

6

Reasoning About Functional Programs

Most programmers know how hard it is to make a program work. In the 1970s, it became apparent that programmers could no longer cope with software projects that were growing ever more complex. Systems were delayed and cancelled; costs escalated. In response to this software crisis, several new methodologies have arisen — each an attempt to master the complexity of large systems.

Structured programming seeks to organize programs into simple parts with simple interfaces. An *abstract data type* lets the programmer view a data structure, with its operations, as a mathematical object. The next chapter, on modules, will say more about these topics.

Functional programming and *logic programming* aim to express computations directly in mathematics. The complicated machine state is made invisible; the programmer has to understand only one expression at a time.

Program correctness proofs are introduced in this chapter. Like the other responses to the software crisis, formal methods aim to increase our understanding. The first lesson is that a program only ‘works’ if it is *correct* with respect to its *specification*. Our minds cannot cope with the billions of steps in an execution. If the program is expressed in a mathematical form, however, then each stage of the computation can be described by a formula. Programs can be *verified* — proved correct — or *derived* from a specification. Most of the early work on program verification focused on Pascal and similar languages; functional programs are easier to reason about because they involve no machine state.

Chapter outline

The chapter presents proofs about functional programs, paying particular attention to induction. The proof methods are rigorous but informal. Their purpose is to increase our understanding of the programs.

The chapter contains the following sections:

Some principles of mathematical proof. A class of ML programs can be treated

within elementary mathematics. Some integer functions are verified using mathematical induction.

Structural induction. This principle generalizes mathematical induction to finite lists and trees. Proofs about higher-order functions are presented.

A general induction principle. Some unusual inductive proofs are discussed. Well-founded induction provides a uniform framework for such proofs.

Specification and verification. The methods of the chapter are applied to an extended example: the verification of a merge sort function. Some limitations of verification are discussed.

Some principles of mathematical proof

The proofs in this chapter are conducted in a style typical of discrete mathematics. Most proofs are by induction. Much of the reasoning is equational, replacing equals by equals, although the logical connectives and quantifiers are indispensable.

6.1 ML programs and mathematics

Our proofs treat Standard ML programs as mathematical objects, subject to mathematical laws. A theory of the full language would be too complicated; let us restrict the form of programs. Only functional programs will be allowed; ML's imperative features will be forbidden. Types will be interpreted as sets, which restricts the form of `datatype` declarations. Exceptions are forbidden, although it would not be hard to incorporate them into our framework. We shall allow only well-defined expressions. They must be legally typed, and must denote terminating computations.

If all computations must terminate, recursive function definitions have to be restricted. Recall the function *facti*, declared as follows:

```
fun facti (n, p) =
  if n=0 then p else facti (n-1, n*p);
```

Recall from Section 2.11 that functional programs are computed by reduction:

$$\text{facti}(4, 1) \Rightarrow \text{facti}(4 - 1, 4 \times 1) \Rightarrow \text{facti}(3, 4) \Rightarrow \dots \Rightarrow 24$$

Computing *facti*(*n*, *p*) yields a unique result for all $n \geq 0$; thus *facti* is a mathematical function satisfying these laws:

$$\begin{aligned} \text{facti}(0, p) &= p \\ \text{facti}(n, p) &= \text{facti}(n - 1, n \times p) \quad \text{for } n > 0 \end{aligned}$$

If $n < 0$ then $facti(n, p)$ produces a computation that runs forever; it is undefined. We may regard $facti(n, p)$ as meaningful only for $n \geq 0$, which is the function's **precondition**.

For another example, consider the following declaration:

```
fun undef (x) = undef (x) - 1;
```

Since $undef(x)$ does not terminate for any x , we shall not regard it as meaningful. We may not adopt $undef(x) = undef(x) - 1$ as a law about numbers, for it is clearly false.

It is possible to introduce the value \perp (called 'bottom') for the value of a nonterminating computation, and develop a **domain theory** for reasoning about arbitrary recursive function definitions. Domain theory interprets $undef$ as the function satisfying $undef(x) = \perp$ for all x . It turns out that $\perp - 1 = \perp$, so $undef(x) = undef(x) - 1$ means simply $\perp = \perp$, which is valid. But domain theory is complex and difficult. The value \perp induces a partial ordering on all types. All functions in the theory must be monotonic and continuous over this partial ordering; recursive functions denote least fixed points. By insisting upon termination, we can work within elementary set theory.

Restricting ourselves to terminating computations entails some sacrifices. It is harder to reason about programs that do not always terminate, such as interpreters. Nor can we reason about lazy evaluation — which is a pity, for using this sophisticated form of functional programming requires mathematical insights. Most functional programmers eventually learn some domain theory; there is no other way to understand what computation over an infinite list really means.

Logical notation. This chapter assumes you have some familiarity with formal proof. We shall adopt the following notation for logical formulæ:

$\neg\phi$	not ϕ
$\phi \wedge \psi$	ϕ and ψ
$\phi \vee \psi$	ϕ or ψ
$\phi \rightarrow \psi$	ϕ implies ψ
$\phi \leftrightarrow \psi$	ϕ if and only if ψ
$\forall x . \phi(x)$	for all x , $\phi(x)$
$\exists x . \phi(x)$	for some x , $\phi(x)$

From highest to lowest precedence, the connectives are \neg , \wedge , \vee , \rightarrow , \leftrightarrow . Here is an example of precedence in formulæ:


$$P \wedge Q \rightarrow P \vee \neg R \text{ abbreviates } (P \wedge Q) \rightarrow (P \vee (\neg R))$$

Quantifiers have the widest possible scope to the right:

$$\forall x . P \wedge \exists y . Q \rightarrow R \text{ abbreviates } \forall x . (P \wedge (\exists y . (Q \rightarrow R)))$$

Many logicians prefer a slightly different quantifier notation, omitting the dots. Then $\forall x P \wedge \exists y Q \rightarrow R$ abbreviates $((\forall x P) \wedge (\exists y Q)) \rightarrow R$. Our unconventional notation works well in proofs because typical formulæ have their quantifiers at the front.

Connectives and quantifiers are used to construct formulæ, not as a substitute for English. We may write ‘for all x , the formula $\forall y . \phi(x, y)$ is true.’

 *Background reading.* Most textbooks on discrete mathematics cover predicate logic and induction. Mattson (1993) has extensive coverage of both topics. Reeves and Clarke (1990) have little to say about induction but describe logic in detail, including a chapter on natural deduction. Winskel (1993) includes chapters on basic logic, induction and domain theory. Gunter (1992) describes domain theory and other advanced topics related to lazy evaluation, ML and polymorphism.

6.2 Mathematical induction and complete induction

Let us begin with a review of mathematical induction. Suppose $\phi(n)$ is a property that we should like to prove for all natural numbers n (all non-negative integers). To prove it by induction, it suffices to prove two things: the **base case**, namely $\phi(0)$, and the **induction step**, namely that $\phi(k)$ implies $\phi(k+1)$ for all k .

The rule can be displayed as follows:

$$\frac{\begin{array}{c} [\phi(k)] \\ \phi(0) \quad \phi(k+1) \end{array}}{\phi(n)} \quad \text{proviso: } k \text{ must not occur in} \\ \text{other assumptions of } \phi(k+1).$$

In this notation, the **premises** appear above the line and the **conclusion** below. These premises are the base case and the induction step. The formula $\phi(k)$, which appears in brackets, is the **induction hypothesis**; it may be assumed while proving $\phi(k+1)$. The proviso means that k must be a new variable, not already present in other induction hypotheses (or other assumptions); thus k stands for an arbitrary value. This rule notation comes from **natural deduction**, a formal theory of proofs.

In the induction step, we prove $\phi(k+1)$ under the assumption $\phi(k)$. We assume the very property we are trying to prove, but of k only. This may look like circular reasoning, especially since n and k are often the same variable (to avoid having to write the induction hypothesis explicitly). Why is induction sound? If the base case and induction step hold, then we have $\phi(0)$ by the base

case and $\phi(1), \phi(2), \dots$, by repeated use of the induction step. Therefore $\phi(n)$ holds for all n .

As a trivial example of induction, let us prove the following.

Theorem 1 *Every natural number is even or odd.*

Proof The inductive property is

n is even or n is odd

which we prove by induction on n .

The base case, 0 is even or 0 is odd, is trivial: 0 is even.

In the induction step, we assume the induction hypothesis

k is even or k is odd

and prove

$k + 1$ is even or $k + 1$ is odd.

By the induction hypothesis, there are two cases: if k is even then $k + 1$ is odd; if k is odd then $k + 1$ is even. Since the conclusion holds in both cases, the proof is finished. \square

Notice that a box marks the end of a proof.

This proof not only tells us that every natural number is even or odd, but contains a method for testing a given number. The test can be formalized in ML as a recursive function:

```
datatype evenodd = Even | Odd;
fun test 0 = Even
  | test n = (case test (n-1) of
               Even => Odd
               | Odd  => Even);
```

In some formal theories of constructive mathematics, a recursive function can be extracted automatically from every inductive proof. We shall not study such theories here, but shall try to learn as much as possible from our proofs. Mathematics would be barren indeed if each proof gave us nothing but a single formula. By sharpening the statement of the theorem, we can obtain more information from its proof.

Theorem 2 *Every natural number has the form $2m$ or $2m + 1$ for some natural number m .*

Proof Since this property is fairly complicated, let us express it in logical notation:

$$\exists m . n = 2m \vee n = 2m + 1$$

The proof is by induction on n .

The base case is

$$\exists m . 0 = 2m \vee 0 = 2m + 1.$$

This holds with $m = 0$ since $0 = 2 \times 0$.

For the induction step, assume the induction hypothesis

$$\exists m . k = 2m \vee k = 2m + 1$$

and show (renaming m as m' to avoid confusion)

$$\exists m' . k + 1 = 2m' \vee k + 1 = 2m' + 1.$$

By the induction hypothesis, there exists some m such that either $k = 2m$ or $k = 2m + 1$. In either case we can exhibit some m' such that $k + 1 = 2m'$ or $k + 1 = 2m' + 1$.

- If $k = 2m$ then $k + 1 = 2m + 1$, so $m' = m$.
- If $k = 2m + 1$ then $k + 1 = 2m + 2 = 2(m + 1)$, so $m' = m + 1$.

This concludes the proof. □

The function contained in this more detailed proof does not just test whether a number is even or odd, but also yields the quotient after division by two. This is enough information to reconstruct the original number. Thus, we have a means of checking the result.

```

fun half 0 = (Even, 0)
  | half n = (case half (n-1) of
              (Even, m) => (Odd, m)
              | (Odd, m) => (Even, m+1));

```

Complete induction. Mathematical induction reduces the problem $\phi(k)$ to the subproblem $\phi(k - 1)$, if $k > 0$. Complete induction reduces $\phi(k)$ to the k subproblems $\phi(0), \phi(1), \dots, \phi(k - 1)$. It includes mathematical induction as a special case.

To prove $\phi(n)$ for all integer $n \geq 0$ by complete induction on n , it suffices to prove the following induction step:

$$\phi(k) \text{ assuming } \forall i < k . \phi(i)$$

The induction step comprises an infinite sequence of statements:

$$\begin{aligned} &\phi(0) \\ &\phi(1) \text{ assuming } \phi(0) \\ &\phi(2) \text{ assuming } \phi(0) \text{ and } \phi(1) \\ &\phi(3) \text{ assuming } \phi(0), \phi(1) \text{ and } \phi(2) \\ &\vdots \end{aligned}$$

Clearly it implies $\phi(n)$ for all n ; complete induction is sound. The rule is portrayed as follows:

$$\frac{[\forall i < k . \phi(i)] \quad \phi(k)}{\phi(n)} \quad \text{proviso: } k \text{ must not occur in other assumptions of the premise.}$$

We now consider a simple proof.

Theorem 3 *Every natural number $n \geq 2$ can be written as a product of prime numbers, $n = p_1 \cdots p_k$.*

Proof By complete induction on n . There are two cases.


If n is prime then the result is trivial and $k = 1$.

If n is not prime then it is divisible by some natural number m such that $1 < m < n$. Since $m < n$ and $n/m < n$, we may appeal twice to the induction hypotheses of complete induction, writing these numbers as products of primes:

$$m = p_1 \cdots p_k \quad \text{and} \quad n/m = q_1 \cdots q_l$$

Now $n = m \times (n/m) = p_1 \cdots p_k q_1 \cdots q_l$. □

This is the easy part of the Fundamental Theorem of Arithmetic. The hard part is to show that the factorization into primes is unique, regardless of the choice of m in the proof (Davenport, 1952). As it stands, the proof provides a nondeterministic algorithm for factoring numbers into primes.

 *Proofs as programs.* There is a precise correspondence between constructive proofs and functional programs. If we can extract programs from proofs, then by proving theorems we obtain verified programs. Of course, not every proof is suitable. Not only must the proof be constructive (crucial parts of it at least), but it has to describe an efficient construction. The usual conception of the natural numbers corresponds to unary notation, which leads to hopelessly inefficient programs. The extracted programs contain computations that correspond to logical arguments; as they do not affect the result, they ought to be removed. Many people are investigating questions such as these. Thompson (1991) and Turner (1991) introduce this research area.

Exercise 6.1 Prove, by induction, the basic theorem of integer division: if n and d are natural numbers with $d \neq 0$, then there exist natural numbers q and r such that $n = dq + r$ and $0 \leq r < d$. Express the corresponding division function in ML. How efficient is it?

Exercise 6.2 Show that if $\phi(n)$ can be proved by mathematical induction on n , then it can also be proved by complete induction.

Exercise 6.3 Show that if $\phi(n)$ can be proved by complete induction on n , then it can also be proved using mathematical induction. (Hint: use a different induction formula.)

6.3 Simple examples of program verification

A *specification* is a precise description of the properties required of a program execution. It specifies the result of the computation, not the method. The specification of sorting states that the output contains the same elements as the input, arranged in increasing order. Any sorting algorithm satisfies this specification. A specification (for the present purposes, at least) says nothing about performance.

Program verification means proving that a program satisfies its specification. The complexity of a realistic specification makes verification difficult. Each of the programs verified below has a trivial specification: the result is a simple function of the input. We shall verify ML functions to compute factorials, Fibonacci numbers and powers.

The key step in these proofs is to formulate an induction suitable for the function. To be of any use, the induction hypothesis should be applicable to some recursive call of the function. The base case and induction step are simplified using function definitions, other mathematical laws and the induction hypothesis. If we are lucky, the simplified formula will be trivially true; if not, it may at least suggest a lemma to prove first.

Factorials. The iterative function *facti* is intended to compute factorials. Let us prove that $\text{facti}(n, 1) = n!$ for all $n \geq 0$. Recall that $0! = 1$ and $n! = (n - 1)! \times n$ for $n > 0$. The definition of *facti* was repeated in Section 6.1.

Induction on $\text{facti}(n, 1) = n!$ leads nowhere because it says nothing about argument p of *facti*. The induction hypothesis would be useless. We must find a relationship involving $\text{facti}(n, p)$ and $n!$ that implies $\text{facti}(n, 1) = n!$ and that can be proved by induction. A good try is $\text{facti}(n, p) = n! \times p$, but this will not quite do. It refers to some particular n and p , but p varies in the recursive

calls. The correct formulation has a universal quantifier:

$$\forall p . facti(n, p) = n! \times p$$

As an induction hypothesis about some fixed n , it asserts the equality for all p .

Theorem 4 For every natural number n , $facti(n, 1) = n!$

Proof This will follow by putting $p = 1$ in the following formula, which is proved by induction on n :

$$\forall p . facti(n, p) = n! \times p$$

By using n rather than k in the induction step, we can use this formula as the induction hypothesis.

For the base case we must show

$$\forall p . facti(0, p) = 0! \times p.$$

This holds because $facti(0, p) = p = 1 \times p = 0! \times p$.

For the induction step, the induction hypothesis is as stated above. We must show

$$\forall p . facti(n + 1, p) = (n + 1)! \times p.$$

Let us drop the universal quantifier and show the equality for arbitrary p . To simplify the equality, reduce the left side to the right side:

$$\begin{aligned} facti(n + 1, p) &= facti(n, (n + 1) \times p) && [facti] \\ &= n! \times ((n + 1) \times p) && [ind\ hyp] \\ &= (n! \times (n + 1)) \times p && [associativity] \\ &= (n + 1)! \times p && [factorial] \end{aligned}$$

The comments in brackets are read as follows:

[*facti*] means ‘by the definition of *facti*’
 [ind hyp] means ‘by the induction hypothesis’
 [associativity] means ‘by the associative law for \times ’
 [factorial] means ‘by the definition of factorials’

Both sides are equal in the induction step. Observe that the quantified variable p of the induction hypothesis is replaced by $(n + 1) \times p$. \square

Formal proofs should help us understand our programs. This proof explains the rôle of p in $facti(n, p)$. The induction formula is analogous to a **loop invariant** in procedural program verification. Since the proof depends on the associative law for \times , it suggests that $facti(n, 1)$ computes $n!$ by multiplying the same

numbers in a different order. Later we shall generalize this to a theorem about transforming recursive functions into iterative functions (Section 6.9). Many of our theorems concern implementing some function efficiently.

Fibonacci numbers. Recall that the Fibonacci sequence is defined by $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-2} + F_{n-1}$ for $n \geq 2$. We shall prove that they can be computed by the function *itfib* (from Section 2.15):

```
fun itfib (n, prev, curr) : int =
  if n=1 then curr
  else itfib (n-1, curr, prev+curr);
```

Observing that *itfib*(n , *prev*, *curr*) is defined for all $n \geq 1$, we set out to prove $\text{itfib}(n, 0, 1) = F_n$. As in the previous example, the induction formula must be generalized to say something about all the arguments of the function. There is no automatic procedure for doing this, but examining some computations of $\text{itfib}(n, 0, 1)$ reveals that *prev* and *curr* are always Fibonacci numbers. This suggests the relationship

$$\text{itfib}(n, F_k, F_{k+1}) = F_{k+n}.$$

Again, a universal quantifier must be inserted before induction.

Theorem 5 For every integer $n \geq 1$, $\text{itfib}(n, 0, 1) = F_n$.

Proof Put $k = 0$ in the following formula, which is proved by induction on n :

$$\forall k. \text{itfib}(n, F_k, F_{k+1}) = F_{k+n}$$

Since $n \geq 1$, the base case is to prove this for $n = 1$:

$$\forall k. \text{itfib}(1, F_k, F_{k+1}) = F_{k+1}$$

This is immediate by the definition of *itfib*.

For the induction step, the induction hypothesis is given above; we must show

$$\forall k. \text{itfib}(n + 1, F_k, F_{k+1}) = F_{k+(n+1)}.$$

We prove this by simplifying the left side:

$$\begin{aligned} & \text{itfib}(n + 1, F_k, F_{k+1}) \\ &= \text{itfib}(n, F_{k+1}, F_k + F_{k+1}) && [\text{itfib}] \\ &= \text{itfib}(n, F_{k+1}, F_{k+2}) && [\text{Fibonacci}] \\ &= F_{(k+1)+n} && [\text{ind hyp}] \\ &= F_{k+(n+1)} && [\text{arithmetic}] \end{aligned}$$

The induction hypothesis is applied with $k + 1$ in place of k , instantiating the quantifier. \square

This proof shows how pairs of Fibonacci numbers are generated successively. The induction formula is a key property of *itfib*, and is not at all obvious. It is good practice to state such a formula as a comment by the function declaration.

Powers. We now prove that $power(x, k) = x^k$ for every real number x and integer $k \geq 1$. Recall the definition of *power* (Section 2.14):

```
fun power(x, k) : real =
  if k=1 then x
  else if k mod 2 = 0 then   power(x*x, k div 2)
                        else x * power(x*x, k div 2);
> val power = fn : real * int -> real
```

The proof will assume that ML's real arithmetic is exact, ignoring roundoff errors. It is typical of program verification to ignore the limitations of physical hardware. To demonstrate that *power* is suitable for actual computers would require an error analysis as well, which would involve much more work.

We must check that $power(x, k)$ is defined for $k \geq 1$. The case $k = 1$ is obvious. If $k \geq 2$ then we need to examine the recursive calls, which replace k by $k \text{ div } 2$. These terminate because $1 \leq k \text{ div } 2 < k$.

Since x varies during the computation of $power(x, k)$, the induction formula must have a quantifier:

$$\forall x . power(x, k) = x^k$$

However, ordinary mathematical induction is not appropriate. In $power(x, k)$ the recursive call replaces k by $k \text{ div } 2$, not $k - 1$. We use complete induction in order to have an induction hypothesis for $k \text{ div } 2$.

Theorem 6 For every integer $k \geq 1$, $\forall x . power(x, k) = x^k$.

Proof The formula is proved by complete induction on k .

Although complete induction has no separate base case, we may perform case analysis on k . Since $k \geq 1$, let us consider $k = 1$ and $k \geq 2$ separately.

Case $k = 1$. We must prove

$$\forall x . power(x, 1) = x^1.$$

This holds because $power(x, 1) = x = x^1$.

Case $k \geq 2$. We consider subcases. If k is even then $k = 2j$, and if k is odd then $k = 2j + 1$, for some integer j (namely $k \text{ div } 2$). In both cases $1 \leq j < k$,

so there is an induction hypothesis for j :

$$\forall x . power(x, j) = x^j$$

If $k = 2j$ then $k \bmod 2 = 0$ and

$$\begin{aligned} power(x, 2j) &= power(x^2, j) && [power] \\ &= (x^2)^j && [ind\ hyp] \\ &= x^{2j}. && [arithmetic] \end{aligned}$$

If $k = 2j + 1$ then $k \bmod 2 = 1$ and

$$\begin{aligned} power(x, 2j + 1) &= x \times power(x^2, j) && [power] \\ &= x \times (x^2)^j && [ind\ hyp] \\ &= x^{2j+1}. && [arithmetic] \end{aligned}$$

In both of these cases, the induction hypothesis is applied with x^2 in place of x . □

Exercise 6.4 Verify that *introot* computes integer square roots (Section 2.16).

Exercise 6.5 Recall *sqroot* of Section 2.17, which computes real square roots by the Newton-Raphson method. Discuss the problems involved in verifying this function.

Structural induction

Mathematical induction establishes $\phi(n)$ for all natural numbers n by considering how a natural number is constructed. Although there are infinitely many natural numbers, they are constructed in just two ways:

- 0 is a number.
- If k is a number then so is $k + 1$.

Strictly speaking, we should introduce the successor function *suc* and reformulate the above:

- If k is a number then so is *suc*(k).

Addition and other arithmetic functions are then defined recursively in terms of 0 and *suc*, which are essentially the constructors of an ML datatype. **Structural induction** is a generalization of mathematical induction to datatypes such as lists and trees.

6.4 Structural induction on lists

Suppose $\phi(xs)$ is a property that we should like to prove for all lists xs . Let xs have type τ *list* for some type τ . To prove $\phi(xs)$ by structural induction, it suffices to prove two premises:

- The base case is $\phi([])$.
- The induction step is that $\phi(ys)$ implies $\phi(y :: ys)$ for all y of type τ and ys of type τ *list*. The induction hypothesis is $\phi(ys)$.

The rule can be displayed as follows:

$$\frac{[\phi(ys)] \quad \phi([]) \quad \phi(y :: ys)}{\phi(xs)} \quad \text{proviso: } y \text{ and } ys \text{ must not occur in other assumptions of } \phi(y :: ys).$$

Why is structural induction sound? We have $\phi([])$ by the base case. By the induction step, we have $\phi([y])$ for all y ; the conclusion holds for all 1-element lists. Using the induction step again, the conclusion holds for all 2-element lists. Continuing this process, the conclusion $\phi(xs)$ holds for every n -element list xs ; all lists are reached eventually. The rule can also be justified by mathematical induction on the length of the list.

To illustrate the rule, let us prove a fundamental property of lists.

Theorem 7 *No list equals its own tail.*

Proof The statement of the theorem can be formalized as follows:

$$\forall x . x :: xs \neq xs$$

This is proved by structural induction on the list xs .

The base case, $\forall x . [x] \neq []$, is trivial by the definition of equality on lists. Two lists are equal if they have the same length and the corresponding elements are equal.

In the induction step, assume the induction hypothesis

$$\forall x . x :: ys \neq ys$$

and show (for arbitrary y and ys)

$$\forall x . x :: (y :: ys) \neq y :: ys$$

By the definition of list equality, it is enough to show that the tails differ: to show $y :: ys \neq ys$. This follows by the induction hypothesis, putting y for the quantified variable x . Again, the quantifier in the induction formula is essential.

□

This theorem does not apply to infinite lists, for $[1,1,1,\dots]$ equals its own tail. The structural induction rules given here are sound for finite objects only. In domain theory, induction can be extended to infinite lists — but not for arbitrary formulæ! The restrictions are complicated; roughly speaking, the conclusion holds for infinite lists only if the induction formula is a conjunction of equations. So $x :: xs \neq xs$ cannot be proved for infinite lists.

Let us prove theorems about some of the list functions of Chapter 3. Each of these functions terminates for all arguments because each recursive call involves a shorter list.

The length of a list:

```
fun nlength [] = 0
  | nlength (x::xs) = 1 + nlength xs;
```

The infix operator @, which appends two lists:

```
fun [] @ ys = ys
  | (x::xs) @ ys = x :: (xs@ys);
```

The naïve reverse function:

```
fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x];
```

An efficient reverse function:

```
fun revAppend ([], ys) = ys
  | revAppend (x::xs, ys) = revAppend (xs, x::ys);
```

Length and append. Here is an obvious property about the length of the concatenation of two lists.

Theorem 8 For all lists xs and ys , $nlength(xs@ys) = nlength\ xs + nlength\ ys$.

Proof By structural induction on xs . We avoid renaming this variable; thus, the formula above also serves as the induction hypothesis.

The base case is

$$nlength([] @ ys) = nlength[] + nlength\ ys.$$

This holds because

$$\begin{aligned} nlength([] @ ys) &= nlength\ ys && \text{[@]} \\ &= 0 + nlength\ ys && \text{[arithmetic]} \\ &= nlength[] + nlength\ ys. && \text{[nlength]} \end{aligned}$$

For the induction step, assume the induction hypothesis and show, for all x and xs , that

$$nlength((x :: xs) @ ys) = nlength(x :: xs) + nlength ys.$$

This holds because

$$\begin{aligned} nlength((x :: xs) @ ys) & \\ &= nlength(x :: (xs @ ys)) && \text{[@]} \\ &= 1 + nlength(xs @ ys) && \text{[nlength]} \\ &= 1 + (nlength xs + nlength ys) && \text{[ind hyp]} \\ &= (1 + nlength xs) + nlength ys && \text{[associativity]} \\ &= nlength(x :: xs) + nlength ys. && \text{[nlength]} \end{aligned}$$

We could have written $1 + nlength xs + nlength ys$, omitting parentheses, instead of applying the associative law explicitly. \square

The proof brings out the correspondence between inserting the list elements and counting them. Induction on xs works because the base case and induction step can be simplified using function definitions. Induction on ys leads nowhere: try it.

Efficient list reversal. The function $nrev$ is a mathematical definition of list reversal, while $revAppend$ reverses lists efficiently. The proof that they are equivalent is similar to Theorem 4, the correctness of *facti*. In both proofs, the induction formula is universally quantified over an accumulating argument.

Theorem 9 For every list xs , we have $\forall ys. revAppend(xs, ys) = nrev(xs) @ ys$.

Proof By structural induction on xs , taking the formula above as the induction hypothesis. The base case is

$$\forall ys. revAppend([], ys) = nrev[] @ ys.$$

It holds because $revAppend([], ys) = ys = [] @ ys = nrev[] @ ys$.

The induction step is to show, for arbitrary x and xs , the formula

$$\forall ys. revAppend(x :: xs, ys) = nrev(x :: xs) @ ys.$$

Simplifying the right side of the equality yields

$$nrev(x :: xs) @ ys = (nrev(xs) @ [x]) @ ys. \quad \text{[nrev]}$$

Simplifying the left side yields

$$\begin{aligned}
 \text{revAppend}(x :: xs, ys) &= \text{revAppend}(xs, x :: ys) && [\text{revAppend}] \\
 &= \text{nrev}(xs) @ (x :: ys) && [\text{ind hyp}] \\
 &= \text{nrev}(xs) @ ([x] @ ys). && [@]
 \end{aligned}$$

The induction hypothesis is applied with $x :: xs$ for the quantified variable ys .

Are we finished? Not quite: the parentheses do not agree. It remains to show

$$\text{nrev}(xs) @ ([x] @ ys) = (\text{nrev}(xs) @ [x]) @ ys.$$

This formula looks more complicated than the one we set out to prove. How shall we proceed? Observe that the formula is a special case of something simple and plausible: that $@$ is associative. We have only to prove

$$l_1 @ (l_2 @ l_3) = (l_1 @ l_2) @ l_3.$$

This routine induction is left as an exercise. □

It would be tidier to prove each theorem in the correct order, making a flawless presentation. This example attempts to show how the need for a theorem is discovered. The hardest problem in a verification is recognizing what properties ought to be proved. The need here for the associative law may be obvious — but not if we are dazzled by the symbols, which happens all too easily.

Append and reverse. We now prove a relationship involving list concatenation and reversal.

Theorem 10 *For all lists xs and ys , $\text{nrev}(xs @ ys) = \text{nrev } ys @ \text{nrev } xs$.*

Proof By structural induction on xs . The base case is

$$\text{nrev}([] @ ys) = \text{nrev } ys @ \text{nrev}[].$$

This holds using the lemma $l @ [] = l$, which is left as an exercise.

The induction step is

$$\text{nrev}((x :: xs) @ ys) = \text{nrev } ys @ \text{nrev}(x :: xs).$$

This holds because

$$\begin{aligned}
 \text{nrev}((x :: xs) @ ys) &= \text{nrev}(x :: (xs @ ys)) && [@] \\
 &= \text{nrev}(xs @ ys) @ [x] && [\text{nrev}] \\
 &= \text{nrev } ys @ \text{nrev } xs @ [x] && [\text{ind hyp}] \\
 &= \text{nrev } ys @ \text{nrev}(x :: xs). && [\text{nrev}]
 \end{aligned}$$

In $nrev\ ys\ @\ nrev\ xs\ @\ [x]$ we have implicitly applied the associativity of $@$ by omitting parentheses. \square

These last two theorems show that $nrev$, though inefficient to compute, is a good specification of reversal. It permits simple proofs. A literal specification, like

$$reverse[x_1, x_2, \dots, x_n] = [x_n, \dots, x_2, x_1],$$

would be most difficult to formalize. The function $revAppend$ is not a good specification either; its performance is irrelevant and it is too complicated. But $nlength$ is a good specification of the length of a list.

Exercise 6.6 Prove $xs\ @\ [] = xs$ for every list xs , by structural induction.

Exercise 6.7 Prove $l_1\ @\ (l_2\ @\ l_3) = (l_1\ @\ l_2)\ @\ l_3$ for all lists l_1, l_2 and l_3 , by structural induction.

Exercise 6.8 Prove $nrev(nrev\ xs) = xs$ for every list xs .

Exercise 6.9 Show that $nlength\ xs = length\ xs$ for every list xs . (The function $length$ was defined in Section 3.4.)

6.5 Structural induction on trees

In Chapter 4 we studied binary trees defined as follows:

```
datatype 'a tree = Lf
                | Br of 'a * 'a tree * 'a tree;
```

Binary trees admit structural induction. In most respects, their treatment resembles that of lists. Suppose $\phi(t)$ is a property of trees, where t has type $\tau\ tree$. To prove $\phi(t)$ by structural induction, it suffices to prove two premises:

- The base case is $\phi(Lf)$.
- The induction step is to show that $\phi(t_1)$ and $\phi(t_2)$ imply $\phi(Br(x, t_1, t_2))$ for all x of type τ and t_1, t_2 of type $\tau\ tree$. There are two induction hypotheses: $\phi(t_1)$ and $\phi(t_2)$.

The rule can be portrayed thus:

$$\frac{\phi(Lf) \quad \phi(Br(x, t_1, t_2))}{\phi(t)} \quad \text{proviso: } x, t_1 \text{ and } t_2 \text{ must not occur in other assumptions of } \phi(Br(x, t_1, t_2)).$$

This structural induction rule is sound because it covers all the ways of building a tree. The base case establishes $\phi(Lf)$. Applying the induction step once establishes $\phi(Br(x, Lf, Lf))$ for all x , covering all trees containing one Br node. Applying the induction step twice establishes $\phi(t)$ where t is any tree containing two Br nodes. Further applications of the induction step cover larger trees.

We can also justify the rule by complete induction on the number of labels in the tree, because every tree is finite and its subtrees are smaller than itself. Structural induction is not sound in general for infinite trees.

We shall prove some facts about the following functions on binary trees, from Section 4.10.

The number of labels in a tree:

```
fun size Lf = 0
  | size (Br (v, t1, t2)) = 1 + size t1 + size t2;
```

The depth of a tree:

```
fun depth Lf = 0
  | depth (Br (v, t1, t2)) = 1 + Int.max (depth t1, depth t2);
```

Reflection of a tree:

```
fun reflect Lf = Lf
  | reflect (Br (v, t1, t2)) = Br (v, reflect t2, reflect t1);
```

The preorder listing of a tree's labels:

```
fun preorder Lf = []
  | preorder (Br (v, t1, t2)) = [v] @ preorder t1 @ preorder t2;
```

The postorder listing of a tree's labels:

```
fun postorder Lf = []
  | postorder (Br (v, t1, t2)) = postorder t1 @ postorder t2 @ [v];
```

Double reflection. We begin with an easy example: reflecting a tree twice yields the original tree.

Theorem 11 For every binary tree t , $reflect(reflect t) = t$.

Proof By structural induction on t . The base case is

$$reflect(reflect Lf) = Lf.$$

This holds by the definition of $reflect$: $reflect(reflect Lf) = reflect Lf = Lf$.

For the induction step we have the two induction hypotheses

$$\text{reflect}(\text{reflect } t_1) = t_1 \quad \text{and} \quad \text{reflect}(\text{reflect } t_2) = t_2$$

and must show

$$\text{reflect}(\text{reflect}(Br(x, t_1, t_2))) = Br(x, t_1, t_2).$$

Simplifying,

$$\begin{aligned} & \text{reflect}(\text{reflect}(Br(x, t_1, t_2))) \\ &= \text{reflect}(Br(x, \text{reflect } t_2, \text{reflect } t_1)) && [\text{reflect}] \\ &= Br(x, \text{reflect}(\text{reflect } t_1), \text{reflect}(\text{reflect } t_2)) && [\text{reflect}] \\ &= Br(x, t_1, \text{reflect}(\text{reflect } t_2)) && [\text{ind hyp}] \\ &= Br(x, t_1, t_2). && [\text{ind hyp}] \end{aligned}$$

Both induction hypotheses have been applied. We can observe the two calls of *reflect* cancelling each other. \square

Preorder and postorder. If the concepts of preorder and postorder are obscure to you, then the following theorem may help. A key fact is Theorem 10, concerning *nrev* and @, which we have recently proved.

Theorem 12 *For every binary tree t , $\text{postorder}(\text{reflect } t) = \text{nrev}(\text{preorder } t)$.*

Proof By structural induction on t . The base case is

$$\text{postorder}(\text{reflect } Lf) = \text{nrev}(\text{preorder } Lf).$$

This is routine; both sides are equal to [].

For the induction step we have the induction hypotheses

$$\begin{aligned} \text{postorder}(\text{reflect } t_1) &= \text{nrev}(\text{preorder } t_1) \\ \text{postorder}(\text{reflect } t_2) &= \text{nrev}(\text{preorder } t_2) \end{aligned}$$

and must show

$$\text{postorder}(\text{reflect}(Br(x, t_1, t_2))) = \text{nrev}(\text{preorder}(Br(x, t_1, t_2))).$$

First, we simplify the right-hand side:

$$\begin{aligned} & \text{nrev}(\text{preorder}(Br(x, t_1, t_2))) \\ &= \text{nrev}([x] @ \text{preorder } t_1 @ \text{preorder } t_2) && [\text{preorder}] \\ &= \text{nrev}(\text{preorder } t_2) @ \text{nrev}(\text{preorder } t_1) @ \text{nrev}[x] && [\text{Theorem 10}] \\ &= \text{nrev}(\text{preorder } t_2) @ \text{nrev}(\text{preorder } t_1) @ [x] && [\text{nrev}] \end{aligned}$$

Some steps have been skipped. Theorem 10 has been applied twice, to both occurrences of @, and $nrev[x]$ is simplified directly to $[x]$.

Now we simplify the left-hand side:

$$\begin{aligned}
& postorder(reflect(Br(x, t_1, t_2))) \\
&= postorder(Br(x, reflect t_2, reflect t_1)) && [reflect] \\
&= postorder(reflect t_2) @ postorder(reflect t_1) @ [x] && [postorder] \\
&= nrev(preorder t_2) @ nrev(preorder t_1) @ [x] && [ind hyp]
\end{aligned}$$

Thus, both sides are equal. \square

Count and depth. We now prove a law relating the number of labels in a binary tree to its depth. The theorem is an inequality, reminding us that formal methods involve more than mere equations.

Theorem 13 For every binary tree t , $size\ t \leq 2^{depth\ t} - 1$.

Proof By structural induction on t . The base case is

$$size\ Lf \leq 2^{depth\ Lf} - 1.$$

It holds because $size\ Lf = 0 = 2^0 - 1 = 2^{depth\ Lf} - 1$.

In the induction step the induction hypotheses are

$$size\ t_1 \leq 2^{depth\ t_1} - 1 \quad \text{and} \quad size\ t_2 \leq 2^{depth\ t_2} - 1$$

and we must demonstrate

$$size(Br(x, t_1, t_2)) \leq 2^{depth(Br(x, t_1, t_2))} - 1.$$

First, simplify the right-hand side:

$$\begin{aligned}
2^{depth(Br(x, t_1, t_2))} - 1 &= 2^{1+\max(depth\ t_1, depth\ t_2)} - 1 && [depth] \\
&= 2 \times 2^{\max(depth\ t_1, depth\ t_2)} - 1 && [arithmetic]
\end{aligned}$$

Next, show that the left side is less than or equal to this:

$$\begin{aligned}
size(Br(x, t_1, t_2)) &= 1 + size\ t_1 + size\ t_2 && [size] \\
&\leq 1 + (2^{depth\ t_1} - 1) + (2^{depth\ t_2} - 1) && [ind\ hyp] \\
&= 2^{depth\ t_1} + 2^{depth\ t_2} - 1 && [arithmetic] \\
&\leq 2 \times 2^{\max(depth\ t_1, depth\ t_2)} - 1 && [arithmetic]
\end{aligned}$$

Here we have identified \max , the mathematical function for the maximum of two integers, with the library function $Int.\ max$. \square



Problematical datatypes. Our simple methods do not admit all ML datatypes. Consider this declaration:

```
datatype lambda = F of lambda -> lambda;
```

The mathematics in this chapter is based on set theory. Since there is no set A that is isomorphic to the set of functions $A \rightarrow A$, we can make no sense of this declaration. In domain theory, this declaration can be interpreted because there is a domain D isomorphic to $D \rightarrow D$, which is the domain of *continuous* functions from D to D . Even in domain theory, no induction rule useful for reasoning about D is known. This is because the type definition involves recursion to the left of the function arrow (\rightarrow). We shall not consider datatypes involving functions.

The declaration of type *term* (Section 5.11) refers to lists:

```
datatype term = Var of string
              | Fun of string * term list;
```

Type *term* denotes a set of finite terms and satisfies a structural induction rule. However, the involvement of lists in the type complicates the theory and proofs (Paulson, 1995, Section 4.4).

Exercise 6.10 Formalize and prove: *No binary tree equals its own left subtree.*

Exercise 6.11 Prove $size(reflect\ t) = size\ t$ for every binary tree t .

Exercise 6.12 Prove $nlength(preorder\ t) = size\ t$ for every binary tree t .

Exercise 6.13 Prove $nrev(inorder(reflect\ t)) = inorder\ t$ for every binary tree t .

Exercise 6.14 Define a function *leaves* to count the *Lf* nodes in a binary tree. Then prove $leaves\ t = size\ t + 1$ for all t .

Exercise 6.15 Verify the function *preord* of Section 4.11. In other words, prove $preord(t, []) = preorder\ t$ for every binary tree t .

6.6 Function values and functionals

Our mathematical methods extend directly to proofs about higher-order functions (functionals). The notion of ‘functions as values’ is familiar to mathematicians. In set theory, for example, functions are sets and are treated no differently from other sets.

We can prove many facts about functionals without using any additional rules. The laws of the λ -calculus could be introduced for reasoning about ML’s `fn` notation, although this will not be done here. Our methods, needless to say, apply only to pure functions — not to ML functions with side effects.

Equality of functions. The **law of extensionality** states that functions f and g are equal if $f(x) = g(x)$ for all x (of suitable type). For instance, these three doubling functions are extensionally equal:

```
fun double1 (n) = 2*n;
fun double2 (n) = n*2;
fun double3 (n) = (n-1)+(n+1);
```

The extensionality law is valid because the only operation that can be performed on an ML function is application to an argument. Replacing f by g , if these functions are extensionally equal, does not affect the value of any application of f .¹

A different concept of equality, called **intensional equality**, regards two functions as equal only if their definitions are identical. Our three doubling functions are all distinct under intensional equality. This concept resembles function equality in Lisp, where a function value is a piece of Lisp code that can be taken apart.

There is no general, computable method of testing whether two functions are extensionally equal. Therefore ML has no equality test for function values. Lisp tests equality of functions by comparing their internal representations.

We now prove a few statements about function composition (the infix o) and the functional map (of Section 5.7).

```
fun (f o g) x = f (g x);
fun map f []      = []
  | map f (x::xs) = (f x) :: map f xs;
```

The associativity of composition. Our first theorem is trivial. It asserts that function composition is associative.

Theorem 14 *For all functions f , g and h (of appropriate type),*

$$(f \circ g) \circ h = f \circ (g \circ h).$$

Proof By the law of extensionality, it is enough to show

$$((f \circ g) \circ h) x = (f \circ (g \circ h)) x$$

¹ The extensionality law relies on our global assumption that functions terminate. ML distinguishes \perp (the undefined function value) from $\lambda x.\perp$ (the function that never terminates when applied) although both functions yield \perp when applied to any argument.

for all x . This holds because

$$\begin{aligned} ((f \circ g) \circ h) x &= (f \circ g)(h x) \\ &= f(g(h x)) \\ &= f((g \circ h) x) \\ &= (f \circ (g \circ h)) x. \end{aligned}$$

Each step holds by the definition of composition. \square

As stated, the theorem holds only for functions of appropriate type; the equation must be properly typed. Typing restrictions apply to all our theorems and will not be mentioned again.

The list functional map. Functionals enjoy many laws. Here is a theorem about *map* and composition that can be used to avoid computing intermediate lists.

Theorem 15 For all functions f and g , $(\text{map } f) \circ (\text{map } g) = \text{map } (f \circ g)$.

Proof By the extensionality law, this equality holds if

$$((\text{map } f) \circ (\text{map } g)) xs = \text{map } (f \circ g) xs$$

for all xs . Using the definition of \circ , this can be simplified to

$$\text{map } f (\text{map } g xs) = \text{map } (f \circ g) xs$$

Since xs is a list, we may use structural induction. This formula will also be our induction hypothesis. The base case is

$$\text{map } f (\text{map } g []) = \text{map } (f \circ g) [].$$

It holds because both sides equal $[]$:

$$\text{map } f (\text{map } g []) = \text{map } f [] = [] = \text{map } (f \circ g) []$$

For the induction step, we assume the induction hypothesis and show (for arbitrary x and xs)

$$\text{map } f (\text{map } g (x :: xs)) = \text{map } (f \circ g) (x :: xs).$$

Straightforward reasoning yields

$$\begin{aligned}
& \text{map } f \text{ (map } g \text{ (} x :: xs \text{))} \\
&= \text{map } f \text{ ((} g \text{ } x \text{) :: (map } g \text{ } xs \text{))} && \text{[map]} \\
&= f(g \text{ } x) :: (\text{map } f \text{ (map } g \text{ } xs \text{)}) && \text{[map]} \\
&= f(g \text{ } x) :: (\text{map } (f \circ g) \text{ } xs) && \text{[ind hyp]} \\
&= (f \circ g)(x) :: (\text{map } (f \circ g) \text{ } xs) && \text{[o]} \\
&= \text{map } (f \circ g) \text{ (} x :: xs \text{)}. && \text{[map]}
\end{aligned}$$

Despite the presence of function values, the proof is a routine structural induction. \square

The list functional foldl. The functional *foldl* applies a 2-argument function over the elements of a list. Recall its definition from Section 5.10:

$$\begin{aligned}
\text{fun foldl } f \text{ } e \text{ []} &= e \\
| \text{foldl } f \text{ } e \text{ (} x :: xs \text{)} &= \text{foldl } f \text{ (} f(x, e) \text{) } xs;
\end{aligned}$$

If \oplus is an associative operator then $\text{foldl } (\text{op}\oplus) \text{ (} y \oplus z \text{) } xs = (\text{foldl } (\text{op}\oplus) \text{ } y \text{ } xs) \oplus z$. For if $xs = [x_1, x_2, \dots, x_n]$, this is equivalent to

$$x_n \oplus \dots (x_2 \oplus (x_1 \oplus (y \oplus z))) \dots = x_n \oplus \dots (x_2 \oplus (x_1 \oplus y)) \oplus z.$$

Since \oplus is associative we may erase the parentheses, reducing both sides to $x_n \oplus \dots x_2 \oplus x_1 \oplus y \oplus z$. We see the notational advantage of working with an infix operator \oplus instead of a function f . Now let us see the formal proof.

Theorem 16 *Suppose \oplus is an infix operator that is associative, satisfying $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ for all x, y and z . Then for all y, z and xs ,*

$$\forall y. \text{foldl } (\text{op}\oplus) \text{ (} y \oplus z \text{) } xs = (\text{foldl } (\text{op}\oplus) \text{ } y \text{ } xs) \oplus z.$$

Proof By structural induction on the list xs . The base case,

$$\text{foldl } (\text{op}\oplus) \text{ (} y \oplus z \text{) []} = (\text{foldl } (\text{op}\oplus) \text{ } y \text{ []}) \oplus z,$$

is trivial; both sides reduce to $y \oplus z$.

For the induction step we show

$$\text{foldl } (\text{op}\oplus) \text{ (} y \oplus z \text{) (} x :: xs \text{)} = (\text{foldl } (\text{op}\oplus) \text{ } y \text{ (} x :: xs \text{)}) \oplus z.$$

for arbitrary y , x and xs :

$$\begin{aligned}
 & \text{foldl } (\text{op} \oplus) (y \oplus z) (x :: xs) \\
 &= \text{foldl } (\text{op} \oplus) (x \oplus (y \oplus z)) xs && \text{[foldl]} \\
 &= \text{foldl } (\text{op} \oplus) ((x \oplus y) \oplus z) xs && \text{[associativity]} \\
 &= (\text{foldl } (\text{op} \oplus) (x \oplus y) xs) \oplus z && \text{[ind hyp]} \\
 &= (\text{foldl } (\text{op} \oplus) y (x :: xs)) \oplus z && \text{[foldl]}
 \end{aligned}$$

The induction hypothesis has been applied with $x \oplus y$ replacing the quantified variable y . \square

Exercise 6.16 Prove $\text{map } f (xs @ ys) = (\text{map } f xs) @ (\text{map } f ys)$.

Exercise 6.17 Prove $(\text{map } f) \circ \text{nrev} = \text{nrev} \circ (\text{map } f)$.

Exercise 6.18 Declare a functional maptree on binary trees, satisfying the following equations (which you should prove):

$$\begin{aligned}
 (\text{maptree } f) \circ \text{reflect} &= \text{reflect} \circ (\text{maptree } f) \\
 (\text{map } f) \circ \text{preorder} &= \text{preorder} \circ (\text{maptree } f)
 \end{aligned}$$

Exercise 6.19 Prove $\text{foldr } (\text{op} ::) ys xs = xs @ ys$.

Exercise 6.20 Prove $\text{foldl } f z (xs @ ys) = \text{foldl } f (\text{foldl } f z xs) ys$.

Exercise 6.21 Suppose that \odot and e satisfy, for all x , y and z ,

$$x \odot (y \odot z) = (x \odot y) \odot z \quad \text{and} \quad e \odot x = x.$$

Let F abbreviate $\text{foldr } (\text{op} \odot)$. Prove that for all y and l , $(F e l) \odot y = F y l$.

Exercise 6.22 Let \odot , e and F be as in the previous exercise. Define the function G by $G(l, z) = F z l$. Prove that for all ls , $\text{foldr } G e ls = F e (\text{map } (F e) ls)$.

A general induction principle

In a structural induction proof on lists, we assume $\phi(xs)$ and then show $\phi(x :: xs)$. Typically the induction formula involves a recursive list function such as nrev . The induction hypothesis, $\phi(xs)$, says something about $\text{nrev}(xs)$. Since $\text{nrev}(x :: xs)$ is defined in terms of $\text{nrev}(xs)$, we can reason about $\text{nrev}(x :: xs)$ to show $\phi(x :: xs)$.

The list function *nrev* makes its recursive call on the tail of its argument. This kind of recursion is called **structural recursion** by analogy with structural induction. However, recursive functions can shorten the list in other ways. The function *maxl*, when applied to $m :: n :: ns$, may call itself on $m :: ns$:

```
fun maxl [m] : int = m
  | maxl (m::n::ns) = if m>n then maxl(m::ns)
                      else maxl(n::ns);
```

Quick sort and merge sort divide a list into two smaller lists and sort them recursively. Matrix transpose (Section 3.9) and Gaussian elimination make recursive calls on a smaller matrix obtained by deleting rows and columns.

Most functions on trees use structural recursion: their recursive calls involve a node's immediate subtrees. The function *nnf*, which converts a proposition into negation normal form, is not structurally recursive. We shall prove theorems about *nnf* in this section.

Structural induction works best with functions that are structurally recursive. With other functions, **well-founded induction** is often superior. Well-founded induction is a powerful generalization of complete induction. Because the rule is abstract and seldom required in full generality, our proofs will be done by a special case: induction on size. For instance, the function *nlength* formalizes the size of a list. In the induction step we have to prove $\phi(xs)$ under the induction hypothesis

$$\forall ys. \text{nlength } ys < \text{nlength } xs \rightarrow \phi(ys).$$

Thus, we may assume $\phi(ys)$ provided *ys* is a shorter list than *xs*.

6.7 Computing normal forms

Our tautology checker uses functions to compute normal forms of propositions (Section 4.19). These functions involve unusual recursions; structural induction seems inappropriate. First, let us recall some definitions.

The declaration of *prop*, the datatype of propositions, is

```
datatype prop = Atom of string
              | Neg   of prop
              | Conj  of prop * prop
              | Disj  of prop * prop;
```

Function *nnf* computes the negation normal form of a proposition. It is practically a literal rendering of the rewrite rules for this normal form, and has complex patterns.

```

fun nnf (Atom a)           = Atom a
  | nnf (Neg (Atom a))     = Neg (Atom a)
  | nnf (Neg (Neg p))      = nnf p
  | nnf (Neg (Conj (p, q))) = nnf (Disj (Neg p, Neg q))
  | nnf (Neg (Disj (p, q))) = nnf (Conj (Neg p, Neg q))
  | nnf (Conj (p, q))      = Conj (nnf p, nnf q)
  | nnf (Disj (p, q))      = Disj (nnf p, nnf q);

```

The mutually recursive functions *nnfpos* and *nnfneg* compute the same normal form, but more efficiently:

```

fun nnfpos (Atom a)       = Atom a
  | nnfpos (Neg p)        = nnfneg p
  | nnfpos (Conj (p, q)) = Conj (nnfpos p, nnfpos q)
  | nnfpos (Disj (p, q)) = Disj (nnfpos p, nnfpos q)
and nnfneg (Atom a)      = Neg (Atom a)
  | nnfneg (Neg p)       = nnfpos p
  | nnfneg (Conj (p, q)) = Disj (nnfneg p, nnfneg q)
  | nnfneg (Disj (p, q)) = Conj (nnfneg p, nnfneg q);

```

We must verify that these functions terminate. The functions *nnfpos* and *nnfneg* are structurally recursive — recursion is always applied to an immediate constituent of the argument — and therefore terminate. For *nnf*, termination is not so obvious. Consider *nnf(Neg(Conj(p, q)))*, which makes a recursive call on a large expression. But this reduces in a few steps to

$$\text{Disj}(\text{nnf}(\text{Neg } p), \text{nnf}(\text{Neg } q)).$$

Thus the recursive calls after *Neg(Conj(p, q))* involve the smaller propositions *Neg p* and *Neg q*. The other complicated pattern, *Neg(Disj(p, q))*, behaves similarly. In every case, recursive computations in *nnf* involve smaller and smaller propositions, and therefore terminate.

Let us prove that *nnfpos* and *nnf* are equal. The termination argument suggests that theorems involving *nnf p* should be proved by induction on the size of *p*. Let us write *nodes(p)* for the number of *Neg*, *Conj* and *Disj* nodes in *p*. This function can easily be coded in ML.

Theorem 17 *For all propositions p, nnf p = nnfpos p.*

Proof By mathematical induction on *nodes(p)*, taking as induction hypotheses *nnf q = nnfpos q* for all *q* such that *nodes(q) < nodes(p)*. We consider seven cases, corresponding to the definition of *nnf*.

If *p = Atom a* then *nnf(Atom a) = Atom a = nnfpos(Atom a)*.

If *p = Neg(Atom a)* then

$$\text{nnf}(\text{Neg}(\text{Atom } a)) = \text{Neg}(\text{Atom } a) = \text{nnfpos}(\text{Neg}(\text{Atom } a)).$$

If $p = \text{Conj}(r, q)$ then

$$\begin{aligned} \text{nnf}(\text{Conj}(r, q)) &= \text{Conj}(\text{nnf } r, \text{nnf } q) && [\text{nnf}] \\ &= \text{Conj}(\text{nnfpos } r, \text{nnfpos } q) && [\text{ind hyp}] \\ &= \text{nnfpos}(\text{Conj}(r, q)). && [\text{nnfpos}] \end{aligned}$$

The case $p = \text{Disj}(r, q)$ is similar.

If $p = \text{Neg}(\text{Conj}(r, q))$ then

$$\begin{aligned} \text{nnf}(\text{Neg}(\text{Conj}(r, q))) &= \text{nnf}(\text{Disj}(\text{Neg } r, \text{Neg } q)) && [\text{nnf}] \\ &= \text{Disj}(\text{nnf}(\text{Neg } r), \text{nnf}(\text{Neg } q)) && [\text{nnf}] \\ &= \text{Disj}(\text{nnfpos}(\text{Neg } r), \text{nnfpos}(\text{Neg } q)) && [\text{ind hyp}] \\ &= \text{nnfneg}(\text{Conj}(r, q)) && [\text{nnfneg}] \\ &= \text{nnfpos}(\text{Neg}(\text{Conj}(r, q))). && [\text{nnfpos}] \end{aligned}$$

We have induction hypotheses for $\text{Neg } r$ and $\text{Neg } q$ because they are smaller, as measured by *nodes*, than $\text{Neg}(\text{Conj}(r, q))$.

The case $p = \text{Neg}(\text{Disj}(r, q))$ is similar.

If $p = \text{Neg}(\text{Neg } r)$ then

$$\begin{aligned} \text{nnf}(\text{Neg}(\text{Neg } r)) &= \text{nnf } r && [\text{nnf}] \\ &= \text{nnfpos } r && [\text{ind hyp}] \\ &= \text{nnfneg}(\text{Neg } r) && [\text{nnfneg}] \\ &= \text{nnfpos}(\text{Neg}(\text{Neg } r)). && [\text{nnfpos}] \end{aligned}$$

An induction hypothesis applies since r contains fewer nodes than $\text{Neg}(\text{Neg } r)$. \square

The conjunctive normal form. We now consider a different question: whether computing the conjunctive normal form preserves the meaning of a proposition. A **truth valuation** for propositions is a predicate that respects the connectives:

$$\begin{aligned} \text{Tr}(\text{Neg } p) &\leftrightarrow \neg \text{Tr}(p) \\ \text{Tr}(\text{Conj}(p, q)) &\leftrightarrow \text{Tr}(p) \wedge \text{Tr}(q) \\ \text{Tr}(\text{Disj}(p, q)) &\leftrightarrow \text{Tr}(p) \vee \text{Tr}(q) \end{aligned}$$

The predicate is completely determined by its valuation of atoms, $\text{Tr}(\text{Atom } a)$. To show that the normal forms preserve truth for all valuations, we make no assumptions about which atoms are true.

Most of the work of computing CNF is performed by *distrib*:

```

fun distrib (p, Conj (q, r)) = Conj (distrib (p, q), distrib (p, r))
  | distrib (Conj (q, r), p) = Conj (distrib (q, p), distrib (r, p))
  | distrib (p, q)           = Disj (p, q)    (*no conjunctions*);

```

This function is unusual in its case analysis and its recursive calls.

The first two cases overlap if both arguments in $distrib(p, q)$ are conjunctions. Because ML tries the first case before the second, the second case cannot simply be taken as an equation. There seems to be no way of making the cases separate except to write nearly every combination of one *Atom*, *Neg*, *Disj* or *Conj* with another: at least 13 cases seem necessary. To avoid this, take the second case of $distrib$ as a conditional equation; if p does not have the form $Conj(p_1, p_2)$ then

$$distrib(Conj(q, r), p) = Conj(distrib(q, p), distrib(r, p)).$$

The computation of $distrib(p, q)$ may make recursive calls affecting either p or q . It terminates because every call reduces the value of $nodes(p) + nodes(q)$. We shall use this measure for induction.

The task of $distrib(p, q)$ is to compute a proposition equivalent to $Disj(p, q)$, but in conjunctive normal form. Its correctness can be stated as follows.

Theorem 18 For all propositions p, q and truth valuations Tr ,

$$Tr(distrib(p, q)) \leftrightarrow Tr(p) \vee Tr(q).$$

Proof We prove this by induction on $nodes(p) + nodes(q)$. The induction hypothesis is

$$Tr(distrib(p', q')) \leftrightarrow Tr(p') \vee Tr(q')$$

for all p' and q' such that $nodes(p') + nodes(q') < nodes(p) + nodes(q)$. The proof considers the same cases as in the definition of $distrib$.

If $q = Conj(q', r)$ then

$$\begin{aligned}
& Tr(distrib(p, Conj(q', r))) \\
& \leftrightarrow Tr(Conj(distrib(p, q'), distrib(p, r))) && [distrib] \\
& \leftrightarrow Tr(distrib(p, q') \wedge distrib(p, r)) && [Tr] \\
& \leftrightarrow (Tr(p) \vee Tr(q')) \wedge (Tr(p) \vee Tr(r)) && [ind\ hyp] \\
& \leftrightarrow Tr(p) \vee (Tr(q') \wedge Tr(r)) && [distributive\ law] \\
& \leftrightarrow Tr(p) \vee Tr(Conj(q', r)). && [Tr]
\end{aligned}$$

The induction hypothesis has been applied twice using these facts:

$$\begin{aligned} \text{nodes}(p) + \text{nodes}(q') &< \text{nodes}(p) + \text{nodes}(\text{Conj}(q', r)) \\ \text{nodes}(p) + \text{nodes}(r) &< \text{nodes}(p) + \text{nodes}(\text{Conj}(q', r)) \end{aligned}$$

We may now assume that q is not a *Conj*. If $p = \text{Conj}(p', r)$ then the conclusion follows as in the previous case. If neither p nor q is a *Conj* then

$$\begin{aligned} \text{Tr}(\text{distrib}(p, q)) &\leftrightarrow \text{Tr}(\text{Disj}(p, q)) && [\text{distrib}] \\ &\leftrightarrow \text{Tr}(p) \vee \text{Tr}(q). && [\text{Tr}] \end{aligned}$$

The conclusion holds in every case. \square

The proof exploits the distributive law of \vee over \wedge , as might be expected. The overlapping cases in *distrib* do not complicate the proof at all. On the contrary, they permit a concise definition of this function and a simple case analysis.

Exercise 6.23 State and justify a rule for structural induction on values of type *prop*. To demonstrate it, prove the following formula by structural induction on p :

$$\text{nnf } p = \text{nnfpos } p \wedge \text{nnf}(\text{Neg } p) = \text{nnfneg } p$$

Exercise 6.24 Define a predicate *Isnnf* on propositions such that *Isnnf*(p) holds exactly when p is in negation normal form. Prove *Isnnf*($\text{nnf } p$) for every proposition p .

Exercise 6.25 Let *Tr* be an arbitrary truth valuation for propositions. Prove $\text{Tr}(\text{nnf } p) \leftrightarrow \text{Tr}(p)$ for every proposition p .

6.8 Well-founded induction and recursion

Our treatment of induction is rigorous enough for the informal proofs we have been performing, but is not formal enough to be automated. Many induction rules can be formally derived from mathematical induction alone. A more uniform approach is to adopt the rule of **well-founded induction**, which includes most other induction rules as instances.

Well-founded relations. The relation $<$ is **well-founded** if there exist no infinite decreasing chains

$$\cdots < x_n < \cdots < x_2 < x_1.$$

For instance, ‘less than’ ($<$) on the natural numbers is well-founded. ‘Less than’ on the integers is not well-founded: there exists the decreasing chain

$$\cdots < -n < \cdots < -2 < -1.$$

‘Less than’ on the rational numbers is not well-founded either; consider

$$\cdots < \frac{1}{n} < \cdots < \frac{1}{2} < \frac{1}{1}.$$

Observe that we have to state the domain of the relation — the set of values it is defined over — and not simply say that $<$ is well-founded.

Another well-founded relation is the **lexicographic ordering** of pairs of natural numbers, defined by

$$(i', j') <_{\text{lex}} (i, j) \quad \text{if and only if} \quad i' < i \vee (i' = i \wedge j' < j).$$

To see that $<_{\text{lex}}$ is well-founded, suppose there is an infinite decreasing chain

$$\cdots <_{\text{lex}} (i_n, j_n) <_{\text{lex}} \cdots <_{\text{lex}} (i_2, j_2) <_{\text{lex}} (i_1, j_1).$$

If $(i', j') <_{\text{lex}} (i, j)$ then $i' \leq i$. Since $<$ is well-founded on the natural numbers, the decreasing chain

$$\cdots \leq i_n \leq \cdots \leq i_2 \leq i_1$$

reaches some constant value i after say M steps: thus $i_n = i$ for all $n \geq M$. Now consider the strictly decreasing chain

$$\cdots < j_{M+n} < \cdots < j_{M+1} < j_M.$$

This must eventually terminate at some constant value j after say N steps: thus $(i_n, j_n) = (i, j)$ for all $n \geq M + N$. At this point the chain of pairs becomes constant, contradicting our assumption that it was decreasing under $<_{\text{lex}}$.

Similar reasoning shows that lexicographic orderings for triples, quadruples and so forth, are well-founded. The lexicographic ordering is not well-founded for lists of natural numbers; it admits an infinite decreasing chain:

$$\cdots < [1, 1, \dots, 1, 2] < \cdots < [1, 2] < [2]$$

Another sort of well-founded relation is given by a **measure function**. If f is a function into the natural numbers, then there is a well-founded relation $<_f$ defined by

$$x <_f y \quad \text{if and only if} \quad f(x) < f(y).$$

Clearly, if there were an infinite decreasing chain

$$\cdots <_f x_n <_f \cdots <_f x_2 <_f x_1$$

then there would be an infinite decreasing chain

$$\dots < f(x_n) < \dots < f(x_2) < f(x_1)$$

in the natural numbers, which is impossible. Here f typically ‘measures’ the size of something. The well-founded relations $<_{nlength}$ and $<_{size}$ compare lists and trees by size. Our proof about *distrib* used the measure $nodes(p) + nodes(q)$ on pairs (p, q) of propositions.

The demonstration that $<_f$ is well-founded applies just as well if $<$ is replaced by any other well-founded relation. For instance, f could return pairs of natural numbers to be compared by $<_{lex}$. Similarly, the construction of $<_{lex}$ may be applied to any existing well-founded relations. There are several ways of constructing well-founded relations from others. Frequently we can show that a relation is well-founded by construction, without having to argue about decreasing chains.

Well-founded induction. Let $<$ be a well-founded relation over some type τ , and $\phi(x)$ a property to be proved for all x of type τ . To prove it by well-founded induction, it suffices to prove, for all y , the following induction step:

$$\text{if } \phi(y') \text{ for all } y' < y \text{ then } \phi(y)$$

The rule may be portrayed as follows:

$$\frac{[\forall y' < y . \phi(y')]}{\phi(x)} \quad \text{proviso: } y \text{ must not occur in other assumptions of the premise.}$$

The rule is sound by contradiction: if $\phi(x)$ is false for any x then we obtain an infinite decreasing chain in $<$. By the induction step we know that $\forall y' < x . \phi(y')$ implies $\phi(x)$. If $\neg\phi(x)$ then $\neg\phi(y_1)$ for some $y_1 < x$. Repeating this argument for y_1 , we get $\neg\phi(y_2)$ for some $y_2 < y_1$. We then get $y_3 < y_2$, and so forth.²

Complete induction is an instance of this rule, where $<$ is the well-founded relation $<$ (on the natural numbers). Our other induction rules are instances of well-founded induction for suitable choices of $<$.

² Infinite decreasing chains are intuitively appealing, but other definitions of well-foundedness permit simpler proofs. For instance, $<$ is well-founded just if every non-empty set contains a $<$ -minimal element. There also exist definitions suited for constructive logic.

The predecessor relation on the natural numbers, where $m <_N n$ just if $m + 1 = n$, is obviously well-founded. Now consider proving $\phi(y)$ under the induction hypothesis $\forall y' <_N y . \phi(y')$. There are two cases:

- If $y = 0$ then, since $y' <_N 0$ never holds, we must prove $\phi(0)$ outright.
- If $y = k + 1$ then $y' <_N k + 1$ holds just if $y' = k$, so we may assume $\phi(k)$ when proving $\phi(k + 1)$.

Therefore, well-founded induction over $<_N$ is precisely mathematical induction.

Structural induction is obtained similarly. Let $<_L$ be the relation on lists such that $xs <_L ys$ just if $x :: xs = ys$ for some x . Informally, $xs <_L ys$ means that xs is the tail of ys . Induction on $<_L$, which is clearly well-founded, yields structural induction on lists. Let $<_T$ be the relation on trees such that $t' <_T t$ just if $Br(x, t', t'') = t$ or $Br(x, t'', t') = t$ for some x and t'' . Well-founded induction on this ‘subtree of’ relation yields structural induction on trees.

A well-founded relation given by a measure function yields induction on the size of an object. In reasoning about *distrib*, induction on the size of the pair (p, q) saves us from performing nested structural inductions on q and then p .

Well-founded induction can also simulate induction on a quantified formula, as when we proved

$$\forall p . facti(n, p) = n! \times p$$

by mathematical induction. It suffices to prove $facti(n, p) = n! \times p$ by well-founded induction on the pair (n, p) under the relation $<_{fst}$, where

$$(n', p') <_{fst} (n, p) \quad \text{if and only if} \quad n' + 1 = n.$$

Although many induction principles can be derived from mathematical induction alone, the derivations typically involve quantifiers. Well-founded induction makes significant proofs possible within quantifier-free logic.

Well-founded recursion. Let $<$ be a well-founded relation over some type τ . If f is a function with formal parameter x that makes recursive calls $f(y)$ only if $y < x$, then $f(x)$ terminates for all x . In this case, f is defined by **well-founded recursion** on $<$.

Informally, $f(x)$ terminates because $<$ has no infinite decreasing chains: there can be no infinite recursion. A formal justification of well-founded recursion is complex; besides termination, it must show that $f(x)$ is uniquely defined.

For most of our recursive functions, the well-founded relation is obvious. If $n > 0$ then $fact(n)$ recursively calls $fact(n - 1)$, so $fact$ is defined by

well-founded recursion on the predecessor relation, $<_N$. When $facti(n, p)$ recursively calls $facti(n - 1, n \times p)$ it changes the second argument; its well-founded relation is $<_{fst}$. The list functions $nlength$, $@$ and $nrev$ are recursive over the ‘tail of’ relation, $<_L$.

Proving that a function terminates suggests a useful form of induction for it — recall our proofs involving nnf and $distrib$. If a function is defined by well-founded recursion on $<$, then its properties can often be proved by well-founded induction on $<$.



Well-founded relations in use. Well-founded relations are central to the Boyer and Moore (1988) theorem prover, also called NQTHM. It accepts functions defined by well-founded recursion and employs elaborate heuristics to choose the right relation for well-founded induction. Its logic is quantifier-free, but as we have seen, this is not a fatal restriction. NQTHM is one of the most important theorem provers in existence. Demanding proofs in numerous areas of mathematics and computer science have been performed using it. The theorem prover Isabelle formally develops a theory of well-founded relations (Paulson, 1995, Section 3).

6.9 Recursive program schemes

Well-founded relations permit reasoning about program schemes. Suppose that p and g are functions and that \oplus is an infix operator, and consider the ML declarations

```
fun f1(x) = if p(x) then e else f1(g x) ⊕ x;
fun f2(x, y) = if p(x) then y else f2(g x, x ⊕ y);
```

Suppose that we are also given a well-founded relation $<$ such that $g(x) < x$ for all x such that $p(x) = \text{false}$. We then know that $f1$ and $f2$ terminate, and can prove theorems about them.

Theorem 19 *Suppose \oplus is an infix operator that is associative and has identity e ; that is, for all x, y and z ,*

$$\begin{aligned} x \oplus (y \oplus z) &= (x \oplus y) \oplus z \\ e \oplus x &= x = x \oplus e. \end{aligned}$$

Then for all x and a we have $f2(x, e) = f1(x)$.

Proof It suffices to prove the following formula, then put $y = e$:

$$\forall y. f2(x, y) = f1(x) \oplus y.$$

This holds by well-founded induction over $<$. There are two cases.

If $p(x) = \text{true}$ then

$$\begin{aligned} f2(x, y) &= y && [f2] \\ &= e \oplus y && [\text{identity}] \\ &= f1(x) \oplus y. && [f1] \end{aligned}$$

If $p(x) = \text{false}$ then

$$\begin{aligned} f2(x, y) &= f2(g\ x, x \oplus y) && [f2] \\ &= f1(g\ x) \oplus x \oplus y && [\text{ind hyp}] \\ &= f1(x) \oplus y. && [f1] \end{aligned}$$

The induction hypothesis applies because $g(x) < x$. We have implicitly used the associativity of \oplus . \square

Thus we can transform a recursive function ($f1$) into an iterative function with an accumulator ($f2$). The theorem applies to the computation of factorials. Put

$$\begin{aligned} e &= 1 \\ \oplus &= \times \\ g(x) &= x - 1 \\ p(x) &= (x = 0) \\ < &= <_N \end{aligned}$$

Then $f1$ is the factorial function while $f2$ is the function *facti*. The theorem generalizes Theorem 4.

Our approach to program schemes is simpler than resorting to domain theory, but is less general. In domain theory it is simple to prove that any ML function of the form

$$\text{fun } h\ x = \text{if } p\ x \text{ then } x \text{ else } h(h(g\ x));$$

satisfies $h(h\ x) = h\ x$ for all x — regardless of whether the function terminates. Our approach cannot easily handle this. What well-founded relation should we use to demonstrate the termination of the nested recursive call in h ?

Exercise 6.26 Recall the function *fst*, such that $\text{fst}(x, y) = x$ for all x and y . Give an example of a well-founded relation that uses *fst* as a measure function.

Exercise 6.27 Consider the function *ack*:

$$\begin{aligned} \text{fun } \text{ack}(0, n) &= n+1 \\ | \text{ack}(m, 0) &= \text{ack}(m-1, 1) \\ | \text{ack}(m, n) &= \text{ack}(m-1, \text{ack}(m, n-1)); \end{aligned}$$

Use a well-founded relation to show that $ack(m, n)$ is defined for all natural numbers m and n . Prove $ack(m, n) > m + n$ by well-founded induction.

Exercise 6.28 Give an example of a well-founded relation that is not transitive. Show that if $<$ is well-founded then so is $<^+$, its transitive closure.

Exercise 6.29 Consider the function *half*:

```
fun half 0 = 0
  | half n = half (n-2);
```

Show that this function is defined by well-founded recursion. Be sure to specify the domain of the well-founded relation.

Exercise 6.30 Show that well-founded induction on the ‘tail of’ relation $<_L$ is equivalent to structural induction for lists.

Specification and verification

Sorting is a good example for program verification: it is simple but not trivial. Considerable effort is required just to specify what sorting is. Most of our previous correctness proofs concerned the equivalence of two functions, and took little more than a page. Proving the correctness of the function *tmergesort* takes most of this section, even though many details are omitted.

First, consider a simpler specification task: the Greatest Common Divisor. If m and n are natural numbers then k is their GCD just if k divides both m and n exactly, and is the greatest number to do so. Given this specification, it is not hard to verify an ML function that computes the GCD by Euclid’s Algorithm:

```
fun gcd(m, n) =
  if m=0 then n else gcd(n mod m, m);
```

The simplest approach is to observe that the specification defines a mathematical function:

$$GCD(m, n) = \max\{k \mid k \text{ divides both } m \text{ and } n\}$$

The value of $GCD(m, n)$ is uniquely defined unless $m = n = 0$, when the maximum does not exist; we need **not** know whether $GCD(m, n)$ is computable. Using simple number theory it is possible to prove these facts:

$$\begin{aligned} GCD(0, n) &= n && \text{for } n > 0 \\ GCD(m, n) &= GCD(n \bmod m, m) && \text{for } m > 0 \end{aligned}$$

A trivial induction proves that $\text{gcd}(m, n) = \text{GCD}(m, n)$ for all natural numbers m and n not both zero. We thereby learn that $\text{GCD}(m, n)$ is computable.

A sorting function is not verified like this. It is not practical to define a mathematical function *sorting* and to prove $\text{tmergesort}(xs) = \text{sorting}(xs)$. Sorting involves two different correctness properties, which can be considered separately:

- 1 The output must be an ordered list.
- 2 The output must be some rearrangement of the elements of the input.

Too often in program verification, some of the correctness properties are ignored. This is dangerous. A function can satisfy property 1 by returning the empty list, or property 2 by returning its input unchanged. Either property alone is useless.

The specification does not have to specify the output uniquely. We might specify that a compiler generates correct code, but should not specify the precise code to generate. This would be too complicated and would forbid code optimizations. We might specify that a database system answers queries correctly, but should not specify the precise storage layout.

The next sections will prove that *tmergesort* is correct, in the sense that it returns an ordered rearrangement of its input. Let us recall some functions from Chapter 3. Proving that they terminate is left as an exercise.

The list utilities *take* and *drop*:

```
fun take ([], i)      = []
  | take (x::xs, i) = if i>0 then x::take(xs, i-1) else [];

fun drop ([], _)     = []
  | drop (x::xs, i) = if i>0 then drop(xs, i-1) else x::xs;
```

The merging function:

```
fun merge ([], ys)      = ys : real list
  | merge (xs, [])      = xs
  | merge (x::xs, y::ys) = if x<=y then x::merge(xs, y::ys)
                           else y::merge(x::xs, ys);
```

The top-down merge sort:

```

fun tmergesort [] = []
  | tmergesort [x] = [x]
  | tmergesort xs =
    let val k = length xs div 2
    in merge (tmergesort (List.take(xs, k)),
              tmergesort (List.drop(xs, k)))
    end;

```

6.10 An ordering predicate

The predicate *ordered* expresses that the elements of a list are in increasing order under \leq . Its properties include the following:

$$\begin{aligned} & \text{ordered}([]) \\ & \text{ordered}([x]) \\ & \text{ordered}(x :: y :: ys) \leftrightarrow x \leq y \wedge \text{ordered}(y :: ys). \end{aligned}$$

Note that $\text{ordered}(x :: xs)$ implies $\text{ordered}(xs)$. We now prove that merging two ordered lists yields another ordered list.

Theorem 20 For all lists xs and ys ,

$$\text{ordered}(xs) \wedge \text{ordered}(ys) \rightarrow \text{ordered}(\text{merge}(xs, ys)).$$

Proof By induction on the value of $nlength\ xs + nlength\ ys$.

If $xs = []$ or $ys = []$ then the conclusion follows by the definition of *merge*. So assume $xs = x :: xs'$ and $ys = y :: ys'$ for some xs' and ys' . We may assume

$$\text{ordered}(x :: xs') \quad \text{and} \quad \text{ordered}(y :: ys')$$

and must show

$$\text{ordered}(\text{merge}(x :: xs', y :: ys')).$$

Consider the case where $x \leq y$. (The case where $x > y$ is similar and is left as an exercise.) By the definition of *merge*, it remains to show

$$\text{ordered}(x :: \text{merge}(xs', y :: ys')).$$

As we know $\text{ordered}(xs')$, we may apply the induction hypothesis, obtaining

$$\text{ordered}(\text{merge}(xs', y :: ys')).$$

Finally we have to show $x \leq u$, where u is the head of $\text{merge}(xs', y :: ys')$. Determining the head requires further case analysis.

If $xs' = []$ then $\text{merge}(xs', y :: ys') = y :: ys'$. Its head is y and we have already assumed $x \leq y$.

If $xs' = v :: vs$ then there are two subcases:

- If $v \leq y$ then $\text{merge}(xs', y :: ys') = v :: \text{merge}(vs, y :: ys')$. The head is v and $x \leq v$ follows from $\text{ordered}(xs)$ since $xs = x :: v :: vs$.
- If $v > y$ then $\text{merge}(xs', y :: ys') = y :: \text{merge}(xs', ys')$. The head is y and we have assumed $x \leq y$. \square

The proof is surprisingly tedious. Perhaps merge is less straightforward than it looks. Anyway, we are now ready to show that tmergesort returns an ordered list.

Theorem 21 For every list xs , $\text{ordered}(\text{tmergesort } xs)$.

Proof By induction on the length of xs . If $xs = []$ or $xs = [x]$ then the conclusion is obvious, so assume $\text{nlength } xs \geq 2$.

Let $k = (\text{nlength } xs) \text{ div } 2$. Then $1 \leq k < \text{nlength } xs$. It is easy to show these inequalities:

$$\begin{aligned} \text{nlength}(\text{take}(xs, k)) &= k < \text{nlength } xs \\ \text{nlength}(\text{drop}(xs, k)) &= \text{nlength } xs - k < \text{nlength } xs \end{aligned}$$

By the induction hypotheses, we obtain corresponding facts:

$$\begin{aligned} &\text{ordered}(\text{tmergesort}(\text{take}(xs, k))) \\ &\text{ordered}(\text{tmergesort}(\text{drop}(xs, k))) \end{aligned}$$

Since both arguments of merge are ordered, the conclusion follows by the previous theorem. \square

Exercise 6.31 Fill in the details of the proofs in this section.

Exercise 6.32 Write another predicate to define the notion of ordered list, and prove that it is equivalent to ordered .

6.11 Expressing rearrangement through multisets

If the output of the sort is a rearrangement of the input, then there is a function, called a **permutation**, that maps element positions in the input to the corresponding positions in the output. To show the correctness of sorting, we could provide a method of exhibiting the permutation. However, we do not need so much information; it would complicate the proof. This specification is too concrete.

We could show that the input and output of the sort contained the same set of elements, not considering where each element was moved. Unfortunately,

this approach accepts $[1,1,1,1,2]$ as a valid sorting of $[2,1,2]$. Sets do not take account of repeated elements. This specification is too abstract.

Multisets are a good way to specify sorting. A multiset is a collection of elements that takes account of their number but not of their order. The multisets $\langle 1, 1, 2 \rangle$ and $\langle 1, 2, 1 \rangle$ are equal; they differ from $\langle 1, 2 \rangle$. Multisets are often called **bags**, for reasons that should be obvious. Here are some ways of forming multisets:

- \emptyset , the empty bag, contains no elements.
- $\langle u \rangle$, the singleton bag, contains one occurrence of u .
- $b_1 \uplus b_2$, the bag sum of b_1 and b_2 , contains all elements in the bags b_1 and b_2 (accumulating repetitions of elements).

Rather than assume bags as primitive, let us represent them as functions into the natural numbers. If b is a bag then $b(x)$ is the number of occurrences of x in b . Thus, for all x ,

$$\begin{aligned}\emptyset(x) &= 0 \\ \langle u \rangle(x) &= \begin{cases} 0 & \text{if } u \neq x \\ 1 & \text{if } u = x \end{cases} \\ (b_1 \uplus b_2)(x) &= b_1(x) + b_2(x).\end{aligned}$$

These laws are easily checked:

$$\begin{aligned}b_1 \uplus b_2 &= b_2 \uplus b_1 \\ (b_1 \uplus b_2) \uplus b_3 &= b_1 \uplus (b_2 \uplus b_3) \\ \emptyset \uplus b &= b.\end{aligned}$$

Let us define a function to convert lists into bags:

$$\begin{aligned}bag[] &= \emptyset \\ bag(x :: xs) &= \langle x \rangle \uplus bag\ xs.\end{aligned}$$

The ‘rearrangement’ correctness property can finally be specified:

$$bag(tmergesort\ xs) = bag\ xs$$

A preliminary proof. To illustrate reasoning about multisets, let us work through a proof. It is a routine induction.³

³ It would be less routine if we adopted the standard library definitions of *take* and *drop*, where *take*(xs, k) raises an exception unless $0 \leq k < length\ xs$.

Theorem 22 For every list xs and integer k ,

$$bag(take(xs, k)) \uplus bag(drop(xs, k)) = bag\ xs.$$

Proof By structural induction on the list xs . In the base case,

$$\begin{aligned} & bag(take([], k)) \uplus bag(drop([], k)) \\ &= bag[] \uplus bag[] && [take, drop] \\ &= \emptyset \uplus \emptyset && [bag] \\ &= \emptyset && [\uplus] \\ &= bag[]. && [bag] \end{aligned}$$

For the induction step, we must prove

$$bag(take(x :: xs, k)) \uplus bag(drop(x :: xs, k)) = bag(x :: xs).$$

If $k > 0$ then

$$\begin{aligned} & bag(take(x :: xs, k)) \uplus bag(drop(x :: xs, k)) \\ &= bag(x :: take(xs, k - 1)) \uplus \\ & \quad bag(drop(xs, k - 1)) && [take, drop] \\ &= \langle x \rangle \uplus bag(take(xs, k - 1)) \uplus \\ & \quad bag(drop(xs, k - 1)) && [bag] \\ &= \langle x \rangle \uplus bag\ xs && [ind hyp] \\ &= bag(x :: xs). && [bag] \end{aligned}$$

If $k \leq 0$ then

$$\begin{aligned} & bag(take(x :: xs, k)) \uplus bag(drop(x :: xs, k)) \\ &= bag[] \uplus bag(x :: xs) && [take, drop] \\ &= \emptyset \uplus bag(x :: xs) && [bag] \\ &= bag(x :: xs). && [\uplus] \end{aligned}$$

Therefore, the conclusion holds for every integer k . \square

The next step is to show that *merge* combines the elements of its arguments when forming its result.

Theorem 23 For all lists xs and ys , $bag(merge(xs, ys)) = bag\ xs \uplus bag\ ys$.

We should have to constrain k in the statement of the theorem and modify the proof accordingly.

Proof By induction on the value of $nlength\ xs + nlength\ ys$.

If $xs = []$ or $ys = []$ then the conclusion is immediate, so assume $xs = x :: xs'$ and $ys = y :: ys'$ for some xs' and ys' . We must prove

$$bag(merge(x :: xs', y :: ys')) = bag(x :: xs') \uplus bag(y :: ys').$$

If $x \leq y$ then

$$\begin{aligned} & bag(merge(x :: xs', y :: ys')) \\ &= bag(x :: merge(xs', y :: ys')) && [merge] \\ &= \langle x \rangle \uplus bag(merge(xs', y :: ys')) && [bag] \\ &= \langle x \rangle \uplus bag\ xs' \uplus bag(y :: ys') && [ind\ hyp] \\ &= bag(x :: xs') \uplus bag(y :: ys'). && [bag] \end{aligned}$$

The case $x > y$ is similar. \square

Finally, we prove that merge sort preserves the bag of elements given to it.

Theorem 24 For every list xs , $bag(tmergesort\ xs) = bag\ xs$.

Proof By induction on the length of xs . The only hard case is if $nlength\ xs \geq 2$. As in Theorem 21, the induction hypotheses apply to $take(xs, k)$ and $drop(xs, k)$:

$$\begin{aligned} bag(tmergesort(take(xs, k))) &= bag(take(xs, k)) \\ bag(tmergesort(drop(xs, k))) &= bag(drop(xs, k)) \end{aligned}$$

Therefore

$$\begin{aligned} & bag(tmergesort\ xs) \\ &= bag(merge(tmergesort(take(xs, k)), \\ &\quad tmergesort(drop(xs, k)))) && [tmergesort] \\ &= bag(tmergesort(take(xs, k)) \uplus \\ &\quad bag(tmergesort(drop(xs, k)))) && [Theorem\ 23] \\ &= bag(take(xs, k)) \uplus bag(drop(xs, k)) && [ind\ hyp] \\ &= bag\ xs. && [Theorem\ 22] \end{aligned}$$

This concludes the verification of $tmergesort$. \square

Exercise 6.33 Verify that \uplus is commutative and associative. (Hint: recall the extensional equality of functions.)

Exercise 6.34 Prove that insertion sort preserves the bag of elements it is given. In particular, prove these facts:

$$\begin{aligned} \text{bag}(\text{ins}(x, xs)) &= \langle x \rangle \uplus \text{bag } xs \\ \text{bag}(\text{insort } xs) &= \text{bag } xs \end{aligned}$$

Exercise 6.35 Modify merge sort to suppress repetitions: each input element should appear exactly once in the output. Formalize this property and state the theorems required to verify it.

6.12 The significance of verification

We could now announce that *tmergesort* has been verified — but would this mean anything? What, exactly, have we established about *tmergesort*? Formal verification has three fundamental limitations:

- 1 The model of computation may be too imprecise. Typically the hardware is assumed to be infallible. A model can be designed to cope with specific errors, like arithmetic overflow, rounding errors, or running out of store. However, a computer can fail in unanticipated ways. What if somebody takes an axe to it?
- 2 The specification may be incomplete or wrong. Design requirements are difficult to formalize, especially if they pertain to the real world. Satisfying a faulty specification will not satisfy the customer. Software engineers understand the difference between *verification* (did we build the product right?) and *validation* (did we build the right product?).
- 3 Proofs may contain errors. Automated theorem proving can reduce but not eliminate the likelihood of error. All human efforts may have flaws, even our principles of mathematics. This is not merely a philosophical problem. Many errors have been discovered in theorem provers, in proof rules and in published proofs.

Apart from these fundamental limitations, there is a practical one: formal proof is tedious. Look back over the proofs in this chapter; usually they take great pains to prove something elementary. Now consider verifying a compiler. The specification will be gigantic, comprising the syntax and semantics of a programming language along with the complete instruction set of the target machine. The compiler will be a large program. The proof should be divided into parts, separately verifying the parser, the type checker, the intermediate code generator and so forth. There may only be time to verify the most interesting

part of the program: say, the code generator. So the ‘verified’ compiler could fail due to faulty parsing.

Let us not be too negative. Writing a formal specification reveals ambiguities and inconsistencies in the design requirements. Since design errors are far more serious than coding errors, writing a specification is valuable even if the code will not be verified. Many companies go to great expense to produce specifications that are rigorous, if not strictly formal.

The painstaking work of verification yields rewards. Most programs are incorrect, and an attempted proof often pinpoints the error. To see this, insert an error into any program verified in this chapter, and work through the proof again. The proof should fail, and the location of this failure should indicate the precise conditions under which the modified program fails.

A correctness proof is a detailed explanation of how the program or system works. If the proof is simple, we can go through it line by line, reading it as a series of snapshots of the execution. The inductive step traces what happens at a recursive function call. A large proof may consist of hundreds of theorems, examining every component or subsystem.

Specification and verification yield a fuller knowledge of the program and its task. This leads to increased confidence in the system. Formal proof does not eliminate the need for systematic testing, especially for a safety-critical system. Testing is the only way to investigate whether the computational model and the formal specification accurately reflect the real world. However, while testing can detect errors, it cannot guarantee success; nor does it provide insights into how a program works.



Further reading. Bevier *et al.* (1989) have verified a tiny computer system consisting of several levels, both software and hardware. Avra Cohn (1989a) has verified some correctness properties of the Viper microprocessor. Taking her proofs as an example, Cohn (1989b) discusses the fundamental limitations of verification.

Fitzgerald *et al.* (1995) report a study in which two teams independently develop a trusted gateway. The control team uses conventional methods while the experimental team augments these methods by writing a formal specification. An unusually large study involves the AAMP5 pipelined microprocessor. This is a commercial product designed for use in avionics. It has been specified on two levels and some of its microcode proved correct (Srivivas and Miller, 1995). Both studies suggest that writing formal specifications — whether or not they are followed up by formal proofs — uncovers errors.

A major study by Susan Gerhart *et al.* (1994) investigated 12 cases involving the use of formal methods. And in a famous philosophical monograph, Lakatos (1976) argues that we can learn from partial, even faulty, proofs.

Summary of main points

- Many functional programs can be given a meaning within elementary mathematics. Higher-order functions can be handled, but not lazy evaluation or infinite data structures.
- The proof that a function terminates has the same general form as most other proofs about the function.
- Mathematical induction applies to recursive functions over the natural numbers.
- Structural induction applies to recursive functions over lists and trees.
- Well-founded induction and recursion handle a wide class of terminating computations.
- Program proofs require precise and simple program specifications.
- Proofs can be fallible, but usually convey valuable information.