Chapter 4

Induction

A common type of problem is to prove that an object x has some property P. The mathematical notation for this is $P(x)$, where P stands for predicate (or property). For example, if x is 6 and $P(x)$ is the predicate 'x is an even number', then we could express the statement '6 is an even number' with the shorthand mathematical statement $P(6)$.

Many computer science applications require us to prove that *all* the elements of a set S have a certain property P. The statement 'for any element x in the set S the predicate P holds for x' can be written as

$$\forall x \in S \; . \; P(x).$$

Statements like this can be used to assert properties of data: for example, we could state that every item in a database has a certain property. They can also be used to describe the behaviour of computer programs. For example, $P(n)$ might be a statement that a loop computes the correct result if it terminates after n iterations, and the statement $\forall n \in N \; . \; P(n)$ says that the loop is correct if it terminates after *any* number of iterations. N denotes the set of natural numbers, so $n \in N$ just says that n is a natural number, and the complete expression $\forall n \in N \; . \; P(n)$ says that for any natural number n, the predicate $P(n)$ holds.

One approach to proving an assertion about all the elements of a set is to write out a separate direct proof for each element of the set. That would be all right if the set were small. For example, to prove that all the elements of the set $\{4, 6\}$ are even, you could just prove that 4 is even and also that 6 is even. However, this direct approach quickly becomes tedious for large sets: to prove that all the elements of $\{2, 4, 6, \ldots, 1000\}$ are even, you would need 500 separate proofs! Even worse, the brute-force method doesn't work at all—even in principle—for infinite sets, because proofs are always required to be finitely long.

Induction is a powerful method for proving that every element of a set has a certain property. Induction is a valuable technique, because it is not tedious

even if the set is large, and it works even for countably infinite sets. Induction is used frequently in computer science applications. This chapter shows you how inductive proofs work and gives many examples, including both mathematical and computing applications. We will use two forms of induction: *mathematical induction* for proving properties about natural numbers, and *structural induction* for proving properties about lists.

4.1 The Principle of Mathematical Induction

Induction is used to prove that a property $P(x)$ holds for every element of a set S. In this section, we will restrict the set S to be the set N of natural numbers; later we will generalise induction to handle more general sets. Instead of proving $P(x)$ separately for every x, induction relies on a systematic procedure: you just have to prove a few facts about the property P, and then a theorem called the Principle of Mathematical Induction allows you to conclude $\forall x \in N \,.\, P(x)$.

The basic idea is to define a systematic procedure for proving that P holds for an arbitrary element. To do this, we must prove two statements:

1. The *base case* $P(0)$ says that the property P holds for the base element 0.

2. The *inductive case* $P(i) \rightarrow P(i+1)$ says that if P holds for an arbitrary element i of the set, then it must also hold for the successor element $i+1$. The symbol $\rightarrow$ is read as *implies*.

Because every element of the set of natural numbers can be reached by starting with 0 and repeatedly adding 1, you can establish that P holds for any particular element using a finite sequence of steps. Given that $P(0)$ holds (the base case) and that $P(0) \rightarrow P(1)$ (an instance of the inductive case), we can conclude that $P(1)$ also holds. This is a simple example of *logical inference*, which will be studied in more detail in the chapters on logic. The same procedure can be used sytematically to prove $P(i)$ for *any* element of the natural numbers. For example, $P(4)$ can be proved using the following steps (where the $\wedge$ symbol denotes *and*):

Conclusion	Justification		
$P(0)$	the base case		
$P(1)$	$P(0) \rightarrow P(1)$	$\wedge$	$P(0)$
$P(2)$	$P(1) \rightarrow P(2)$	$\wedge$	$P(1)$
$P(3)$	$P(2) \rightarrow P(3)$	$\wedge$	$P(2)$
$P(4)$	$P(3) \rightarrow P(4)$	$\wedge$	$P(3)$

Given any element k of N, you can prove $P(k)$ using this strategy, and the proof will be $k+1$ lines long. We have not yet used the Principle of Mathematical Induction; we have just used ordinary logical reasoning. However, proving $P(k)$ for an arbitrary k is not the same as proving $\forall k \in N \,.\, P(k)$, because

there are an infinite number of values of k, so the proof would be infinitely long. The size of a proof must always be finite.

What the Principle of Mathematical Induction allows us to conclude is that P holds for *all* the elements of N, even if there is an infinite number of them. Thus it introduces something new—it isn't just a macro that expands out into a long proof.

Theorem 5 (Principle of Mathematical Induction)**.** Let $P(x)$ be a predicate that is either true or false for every element of the set N of natural numbers. If $P(0)$ is true and $P(n) \to P(n+1)$ holds for all $n \geq 0$, then $\forall x \in N \ . \ P(x)$ is true.

This theorem can be proved using axiomatic set theory, which is beyond the scope of this book. For most applications in computer science, it is sufficient to have an intuitive understanding of what the theorem says and to have a working understanding of how to use it in proving other theorems.

The proof of the base case will usually turn out to be a straightforward calculation.

The expression $P(n) \to P(n+1)$ means that 'if $P(n)$ is true then so is $P(n+1)$'. We can establish this by temporarily assuming $P(n)$, and then—in the context of this assumption—proving $P(n+1)$. The assumption $P(n)$ is called the *induction hypothesis*. It is important to understand that *assuming the induction hypothesis does not actually mean that it is true!* All that matters is that *if* the assumption $P(n)$ enables us to prove $P(n+1)$, *then* the implication $P(n) \to P(n+1)$ holds. We will study logical inference in more detail in Chapter 6.

4.2 Examples of Induction on Natural Numbers

There is a traditional story about Gauss, one of the greatest mathematicians in history. In school one day the teacher told the class to work out the sum $1+2+\cdots+100$. After a short time thinking, Gauss gave the correct answer—5050—long before it would have been possible to work out all the additions. He had noticed that the sum can be arranged into pairs of numbers, like this:

$$(1+100)+(2+99)+(3+98)+\cdots+(50+51)$$

The total of each pair is 101, and there are 50 of the pairs, so the result is $50 \times 101 = 5050$.

Methods like this can often be used to save time in computing, so it is worthwhile to find a solution to the general case of this problem, which is the sum

$$\sum_{i=1}^{n} i \ = \ 1+2+\cdots+n.$$

Figure 4.1: Geometric Interpretation of Sum

If n is even, then we get $\frac{n}{2}$ pairs that all total to $n+1$, so the result is $\frac{n}{2} \times (n+1) = \frac{n \times (n+1)}{2}$. If n is odd, we can start from 0 instead of 1; for example,

$$\sum_{i=0}^{7} i \;=\; (0+7) + (1+6) + (2+5) + (3+4).$$

In this case there are $\frac{n+1}{2}$ pairs each of which totals to n, so the result is again $\frac{n \times (n+1)}{2}$. For any natural number n, we end up with the result

$$\sum_{i=0}^{n} i \;=\; \frac{n \times (n+1)}{2}.$$

This formula is useful because it reduces the computation time required. For example, if n is 1000 then it would take 999 additions to work out the summation by brute force, but the formula always requires just one addition, one multiplication, and one division.

Figure 4.1 shows another way to understand the formula. The rectangle is covered half by dots and half by stars. The number of stars is $1+2+3+4$, and the area of the rectangle is $4 \times (4+1)$, so the total number of stars is $\frac{4 \times (4+1)}{2}$.

So far we have only guessed the general formula for $\sum_{i=0}^{n} i$, and we have considered two ways of understanding it. The next step is to *prove* that this formula always gives the right answer, and induction provides a way to do it.

Theorem 6.

$$\forall n \in N \;.\; \sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}$$

Proof. Define the property P as follows:

$$P(n) \;=\; \left(\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2} \right)$$

Thus the aim is to prove that

$$\forall n \in N \;.\; P(n)$$

and we proceed by induction on n.

Base case. We need to prove $P(0)$.

$$\begin{aligned}\sum_{i=0}^{0} i &= 0 \\ &= \frac{0 \times (0+1)}{2}\end{aligned}$$

Thus we have established the property $P(0)$.

Induction case. The aim is to prove that $P(n) \rightarrow P(n+1)$, and we will prove this implication by assuming $P(n)$ (this is called the *induction hypothesis*) and then using that assumption to prove $P(n+1)$. We start by writing out in full detail the assumption and the aim of the proof. Assume for an arbitrary natural number n that

$$\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}.$$

The aim is to show (for this particular value of n) that

$$\sum_{i=0}^{n+1} i = \frac{(n+1) \times (n+2)}{2}.$$

We do this by starting with the left-hand side of the equation and using algebra to transform it into the right-hand side. The first step uses the assumption given above. This is called the *induction hypothesis*.

$$\begin{aligned}\sum_{i=0}^{n+1} i &= \sum_{i=0}^{n} i + (n+1) \\ &= \frac{n \times (n+1)}{2} + (n+1) \\ &= \frac{n \times (n+1)}{2} + \frac{2 \times (n+1)}{2} \\ &= \frac{n \times (n+1) + 2 \times (n+1)}{2} \\ &= \frac{(n+1) \times (n+2)}{2}\end{aligned}$$

Now we have established that

$$\left(\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}\right) \rightarrow \left(\sum_{i=0}^{n+1} i = \frac{(n+1) \times (n+2)}{2}\right).$$

That is,

$$P(n) \rightarrow P(n+1).$$

Use the principle of induction. To summarise, we have proved the base case $P(0)$, and we have proved the induction case $P(n) \to P(n+1)$. Therefore the principle of induction allows us to conclude that the theorem $\forall n \in N.P(n)$ is true. □

Exercise 1. Let a be an arbitrary real number. Prove, for all natural numbers m and n, that $a^{m \times n} = (a^m)^n$.

Exercise 2. Prove that the sum of the first n odd positive numbers is n^2.

Exercise 3. Prove that $\sum_{i=1}^{n} a^i = (a^{n+1} - 1)/(a - 1)$, where a is a real number and $a \neq 1$.

4.3 Induction and Recursion

Induction is a common method for proving properties of recursively defined functions. The *factorial* function, which is defined recursively, provides a good example of an induction proof.

$$\begin{aligned} &\mathit{factorial} :: \mathit{Natural} \to \mathit{Natural} \\ &\mathit{factorial}\, 0 = 1 \\ &\mathit{factorial}\, (n+1) = (n+1) * \mathit{factorial}\, n \end{aligned}$$

A common problem in computer science is to prove that a program computes the correct answer. Such a theorem requires an abstract mathematical specification of the problem, and it has to show that for all inputs, the program produces the same result that is defined by the specification. The value of $n!$ (factorial of n) is defined as the product of all the natural numbers from 1 through n; the standard notation for this is $\prod_{i=1}^{n} i$. The following theorem says that the *factorial* function as defined above actually computes the value of $n!$.

Theorem 7. For all natural numbers n, $\mathit{factorial}\, n = \prod_{i=1}^{n} i$.

Proof. Induction on n. The base case is

$$\begin{aligned} &\mathit{factorial}\, 0 \\ &\quad = 1 && \{\ \mathit{factorial}.1\ \} \\ &\quad = \textstyle\prod_1^0 i && \{\ \text{def. of } \textstyle\prod\ \} \end{aligned}$$

For the induction case, it is necessary to prove that

$$\left(\mathit{factorial}\, n = \prod_{i=1}^{n} i\right) \to \left(\mathit{factorial}\, (n+1) = \prod_{i=1}^{n+1} i\right).$$

This is done by assuming the left side as the inductive hypothesis: for a particular n, the hypothesis is $\mathit{factorial}\ n\ =\ \prod_{i=1}^{n} i$. Given this assumption, we must prove that $\mathit{factorial}\ (n+1)\ =\ \prod_{i=1}^{n+1} i$.

$$
\begin{array}{ll}
\mathit{factorial}\ (n+1) & \\
\quad = (n+1)\ \times \mathit{factorial}\ n & \{\ \mathit{factorial}.2\ \} \\
\quad = (n+1) \times \prod_{i=1}^{n} i & \{\ \text{hypothesis}\ \} \\
\quad = \prod_{i=1}^{n+1} i & \{\ \text{def. } \textstyle\prod\ \}
\end{array}
$$

□

Exercise 4. (This problem is from [12], where you can find many more.) The nth Fibonacci number is defined as follows:

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

The first few numbers in this famous sequence are $0, 1, 1, 2, 3, 5, \ldots$. Prove the following:

$$\sum_{i=1}^{n} \mathit{fib}\ i\ =\ \mathit{fib}\ (n+2) - 1$$

4.4 Induction on Peano Naturals

The Peano representation of natural numbers is a rich source of examples for induction. Actually, it's hard to do anything at all in Peano arithmetic without induction. For example, how do we even know that a natural number is equal to itself? The following theorem says so, and its proof requires induction.

Theorem 8 (Self equality)**.** $\forall x :: \mathit{Nat}.\ \ \mathit{equals}\ x\ x = \mathit{True}$

Proof. Base case:

$$
\begin{array}{ll}
\mathit{equals}\ \mathit{Zero}\ \mathit{Zero} & \\
\quad = \mathit{True} & \{\ \mathit{equals}.1\ \}
\end{array}
$$

For the inductive case, assume that $\mathit{equals}\ x\ x\ =\ \mathit{True}$. We must prove that $\mathit{equals}\ (\mathit{Succ}\ x)\ (\mathit{Succ}\ x)\ =\ \mathit{True}$.

$$
\begin{array}{ll}
\mathit{equals}\ (\mathit{Succ}\ x)\ (\mathit{Succ}\ x) & \\
\quad = \mathit{equals}\ x\ x & \{\ \mathit{equals}.2\ \} \\
\quad = \mathit{True} & \{\ \text{hypothesis}\ \}
\end{array}
$$

□

This proof illustrates a subtle issue. When we are using the languages of English and mathematics to talk *about* natural numbers, we can assume that anything is equal to itself. We don't normally prove theorems like $x = x$. However, that is not what we have just proved: the theorem above says that x is the same as itself according to *equals*, which is defined *inside* the Peano system. Therefore the proof also needs to work inside the Peano system; hence the induction.

We'll look at a few more typical Peano arithmetic theorems, both to see how the Peano natural numbers work and to get more practice with induction. The following theorem says, in effect, that $(x + y) - x = y$. In elementary algebra, we would prove this by calculating $(x + y) - x = (x - x) + y = 0 + y = y$. The point here, however, is that the addition and subtraction are being performed by the recursive *add* and *sub* functions, and we need to prove the theorem in terms of these functions.

Theorem 9. $sub\ (add\ x\ y)\ x = y$

Proof. Induction over x. The base case is

$$\begin{aligned} &sub\ (add\ Zero\ y)\ Zero \\ &\quad = sub\ y\ Zero && \{\ add.1\ \} \\ &\quad = y && \{\ sub.1\ \} \end{aligned}$$

For the inductive case, assume $sub\ (add\ x\ y)\ x = y$; the aim is to prove $sub\ (add\ (Succ\ x)\ y)\ (Succ\ x) = y$.

$$\begin{aligned} &sub\ (add\ (Succ\ x)\ y)\ (Succ\ x) \\ &\quad = sub\ (Succ\ (add\ x\ y))\ (Succ\ x) && \{\ add.2\ \} \\ &\quad = sub\ (add\ x\ y)\ x && \{\ sub.3\ \} \\ &\quad = y && \{\ \text{hypothesis}\ \} \end{aligned}$$

□

The proof above happens to go through directly and easily, but many simple theorems do not. For example, it is considerably harder to prove $(x+y)-y = x$ than to prove $(x + y) - x = y$. Even worse, there is no end to such theorems. Instead of continuing to choose theorems based on their ease of proof, it is better to proceed systematically by developing the standard properties of natural numbers, such as associativity and commutativity. The attractive feature of the Peano definitions is that all these laws are theorems; the only definitions we need are the basic ones given already. It is straightforward to prove that addition is associative, so we begin with that property.

Theorem 10 (add is associative)**.** $add\ x\ (add\ y\ z)\ =\ add\ (add\ x\ y)\ z$

Proof. Induction over x. The Base case is

$$
\begin{array}{ll}
& add\ Zero\ (add\ y\ z) \\
= & add\ y\ z & \{\ add.1\ \} \\
= & add\ (add\ Zero\ y)\ z & \{\ add.1\ \}
\end{array}
$$

Inductive case. Assume $add\ x\ (add\ y\ z) = add\ (add\ x\ y)\ z$. Then

$$
\begin{array}{ll}
& add\ (Succ\ x)\ (add\ y\ z) \\
= & Succ\ (add\ x\ (add\ y\ z)) & \{\ add.2\ \} \\
= & Succ\ (add\ (add\ x\ y)\ z) & \{\ \text{hypothesis}\ \} \\
= & add\ (Succ\ (add\ x\ y))\ z & \{\ add.2\ \} \\
= & add\ (add\ (Succ\ x)\ y)\ z & \{\ add.2\ \}
\end{array}
$$

□

Next, it would be good to prove that addition is commutative: $x + y = y + x$. To prove this, however, a sequence of simpler theorems is needed—each providing yet another example of induction.

First we need to be able to simplify additions where the *second* argument is Zero. We know already that $add\ Zero\ x = x$; in fact, this is one of the Peano axioms. It is *not* an axiom that $add\ x\ Zero = x$; that is a theorem requiring proof.

Theorem 11. $add\ x\ Zero = x$

Proof. Induction over x. The base case is

$$
\begin{array}{ll}
& add\ Zero\ Zero \\
= & Zero & \{\ add.1\ \}
\end{array}
$$

For the inductive case, we assume $add\ x\ Zero = x$. Then

$$
\begin{array}{ll}
& add\ (Succ\ x)\ Zero \\
= & Succ\ (add\ x\ Zero) & \{\ add.2\ \} \\
= & Succ\ x & \{\ \text{hypothesis}\ \}
\end{array}
$$

□

The next theorem allows us to move a *Succ* from one argument of an addition to the other. It says, in effect, that $(x + 1) + y = x + (y + 1)$. That may not sound very dramatic, but many proofs require the ability to take a little off one argument and add it onto the other.

Theorem 12. $add\ (Succ\ x)\ y = add\ x\ (Succ\ y)$

Proof. Induction over x. Base case:

$$
\begin{array}{ll}
& add\ (Succ\ Zero)\ y \\
= & Succ\ (add\ Zero\ y) & \{\ add.2\ \} \\
= & Succ\ y & \{\ add.1\ \} \\
= & add\ Zero\ (Succ\ y) & \{\ add.1\ \}
\end{array}
$$

Inductive case:

$$\begin{array}{ll} add\ (Succ\ (Succ\ x))\ y & \\ \quad = Succ\ (add\ (Succ\ x)\ y) & \{\ add.2\ \} \\ \quad = Succ\ (add\ x\ (Succ\ y)) & \{\ \text{hypothesis}\ \} \\ \quad = add\ (Succ\ x)\ (Succ\ y) & \{\ add.2\ \} \end{array}$$

□

Now we can prove that Peano addition is commutative.

Theorem 13 (add is commutative)**.** $add\ x\ y\ =\ add\ y\ x$

Proof. Induction over x. Base case:

$$\begin{array}{ll} add\ Zero\ y & \\ \quad = y & \{\ add.1\ \} \\ \quad = add\ y\ Zero & \{\ \text{Theorem 11}\ \} \end{array}$$

Inductive case: assume that $add\ x\ y\ =\ add\ y\ x$. Then

$$\begin{array}{ll} add\ (Succ\ x)\ y & \\ \quad = Succ\ (add\ x\ y) & \{\ add.2\ \} \\ \quad = Succ\ (add\ y\ x) & \{\ \text{hypothesis}\ \} \\ \quad = add\ (Succ\ y)\ x & \{\ add.2\ \} \\ \quad = add\ y\ (Succ\ x) & \{\ \text{Theorem 12}\ \} \end{array}$$

□

4.5 Induction on Lists

Lists are one of the most commonly used data structures in computing, and there is a large family of functions to manipulate them. These functions are typically defined recursively, with a base case for empty lists $[\,]$ and a recursive case for non-empty lists, i.e. lists of the form $(x : xs)$. List induction is the most common method for proving properties of such functions.

Before going on, we discuss some practical techniques that help in coping with theorems. If you aren't familiar with them, mathematical statements sometimes look confusing, and it is easy to develop a bad habit of skipping over the mathematics when reading a book. Here is some advice on better approaches; we will try to illustrate the advice in a concrete way in this section, but these methods will pay off throughout your work in computer science, not just in the section you're reading right now.

- When you are faced with a new theorem, try to understand what it means before trying to prove it. Restate the main idea in English.

- Think about what applications the theorem might have. If it says that two expressions are the same, can you think of situations where there might be a practical advantage in replacing the left-hand side by the right-hand side, or vice versa? (A common situation is that one side of the equation is more natural to write, and the other side is more efficient.)
- Try out the theorem on some small examples. Theorems are often stated as equations; make up some suitable input data and evaluate both sides of the equation to see that they are the same.
- Check what happens in boundary cases. If the equation says something about lists, what happens if the list is empty? What happens if the list is infinite?
- Does the theorem seem related to other ones? Small theorems about functions—the kind that are usually proved by induction—tend to fit together in families. Noticing these relationships helps in understanding, remembering, and applying the results.

The principle of list induction states a technique for proving properties about lists. It is similar to the principle of mathematical induction; the main difference is that the base case is the empty list (rather than 0) and the induction case uses a list with one additional element (`x:xs`) rather than $n+1$. List induction is a special case of a more general technique called *strutural induction*. Induction over lists is used to prove that a proposition $P(xs)$ holds for every list xs.

Theorem 14 (Principle of list induction)**.** Suppose $P(xs)$ is a predicate on lists of type $[a]$, for some type a. Suppose that $P([\,])$ is true (this is the base case). Further, suppose that if $P(xs)$ holds for arbitrary $xs :: [a]$, then $P(x : xs)$ also holds for arbitrary $x :: a$. Then $P(xs)$ holds for every list xs that has finite length.

Thus the base case is to prove that the predicate holds for the empty list, and the inductive case is to prove that *if* P holds for a list xs, then it must *also* hold for any list of the form $x : xs$. When the base and inductive case are established, then the principle of induction allows us to conclude that the predicate holds for all finite lists.

Notice that the principle of list induction cannot be used to prove theorems about infinite lists. This point is discussed in Section 4.8.

We will now work through a series of examples where induction is used to prove theorems about the properties of recursive functions over lists.

The *sum* function takes a list of numbers and adds them up. It defines the sum of an empty list to be 0, and the sum of a non-empty list is computed by adding the first number onto the sum of all the rest.

```
sum :: Num a => [a] -> a
```

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

The following theorem states a useful fact about the relationship between two functions: *sum* and ++. It says that if you have two lists, say *xs* and *ys*, then there are two different ways to compute the combined total of all the elements. You can either append the lists together with ++, and then apply *sum*, or you can apply *sum* independently to the two lists, and add the two resulting numbers.

Theorems of this sort are often useful for transforming programs to make them more efficient. For example, suppose that you have a very long list to sum up, and two computers are available. We could use parallelism to cut the execution time almost in half. The idea is to split up the long list into two shorter ones, which can be summed in parallel. One quick addition will then suffice to get the final result. This technique is an example of the *divide and conquer* strategy for improving the efficiency of algorithms. Obviously there is more to parallel computing and program optimisation than we have covered in this paragraph, but theorems like the one we are considering really do have practical applications.

Theorem 15. $sum\ (xs{+\!\!+}ys) \;=\; sum\ xs + sum\ ys$

The proof of this theorem is a typical induction over lists, and it provides a good model to follow for future problems. The justifications used in the proof steps are written in braces {...}. Many of the justifications cite the definition of a function, along with the number of the equation in the definition; thus { (++).1 } means 'this step is justified by the first equation in the definition of ++'. The most crucial step in an induction proof is the one where the induction hypothesis is used; the justification cited for that step is { hypothesis }.

Proof. Induction over *xs*. The base case is

$$
\begin{array}{ll}
sum\ ([\,] {+\!\!+}\ ys) & \\
\quad = sum\ ys & \{\ (+\!\!+).1\ \ \} \\
\quad = 0 + sum\ ys & \{\ 0 + x = x\ \} \\
\quad = sum\ [\,] + sum\ ys & \{\ sum.1\ \}
\end{array}
$$

The inductive case is

$$
\begin{array}{ll}
sum\ ((x : xs){+\!\!+}ys) & \\
\quad = sum\ (x : (xs{+\!\!+}ys)) & \{\ (+\!\!+).2\ \} \\
\quad = x + sum\ (xs{+\!\!+}ys) & \{\ sum.2\ \} \\
\quad = x + (sum\ xs + sum\ ys) & \{\ \text{hypothesis}\ \} \\
\quad = (x + sum\ xs) + sum\ ys & \{\ +.\ \text{is associative}\ \} \\
\quad = sum\ (x : xs) + sum\ ys & \{\ sum.2\ \}
\end{array}
$$

□

Many theorems describe a relationship between two functions; the previous one is about the combination of *sum* and (++), while this one combines *length* with (++). Its proof is left as an exercise.

Theorem 16. *length* (*xs*++*ys*) = *length xs* + *length ys*

We now consider several theorems that state crucial properties of the *map* function. These theorems are important in their own right, and they are commonly used in program transformation and compiler implementation. They are also frequently used to justify steps in proofs of more complex theorems.

The following theorem says that *map* is 'length preserving': when you apply *map* to a list, the result has the same length as the input list.

Theorem 17. *length* (*map f xs*) = *length xs*

Proof. Induction over *xs*. The base case is

length (*map f* [])
  = *length* [] { *map*.1 }

For the inductive case, assume *length* (*map f xs*) = *length xs*. Then

length (*map f* (*x* : *xs*))
  = *length* (*f x* : *map f xs*) { *map*.2 }
  = 1 + *length* (*map f xs*) { *length*.2 }
  = 1 + *length xs* { hypothesis }
  = *length* (*x* : *xs*) { *length*.2 }

□

The next theorem is reminiscent of Theorem 15; it says you can get the same result by either of two methods: (1) mapping a function over two lists and then appending the results together, and (2) appending the input lists and then performing one longer map over the result. Its proof is yet another good example of induction, and is left as an exercise.

Theorem 18. *map f* (*xs*++*ys*) = *map f xs* ++ *map f ys*

One of the most important properties of map is expressed precisely by the following theorem. Suppose that you have two computations to perform on all the elements of a list. First you want to apply g to an element, getting an intermediate result to which you want to apply f. There are two methods for doing the computation. The first method uses two separate loops, one to perform g on every element and the second loop to perform f on the list of intermediate results. The second method is to use just one loop, and each iteration performs the g and f applications in sequence. This theorem is used commonly by optimising compilers, program transformations (both manual and automatic), and it's also vitally important in parallel programming. Again, we leave the proof as an exercise.

Theorem 19. $(map\ f\ .\ map\ g)\ xs\ =\ map\ (f.g)\ xs$

For a change of pace, we now consider an intriguing theorem. Once you understand what it says, however, it becomes perfectly intuitive. (The notation (1+) denotes a function that adds 1 to a number.)

Theorem 20. $sum\ (map\ (1+)\ xs)\ =\ length\ xs\ +\ sum\ xs$

Proof. Induction over *xs*. The base case is

$$\begin{array}{ll} sum\ (map\ (1+)\ [\,]) & \\ \quad = sum\ [\,] & \{\ map.1\ \} \\ \quad = 0 + sum\ [\,] & \{\ 0 + x = x\ \} \\ \quad = length\ [\,]\ +\ sum\ [\,] & \{\ length.1\ \} \end{array}$$

For the inductive case, assume $sum\ (map\ (1+)\ xs)\ =\ length\ xs\ +\ sum\ xs$. Then

$$\begin{array}{ll} sum\ (map\ (1+)\ (x : xs)) & \\ \quad = sum\ ((1 + x)\ :\ map\ (1+)\ xs) & \{\ map.2\ \} \\ \quad = (1 + x)\ +\ sum\ (map\ (1+)\ xs) & \{\ sum.2\ \} \\ \quad = (1 + x)\ +\ (length\ xs\ +\ sum\ xs) & \{\ \text{hypothesis}\ \} \\ \quad = (1 + length\ xs)\ +\ (x\ +\ sum\ xs) & \{\ (+).\text{algebra}\ \} \\ \quad = length\ (x : xs)\ +\ sum\ (x : xs) & \{\ length.2,\ sum.2\ \} \end{array}$$

□

The *foldr* function is important because it expresses a basic looping pattern. There are many important properties of *foldr* and related functions. Here is one of them:

Theorem 21. $foldr\ (:)\ [\,]\ xs\ =\ xs$

Some of the earlier theorems may be easy to understand at a glance, but that is unlikely to be true for this one! Recall that the *foldr* function takes apart a list and combines its elements using an operator. For example,

$$foldr\ (+)\ 0\ [1, 2, 3]\ =\ 1\ +\ (2\ +\ (3\ +\ 0))$$

Now, what happens if we combine the elements of the list using the cons operator (:) instead of addition, and if we use [ ] as the initial value for the recursion instead of 0? The previous equation then becomes

$$\begin{array}{rcl} foldr\ (:)\ [\,]\ [1, 2, 3] & = & 1\ :\ (2\ :\ (3\ :\ [\,])) \\ & = & [1, 2, 3]. \end{array}$$

We ended up with the same list we started out with, and the theorem says this will always happen, not just with the example $[1, 2, 3]$ used here.

Proof. Induction over *xs*. The base case is

$$
\begin{array}{ll}
\mathit{foldr}\ (:)\ [\,]\ [\,] & \\
\quad = [\,] & \{\ \mathit{foldr}.1\ \ \}
\end{array}
$$

Now assume that $\mathit{foldr}\ (:)\ [\,]\ xs\ =\ xs$; then the inductive case is

$$
\begin{array}{ll}
\mathit{foldr}\ (:)\ [\,]\ (x : xs) & \\
\quad = x\ :\ \mathit{foldr}\ (:)\ [\,]\ xs & \{\ \mathit{foldr}.2\ \} \\
\quad = x : xs & \{\ \text{hypothesis}\ \}
\end{array}
$$

□

Suppose you have a list of lists, of the form $xss\ =\ [xs_0, xs_1, \ldots, xs_n]$. All of the lists xs_i must have the same type $[a]$, and the type of xss is $[[a]]$. We might want to apply a function $f :: a \to b$ to all the elements of all the lists, and build up a list of all the results. There are two different ways to organise this computation:

- Use the *concat* function to make a single flat list of type $[a]$ containing all the values, and then apply $\mathit{map}\ f$ to produce the result with type $[b]$.
- Apply $\mathit{map}\ f$ separately to each xs_i, by computing $\mathit{map}\ (\mathit{map}\ f)\ xss$, producing a list of type $[[b]]$. Then use *concat* to flatten them into a single list of type $[b]$.

The following theorem guarantees that both approaches produce the same result. This is significant because there are many practical situations where it is more convenient to write an algorithm using one approach, yet the other is more efficient.

Theorem 22. $\mathit{map}\ f\ (\mathit{concat}\ xss)\ =\ \mathit{concat}\ (\mathit{map}\ (\mathit{map}\ f)\ xss)$

Proof. Proof by induction over xss. The base case is

$$
\begin{array}{ll}
\mathit{map}\ f\ (\mathit{concat}\ [\,]) & \\
\quad = \mathit{map}\ f\ [\,] & \{\ \mathit{concat}.1\ \ \} \\
\quad = [\,] & \{\ \mathit{map}.1\ \} \\
\quad = \mathit{concat}\ [\,] & \{\ \mathit{concat}.1\ \} \\
\quad = \mathit{concat}\ (\mathit{map}\ (\mathit{map}\ f)\ [\,]) & \{\ \mathit{map}.1\ \}
\end{array}
$$

Assume that $\mathit{map}\ f\ (\mathit{concat}\ xss)\ =\ \mathit{concat}\ (\mathit{map}\ (\mathit{map}\ f)\ xss)$. The inductive case is

$$
\begin{array}{ll}
\mathit{map}\ f\ (\mathit{concat}\ (xs : xss)) & \\
\quad = \mathit{map}\ f\ (xs +\!\!+ \mathit{concat}\ xss) & \{\ \mathit{concat}.2\ \} \\
\quad = \mathit{map}\ f\ xs +\!\!+ \mathit{map}\ f\ (\mathit{concat}\ xss) & \{\ \text{Theorem 18}\ \} \\
\quad = \mathit{map}\ f\ xs +\!\!+ \mathit{concat}\ (\mathit{map}\ (\mathit{map}\ f)\ xss) & \{\ \text{hypothesis}\ \} \\
\quad = \mathit{concat}\ (\mathit{map}\ f\ xs\ :\ \mathit{map}\ (\mathit{map}\ f)\ xss) & \{\ \mathit{concat}.2\ \} \\
\quad = \mathit{concat}\ (\mathit{map}\ (\mathit{map}\ f)\ (xs : xss)) & \{\ \mathit{map}.2\ \}
\end{array}
$$

□

Sometimes you don't need to perform an induction, because a simpler proof technique is already available. Here is a typical example:

Theorem 23. $length\ (xs{+\!\!+}(y : ys)) \ = \ 1 \ + \ length\ xs \ + \ length\ ys$

This theorem could certainly be proved by induction (and that might be good practice for you!) but we already have a similar theorem which says that $length\ (xs{+\!\!+}ys) \ = \ length\ xs \ + \ length\ ys$. Instead of starting a new induction completely afresh, it's more elegant to carry out a few steps that enable us to apply the existing theorem. Just as reuse of software is a good idea, reuse of theorems is good style in theoretical computer science.

$$\begin{aligned} &length\ (xs +\!\!+ (y : ys)) \\ &\quad = length\ xs + length\ (y : ys) \\ &\quad = length\ xs \ + \ (1 + length\ ys) \\ &\quad = 1 + length\ xs + length\ ys \end{aligned}$$

Exercise 5. Prove Theorem 16.

Exercise 6. Prove Theorem 18.

Exercise 7. Prove Theorem 19.

Exercise 8. Recall Theorem 20, which says

$$sum\ (map\ (1+)\ xs) \ = \ length\ xs \ + \ sum\ xs.$$

Explain in English what this theorem says. Using the definitions of the functions involved (*sum*, *length* and *map*), calculate the values of the left and right-hand sides of the equation using $xs = [1, 2, 3, 4]$.

Exercise 9. Invent a new theorem similar to Theorem 20, where $(1+)$ is replaced by $(k+)$. Test it on one or two small examples. Then prove your theorem.

4.6 Functional Equality

Many theorems used in computer science (including most of the ones in this chapter) say that two different algorithms are always guaranteed to produce the same result. The algorithms are defined as functions, and the theorem says that when you apply two different functions to the same argument, they give the same result.

It is simpler and more direct simply to say that the two functions are equal. However, this raises an interesting question: what does it mean to say $f = g$ when f and g are functions? (This issue will be revisited in Chapter 11.) There are at least two standard notions of functional equality that are completely different from each other, so it pays to be careful!

- *Intensional equality.* Two functions f and g are intensionally equal if their definitions are identical. This means, of course, that the functions are not equal if their types are different. If they are computer programs, testing for intensional equality involves comparing the source programs, character by character. The functions are intensionally equal if their definitions *look* the same.

- *Extensional equality.* Two functions f and g are extensionally equal if they have the same type $a \to b$ and $f\ x = g\ x$ for all well typed arguments $x :: a$. More precisely, $f = g$ if and only if

  $$\forall x :: a\ .\ f\ x\ =\ g\ x.$$

  The functions are extensionally equal if they deliver the same results when given the same inputs.

In computer science we are almost always interested in extensional equality. A typical situation is that we have an algorithm, expressed as a function f, and the aim is to replace it by a more efficient function g. This will not affect the correctness of the program as long as f and g are extensionally equal, but they are obviously not intensionally equal.

Some of the theorems given in the previous section can be stated in a simpler fashion using extensional equality. For example, recall Theorem 17, which says that map doesn't change the length of its argument:

$$length\ (map\ f\ xs) = length\ xs$$

A more direct way to state the same fact is to omit the irrelevant argument xs, and just say that these two functions are equal:

$$length\ .\ (map\ f) = length$$

To prove such a theorem of the form $f = g$, we need only prove that $\forall x :: a.\ f\ x = g\ x$, and this can be achieved by choosing an arbitrary $x :: a$, and proving the equation $f\ x = g\ x$.

Theorem 24. $foldr\ (:)\ [\,]\ =\ id$

Proof. The equation states that two functions are equal: the right-hand side, *id*, is a function, and the left-hand side is a partial application (*foldr* takes three arguments, but it has been applied to only two), so that is also a function. Therefore we use the definition of extensional equality of functions; thus we choose an arbitrary list xs, and we must prove that $foldr\ (:)\ [\,]\ xs\ =\ id\ xs\ =\ xs$. Now the right-hand side is just xs, by the definition of *id*, so the equation is proved by Theorem 21. □

Theorem 25. $map\ f\ .\ concat\ =\ concat\ (map\ (map\ f))$.