuck.

For example, after applying the induction hypothesis in the inductive case above to obtain $Succ\ (add\ (add\ x\ y)\ z)$, it may not be clear how to proceed, as there are no more definitions that can be applied. However, if we then focus on the expression that we are aiming towards, $add\ (add\ (Succ\ x)\ y)\ z$, we can simply apply the inner add and then the outer add to produce the expression at which we became stuck, which process can then be reversed (turning applying into unapplying) to complete the calculation.

Although we have introduced induction using the recursive type Nat, the same principle can also be used with the integers that are built-in to Haskell. In particular, to prove that some property p holds for all integers $n \geq 0$, it is sufficient to show that p holds for 0, the base case, and that if p holds for any $n \geq 0$, then it also holds for $n + 1$, the inductive case.

For example, consider the following definition for the library function $replicate$ that produces a list with n identical elements:

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

Then it is easy to show that this function does indeed produce a list with n elements, that is $length\ (replicate\ n\ x) = n$, by induction on $n \geq 0$.

Base case:

$$
\begin{aligned}
& \mathit{length}\ (\mathit{replicate}\ 0\ x) \\
= & \quad \{ \text{ applying } \mathit{replicate} \ \} \\
& \mathit{length}\ [\,] \\
= & \quad \{ \text{ applying } \mathit{length} \ \} \\
& 0
\end{aligned}
$$

Inductive case:

$$
\begin{aligned}
& \mathit{length}\ (\mathit{replicate}\ (n+1)\ x) \\
= & \quad \{ \text{ applying } \mathit{replicate} \ \} \\
& \mathit{length}\ (x : \mathit{replicate}\ n\ x) \\
= & \quad \{ \text{ applying } \mathit{length} \ \} \\
& 1 + \mathit{length}\ (\mathit{replicate}\ n\ x) \\
= & \quad \{ \text{ induction hypothesis } \} \\
& 1 + n \\
= & \quad \{ \text{ commutativity of } + \ \} \\
& n + 1
\end{aligned}
$$

□

13.5 Induction on lists

Induction is not restricted to natural numbers, but can also be used to reason about other recursive types, such as the type of lists. Just as natural numbers are built up recursively from zero by applying the successor function, so lists are built up from the empty list by applying the cons operator.

Suppose we want to prove that some property p holds for all lists. Then the induction principle for lists states that it is sufficient to show that p holds for the empty list [ ], the base case, and that if p holds for any list xs, then it also holds for $x : xs$ for any element x, the inductive case. Of course, both the element x and the list xs must be of the appropriate types.

As an example, let us show that the function $\mathit{reverse}$ defined earlier in this chapter is its own inverse, $\mathit{reverse}\ (\mathit{reverse}\ xs) = xs$, by induction on xs. The base case is verified simply by applying the definition of $\mathit{reverse}$:

$$
\begin{aligned}
& \mathit{reverse}\ (\mathit{reverse}\ [\,]) \\
= & \quad \{ \text{ applying the inner } \mathit{reverse} \ \} \\
& \mathit{reverse}\ [\,] \\
= & \quad \{ \text{ applying } \mathit{reverse} \ \} \\
& [\,]
\end{aligned}
$$

For the inductive case, using the assumption $\mathit{reverse}\ (\mathit{reverse}\ xs) = xs$, we show that $\mathit{reverse}\ (\mathit{reverse}\ (x : xs)) = x : xs$, as follows:

$$
\begin{aligned}
& \mathit{reverse}\ (\mathit{reverse}\ (x : xs)) \\
= & \quad \{ \text{ applying the inner } \mathit{reverse} \ \} \\
& \mathit{reverse}\ (\mathit{reverse}\ xs +\!\!+ [x]) \\
= & \quad \{ \text{ distributivity - see below } \} \\
& \mathit{reverse}\ [x] +\!\!+ \mathit{reverse}\ (\mathit{reverse}\ xs)
\end{aligned}
$$

$$
\begin{array}{ll}
= & \quad \{\text{ singleton lists - see below }\} \\
 & [x] +\!\!+ \mathit{reverse}\ (\mathit{reverse}\ xs) \\
= & \quad \{\text{ induction hypothesis }\} \\
 & [x] +\!\!+ ys \\
= & \quad \{\text{ applying } +\!\!+ \ \} \\
 & x : xs
\end{array}
$$

□

This calculation uses two auxiliary properties of the function $\mathit{reverse}$, namely our earlier result that $\mathit{reverse}$ preserves singleton lists, $\mathit{reverse}\ [x] = [x]$, together with a new result that $\mathit{reverse}$ distributes over append, except that the order of the two argument lists is then swapped:

$$\mathit{reverse}\ (xs +\!\!+ ys) \quad = \quad \mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ xs$$

Technically, we say that the distribution is contravariant. Because the append operator $+\!\!+$ is defined by pattern matching on its first argument, it is natural to attempt to verify this property by induction on xs.

Base case:

$$
\begin{array}{ll}
 & \mathit{reverse}\ ([\,] +\!\!+ ys) \\
= & \quad \{\text{ applying } +\!\!+ \ \} \\
 & \mathit{reverse}\ ys \\
= & \quad \{\text{ unapplying } +\!\!+ \ \} \\
 & [\,] +\!\!+ \mathit{reverse}\ ys \\
= & \quad \{\text{ unapplying } \mathit{reverse}\ \} \\
 & \mathit{reverse}\ [\,] +\!\!+ \mathit{reverse}\ ys
\end{array}
$$

Inductive case:

$$
\begin{array}{ll}
 & \mathit{reverse}\ ((x : xs) +\!\!+ ys) \\
= & \quad \{\text{ applying } +\!\!+ \ \} \\
 & \mathit{reverse}\ (x : (xs +\!\!+ ys)) \\
= & \quad \{\text{ applying } \mathit{reverse}\ \} \\
 & \mathit{reverse}\ (xs +\!\!+ ys) +\!\!+ [x] \\
= & \quad \{\text{ induction hypothesis }\} \\
 & (\mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ xs) +\!\!+ [x] \\
= & \quad \{\text{ associativity of } +\!\!+ \ \} \\
 & \mathit{reverse}\ ys +\!\!+ (\mathit{reverse}\ xs +\!\!+ [x]) \\
= & \quad \{\text{ unapplying the second } \mathit{reverse}\ \} \\
 & \mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ (x : xs)
\end{array}
$$

□

This calculation in turn uses the fact that $+\!\!+$ is associative, which can be verified by induction in a similar manner to our earlier result that add is associative (see exercise 5 for this chapter.)

13.6 Making append vanish

Many recursive functions are naturally defined using the append operator $+\!\!+$ on lists, but this operator carries a considerable efficiency cost when used recursively. In this section, we show how induction can be used to eliminate such uses of append, and hence make functions more efficient. As a first example, consider again the following definition:

$$
\begin{array}{lcl}
\mathit{reverse} & :: & [a] \to [a] \\
\mathit{reverse}\ [\,] & = & [\,] \\
\mathit{reverse}\ (x : \mathit{xs}) & = & \mathit{reverse}\ \mathit{xs} +\!\!+ [x]
\end{array}
$$

How efficient is $\mathit{reverse}$? First of all, it is easy to show that the number of reduction steps required to evaluate $\mathit{xs} +\!\!+ \mathit{ys}$ is one greater than the length of xs, assuming for simplicity that both xs and ys are already fully evaluated. As a result, we say that $+\!\!+$ takes linear time in the length of its first argument. In turn, the number of steps required by $\mathit{reverse}\ \mathit{xs}$ for a list of length n can be shown to be the sum of the integers from 1 to $n + 1$, which is $(n + 1)(n + 2)/2$. Multiplying out the brackets using the equation verified at the start of this chapter gives $(n^2 + 3n + 2)/2$, as a result of which we say that $\mathit{reverse}$ takes quadratic time in the length of its argument.

Quadratic time is bad. For example, reversing a list with ten thousand elements will take approximately fifty million reduction steps. Fortunately, however, through the use of induction it is easy to eliminate the use of append in the definition of $\mathit{reverse}$, and hence improve its efficiency.

The trick is to attempt to define a more general function, which combines the behaviours of $\mathit{reverse}$ and $+\!\!+$. In particular, we seek to define a recursive function $\mathit{reverse}'$ that satisfies the following equation:

$$
\mathit{reverse}'\ \mathit{xs}\ \mathit{ys} \quad = \quad \mathit{reverse}\ \mathit{xs} +\!\!+ \mathit{ys}
$$

That is, applying $\mathit{reverse}'$ to two lists should give the result of reversing the first list, appended together with the second list. If we can define such a function, then $\mathit{reverse}$ itself can be redefined by $\mathit{reverse}\ \mathit{xs} = \mathit{reverse}'\ \mathit{xs}\ [\,]$, using the fact that the empty list is the identity for append.

Rather the giving the definition for $\mathit{reverse}'$, and then showing that it satisfies the above equation, we can in fact use this equation as the driving force for constructing the definition itself. In particular, we simply attempt to verify this equation by induction on xs. The base case results in an equation that gives the definition for $\mathit{reverse}'\ [\,]\ \mathit{ys}$, while the inductive case results in an equation that gives the definition for $\mathit{reverse}'\ (x : \mathit{xs})\ \mathit{ys}$.

Base case:

$$
\begin{array}{ll}
 & \mathit{reverse}'\ [\,]\ \mathit{ys} \\
= & \quad \{ \text{ specification of } \mathit{reverse}' \ \} \\
 & \mathit{reverse}\ [\,] +\!\!+ \mathit{ys} \\
= & \quad \{ \text{ applying } \mathit{reverse} \ \} \\
 & [\,] +\!\!+ \mathit{ys} \\
= & \quad \{ \text{ applying } +\!\!+ \ \} \\
 & \mathit{ys}
\end{array}
$$

Inductive case:

$$
\begin{array}{ll}
 & \mathit{reverse}'\ (x : \mathit{xs})\ \mathit{ys} \\
= & \quad \{ \text{ specification of } \mathit{reverse}' \ \}
\end{array}
$$

$$
\begin{array}{ll}
 & reverse\ (x : xs) +\!\!+\ ys \\
= & \quad \{\ \text{applying } reverse\ \} \\
 & (reverse\ xs +\!\!+ [x]) +\!\!+\ ys \\
= & \quad \{\ \text{associativity of } +\!\!+\ \} \\
 & reverse\ xs +\!\!+ ([x] +\!\!+\ ys) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & reverse'\ xs\ ([x] +\!\!+\ ys) \\
= & \quad \{\ \text{applying } +\!\!+\ \} \\
 & reverse'\ xs\ (x : ys)
\end{array}
$$

□

We conclude from this proof that the definition

$$
\begin{array}{lcl}
reverse' & :: & [a] \rightarrow [a] \rightarrow [a] \\
reverse'\ [\,]\ ys & = & ys \\
reverse'\ (x : xs)\ ys & = & reverse'\ xs\ (x : ys)
\end{array}
$$

suffices to show that $reverse'\ xs\ ys = reverse\ xs +\!\!+\ ys$ by induction. Note that the definition for $reverse'$ does not refer to the original $reverse$ function, or append. Hence, $reverse$ itself can now be redefined as follows:

$$
\begin{array}{lcl}
reverse & :: & [a] \rightarrow [a] \\
reverse\ xs & = & reverse'\ xs\ [\,]
\end{array}
$$

For example, we have:

$$
\begin{array}{ll}
 & reverse\ [1, 2, 3] \\
= & \quad \{\ \text{applying } reverse\ \} \\
 & reverse'\ [1, 2, 3]\ [\,] \\
= & \quad \{\ \text{applying } reverse'\ \} \\
 & reverse'\ [2, 3]\ (1 : [\,]) \\
= & \quad \{\ \text{applying } reverse'\ \} \\
 & reverse'\ [3]\ (2 : (1 : [\,])) \\
= & \quad \{\ \text{applying } reverse'\ \} \\
 & reverse'\ [\,]\ (3 : (2 : (1 : [\,]))) \\
= & \quad \{\ \text{applying } reverse'\ \} \\
 & 3 : (2 : (1 : [\,]))
\end{array}
$$

That is, the list is reversed by using an extra argument to accumulate the final result. The new definition for $reverse$ is perhaps less clear than the original version, but it is much more efficient. In particular, the number of reduction steps required to evaluate $reverse\ xs$ for a list of length n using the new definition is simply $n + 2$, and hence $reverse$ now takes linear time in the length of its argument. For example, reversing a list with ten thousand elements will now take approximately ten thousand steps, in contrast to some fifty million with the original definition — quite an improvement!

Note that we have already seen the use of accumulation to improve the efficiency of functions, in the context of the higher-order library function $foldl$ in chapters 7 and 12. For example, the accumulator version of $reverse$ can also be obtained simply by defining $reverse = foldl\ (:)\ [\,]$. However, it is instructive to see how the same kind of behaviour can be obtained using induction.

As another example of the elimination of append, which also illustrates the use of induction on trees-like types, consider the following type of binary trees, together with a function that flattens such trees to a list:

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree} & = & \mathit{Leaf}\ \mathit{Int} \mid \mathit{Node}\ \mathit{Tree}\ \mathit{Tree} \\
\mathit{flatten} & :: & \mathit{Tree} \rightarrow [\mathit{Int}] \\
\mathit{flatten}\ (\mathit{Leaf}\ n) & = & [n] \\
\mathit{flatten}\ (\mathit{Node}\ l\ r) & = & \mathit{flatten}\ l \mathbin{+\!\!+} \mathit{flatten}\ r
\end{array}
$$

Because of the use of append, the function $\mathit{flatten}$ is inefficient. Let us now construct a more efficient version, by using the same trick as for $\mathit{reverse}$. That is, we seek to define a more general function, $\mathit{flatten}'$, that combines the behaviours of the functions $\mathit{flatten}$ and $\mathbin{+\!\!+}$:

$$\mathit{flatten}'\ t\ ns \quad = \quad \mathit{flatten}\ t \mathbin{+\!\!+} ns$$

In order to prove that some property holds for all trees, the induction principle for the type Tree states that it is sufficient to show that it holds for all trees of the form $\mathit{Leaf}\ n$, and that if the property holds for any trees l and r, then it also holds for $\mathit{Node}\ l\ r$. Using this principle, we construct a definition for $\mathit{flatten}'$ that satisfies the above equation as follows.

Base case:

$$
\begin{array}{ll}
 & \mathit{flatten}'\ (\mathit{Leaf}\ n)\ ns \\
= & \quad \{\ \text{specification of } \mathit{flatten}'\ \} \\
 & \mathit{flatten}\ (\mathit{Leaf}\ n) \mathbin{+\!\!+} ns \\
= & \quad \{\ \text{applying } \mathit{flatten}\ \} \\
 & [n] \mathbin{+\!\!+} ns \\
= & \quad \{\ \text{applying } \mathbin{+\!\!+}\ \} \\
 & n : ns
\end{array}
$$

Inductive case:

$$
\begin{array}{ll}
 & \mathit{flatten}'\ (\mathit{Node}\ l\ r)\ ns \\
= & \quad \{\ \text{specification of } \mathit{flatten}'\ \} \\
 & (\mathit{flatten}\ l \mathbin{+\!\!+} \mathit{flatten}\ r) \mathbin{+\!\!+} ns \\
= & \quad \{\ \text{associativity of } \mathbin{+\!\!+}\ \} \\
 & \mathit{flatten}\ l \mathbin{+\!\!+} (\mathit{flatten}\ r \mathbin{+\!\!+} ns) \\
= & \quad \{\ \text{induction hypothesis for } l\ \} \\
 & \mathit{flatten}'\ l\ (\mathit{flatten}\ r \mathbin{+\!\!+} ns) \\
= & \quad \{\ \text{induction hypothesis for } r\ \} \\
 & \mathit{flatten}'\ l\ (\mathit{flatten}'\ r\ ns)
\end{array}
$$

□

We conclude that the definition

$$
\begin{array}{lcl}
\mathit{flatten}' & :: & \mathit{Tree} \rightarrow [\mathit{Int}] \rightarrow [\mathit{Int}] \\
\mathit{flatten}'\ (\mathit{Leaf}\ n)\ ns & = & n : ns \\
\mathit{flatten}'\ (\mathit{Node}\ l\ r)\ ns & = & \mathit{flatten}'\ l\ (\mathit{flatten}'\ r\ ns)
\end{array}
$$

satisfies the specification for $\mathit{flatten}'$, and hence that the original function $\mathit{flatten}$ can now be redefined as follows:

$$\begin{array}{lcl}
\mathit{flatten} & :: & \mathit{Tree} \rightarrow [\mathit{Int}] \\
\mathit{flatten}\ t & = & \mathit{flatten}'\ t\ [\,]
\end{array}$$

Once again, the new definition for $\mathit{flatten}$ is perhaps less clear than the original version, but is much more efficient, by using an extra argument to accumulate the final result, rather than using append.

13.7 Compiler correctness

We conclude this chapter with an extended example. Recall that in chapter 10 we defined a type of arithmetic expressions built up from integers using an addition operator, together with a function (here called eval) that evaluates an expression directly to an integer value:

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Expr} & = & \mathit{Val\ Int} \mid \mathit{Add\ Expr\ Expr} \\
\mathit{eval} & :: & \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ n) & = & n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = & \mathit{eval}\ x + \mathit{eval}\ y
\end{array}$$

Such expressions can also be evaluated indirectly, by means of code that executes using a stack. In this context, a stack is simply a list of integers, and code comprises a list of push and add operations on the stack:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Stack} & = & [\mathit{Int}] \\
\textbf{type}\ \mathit{Code} & = & [\mathit{Op}] \\
\textbf{data}\ \mathit{Op} & = & \mathit{PUSH\ Int} \mid \mathit{ADD}
\end{array}$$

The meaning of such code is given by defining a function that executes a piece of code using an initial stack to give a final stack:

$$\begin{array}{lcl}
\mathit{exec} & :: & \mathit{Code} \rightarrow \mathit{Stack} \rightarrow \mathit{Stack} \\
\mathit{exec}\ [\,]\ s & = & s \\
\mathit{exec}\ (\mathit{PUSH}\ n : c)\ s & = & \mathit{exec}\ c\ (n : s) \\
\mathit{exec}\ (\mathit{ADD} : c)\ (m : n : s) & = & \mathit{exec}\ c\ (n + m : s)
\end{array}$$

That is, the push operation places a new integer on the top of the stack, while add replaces the top two integers by their sum. Using these operations, it is now straightforward to define a function that compiles an expression into code. An integer value is compiled by simply pushing that value, while an addition is compiled by first compiling the two argument expressions x and y, and then adding the resulting two integers on the stack:

$$\begin{array}{lcl}
\mathit{comp} & :: & \mathit{Expr} \rightarrow \mathit{Code} \\
\mathit{comp}\ (\mathit{Val}\ n) & = & [\mathit{PUSH}\ n] \\
\mathit{comp}\ (\mathit{Add}\ x\ y) & = & \mathit{comp}\ x +\!\!+\ \mathit{comp}\ y +\!\!+ [\mathit{ADD}]
\end{array}$$

Note that when an add operation is performed, the value of expression y will be the top of the stack, and the value of x will be the second top, hence the swapping of these two values in the definition of exec.

To illustrate the behaviour of the three functions defined above, if $e :: Expr$ represents to the expression $(2 + 3) + 4$, then we have:

$> \ eval \ e$
9

$> \ comp \ e$
$[PUSH \ 2, PUSH \ 3, ADD, PUSH \ 4, ADD]$

$> \ exec \ (comp \ e) \ [\,]$
$[9]$

Generalising from this example, the correctness of our compiler for expressions can be expressed by the following equation:

$$exec \ (comp \ e) \ [\,] \quad = \quad [eval \ e]$$

That is, compiling an expression and then executing the resulting code using an empty initial stack gives the same final stack as evaluating the expression and then converting the resulting integer into a singleton stack. For the purposes of proving this result, however, we will see that it is necessary to generalise from the empty initial stack to an arbitrary initial stack:

$$exec \ (comp \ e) \ s \quad = \quad eval \ e : s$$

Using induction for the type $Expr$, which is the same as induction for the type $Tree$ in the previous section except that the names of the constructors are different, the compiler correctness equation can be verified as follows.

Base case:

$$\begin{array}{ll}
 & exec \ (comp \ (Val \ n)) \ s \\
= & \quad \{ \text{ applying } comp \ \} \\
 & exec \ [PUSH \ n] \ s \\
= & \quad \{ \text{ applying } exec \ \} \\
 & n : s \\
= & \quad \{ \text{ unapplying } eval \ \} \\
 & eval \ (Val \ n) : s
\end{array}$$

Inductive case:

$$\begin{array}{ll}
 & exec \ (comp \ (Add \ x \ y)) \ s \\
= & \quad \{ \text{ applying } comp \ \} \\
 & exec \ (comp \ x +\!\!+ \ comp \ y +\!\!+ [ADD]) \ s \\
= & \quad \{ \text{ associativity of } +\!\!+ \ \} \\
 & exec \ (comp \ x +\!\!+ (comp \ y +\!\!+ [ADD])) \ s \\
= & \quad \{ \text{ distributivity - see below } \} \\
 & exec \ (comp \ y +\!\!+ [ADD]) \ (exec \ (comp \ x) \ s) \\
= & \quad \{ \text{ induction hypothesis for } x \ \} \\
 & exec \ (comp \ y +\!\!+ [ADD]) \ (eval \ x : s) \\
= & \quad \{ \text{ distributivity again } \} \\
 & exec \ [ADD] \ (exec \ (comp \ y) \ (eval \ x : s)) \\
= & \quad \{ \text{ induction hypothesis for } y \ \}
\end{array}$$

$$
\begin{array}{ll}
& exec\ [ADD]\ (eval\ y : eval\ x : s) \\
= & \quad \{\text{ applying } exec \} \\
& (eval\ x + eval\ y) : s \\
= & \quad \{\text{ unapplying } eval \} \\
& eval\ (Add\ x\ y) : s
\end{array}
$$

□

Note that without having generalised the result to an arbitrary stack, the second induction hypothesis step would not be applicable, because the stack becomes non-empty at this point. The distributivity property used in the inductive case states that executing two pieces of code appended together gives the same result as executing the two pieces of code in sequence:

$$exec\ (c +\!\!+\ d)\ s \quad = \quad exec\ d\ (exec\ c\ s)$$

The proof of this property proceeds by induction on the code c, with the inductive case being split into two separate cases, depending upon whether the first operation in the code is a push or an addition.

Base case:

$$
\begin{array}{ll}
& exec\ ([\,] +\!\!+\ d)\ s \\
= & \quad \{\text{ applying } +\!\!+ \} \\
& exec\ d\ s \\
= & \quad \{\text{ unapplying } exec \} \\
& exec\ d\ (exec\ [\,]\ s)
\end{array}
$$

Inductive case:

$$
\begin{array}{ll}
& exec\ ((PUSH\ n : c) +\!\!+\ d)\ s \\
= & \quad \{\text{ applying } +\!\!+ \} \\
& exec\ (PUSH\ n : (c +\!\!+\ d))\ s \\
= & \quad \{\text{ applying } exec \} \\
& exec\ (c +\!\!+\ d)\ (n : s) \\
= & \quad \{\text{ induction hypothesis } \} \\
& exec\ d\ (exec\ c\ (n : s)) \\
= & \quad \{\text{ unapplying } exec \} \\
& exec\ d\ (exec\ (PUSH\ n : c)\ s)
\end{array}
$$

Inductive case:

$$
\begin{array}{ll}
& exec\ ((ADD : c) +\!\!+\ d)\ s \\
= & \quad \{\text{ applying } +\!\!+ \} \\
& exec\ (ADD : (c +\!\!+\ d))\ s \\
= & \quad \{\text{ assume } s \text{ of the form } m : n : s' \} \\
& exec\ (ADD : (c +\!\!+\ d))\ (m : n : s') \\
= & \quad \{\text{ applying } exec \} \\
& exec\ (c +\!\!+\ d)\ (n + m : s') \\
= & \quad \{\text{ induction hypothesis } \} \\
& exec\ d\ (exec\ c\ (n + m : s')) \\
= & \quad \{\text{ unapplying } exec \} \\
& exec\ ys\ (exec\ (ADD : c)\ (m : n : s'))
\end{array}
$$

□

The stack not having the assumed form in the second inductive case corresponds to a stack underflow error. In practice, this will never arise, because the structure of the compiler ensures that the stack will always contain at least two integers when an add operation is performed.

In fact, however, both the distributivity property and its consequent underflow issue can be avoided altogether by applying the technique of the previous section to eliminate the use of append. In particular, we seek to define a generalised function $comp'$ with the following property:

$$comp'\ e\ c \quad = \quad comp\ e \mathbin{+\!\!+} c$$

By induction on e, we can construct the definition

$$\begin{array}{lcl} comp' & :: & Expr \rightarrow Code \rightarrow Code \\ comp'\ (Val\ n)\ c & = & PUSH\ n : c \\ comp'\ (Add\ x\ y)\ c & = & comp'\ x\ (comp'\ y\ (ADD : c)) \end{array}$$

from which it follows that we can recover our original definition by $comp\ e = comp'\ e\ [\,]$. In turn, the correctness of the new version of the compiler with respect to our semantics can now be stated as follows:

$$exec\ (comp'\ e\ c)\ s \quad = \quad exec\ c\ (eval\ e : s)$$

That is, compiling an expression and then executing the resulting code together with arbitrary additional code gives the same result as executing the additional code with the value of the expression on top of the original stack. The proof of this result is by induction on the expression e.

Base case:

$$\begin{array}{ll} & exec\ (comp'\ (Val\ n)\ c)\ s \\ = & \quad \{\text{ applying } comp' \} \\ & exec\ (PUSH\ n : c)\ s \\ = & \quad \{\text{ applying } exec \} \\ & exec\ c\ (n : s) \\ = & \quad \{\text{ unapplying } eval \} \\ & exec\ c\ (eval\ (Val\ n) : s) \end{array}$$

Inductive case:

$$\begin{array}{ll} & exec\ (comp'\ (Add\ x\ y)\ c)\ s \\ = & \quad \{\text{ applying } comp' \} \\ & exec\ (comp'\ x\ (comp'\ y\ (ADD : c)))\ s \\ = & \quad \{\text{ induction hypothesis for } x \} \\ & exec\ (comp'\ y\ (ADD : c))\ (eval\ x : s) \\ = & \quad \{\text{ induction hypothesis for } y \} \\ & exec\ (ADD : c)\ (eval\ y : eval\ x : s) \\ = & \quad \{\text{ applying } exec \} \\ & exec\ c\ ((eval\ x + eval\ y) : s) \\ = & \quad \{\text{ unapplying } eval \} \\ & exec\ c\ (eval\ (Add\ x\ y) : s) \end{array}$$

□

Note that with $s = c = [\,]$, this new result simplifies to $exec\ (comp\ e)\ [\,] = [eval\ e]$, our original statement of correctness. In addition to avoiding the problem of stack underflow in the correctness proof, the accumulator version of the compiler has two further benefits. First of all, it avoids the use of $+\!\!+$, and is hence more efficient. And, secondly, the new proof is less than half the combined length of our previous two proofs. As is often the case in formal reasoning, generalising a result in the appropriate manner can considerably simplify its proof. Mathematics is an excellent tool for guiding the development of efficient programs with simple proofs!

13.8 Chapter remarks

Reasoning about functional programs is a subject for a book in its own right, and we have only touched the surface here. Topics for further reading include reasoning about partial and infinite structures (5; 8), relational and point-free approaches (1), automated testing of properties (4), reasoning about effects (24), and techniques that avoid induction (14). The compiler example is adapted from (18), and the phrase "making append vanish" is inspired by (30).

13.9 Exercises

1. Give an example of a function from the standard library in appendix A that is defined using overlapping patterns.

2. Show that $add\ n\ (Succ\ m) = Succ\ (add\ n\ m)$, by induction on n.

3. Using this property, together with $add\ n\ Zero = n$, show that addition is commutative, $add\ n\ m = add\ m\ n$, by induction on n.

4. Using the following definition for the library function that decides if all elements of a list satisfy a predicate

$$\begin{array}{lcl} all\ p\ [\,] & = & True \\ all\ p\ (x : xs) & = & p\ x \wedge all\ p\ xs \end{array}$$

complete the proof of the correctness of $replicate$ by showing that it produces a list with identical elements, $all\ (== x)\ (replicate\ n\ x)$, by induction on $n \geq 0$. Hint: show that the property is always $True$.

5. Using the definition

$$\begin{array}{lcl} [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \end{array}$$

verify the following two properties, by induction on xs:

$$\begin{array}{lcl} xs +\!\!+ [\,] & = & xs \\ xs +\!\!+ (ys +\!\!+ zs) & = & (xs +\!\!+ ys) +\!\!+ zs \end{array}$$

Hint: the proofs are similar to those for the add function.

6. The equation $reverse\ (reverse\ xs) = xs$ can also be proved using a single auxiliary result, $reverse\ (xs +\!\!+ [x]) = x : reverse\ xs$, which can itself be verified by induction on xs. Why might the proof using three auxiliary results as in this chapter be viewed as preferable?

7. Using the definitions

$$\begin{array}{lcl} map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x : map\ f\ xs \\ (f \circ g)\ x & = & f\ (g\ x) \end{array}$$

show that $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$, by induction on xs.

8. Using the definition for $+\!\!+$ given above, together with

$$\begin{array}{lcl} take\ 0\ \_ & = & [\,] \\ take\ (n+1)\ [\,] & = & [\,] \\ take\ (n+1)\ (x : xs) & = & x : take\ n\ xs \\ drop\ 0\ xs & = & xs \\ drop\ (n+1)\ [\,] & = & [\,] \\ drop\ (n+1)\ (\_ : xs) & = & drop\ n\ xs \end{array}$$

show that $take\ n\ xs +\!\!+ drop\ n\ xs = xs$, by simultaneous induction on the integer $n \geq 0$ and the list xs. Hint: there are three cases, one for each pattern of arguments in the definitions of $take$ and $drop$.

9. Given the type declaration

$$\textbf{data}\ Tree \ = \ Leaf\ Int \mid Node\ Tree\ Tree$$

show that the number of leaves in a such a tree is always one greater than the number of nodes, by induction on trees. Hint: start by defining functions that count the number of leaves and nodes in a tree.

10. Given the equation $comp'\ e\ c = comp\ e +\!\!+ c$, show how to construct the recursive definition for $comp'$, by induction on e.