Chapter 7

Proving programs correct

As programmers know through bitter personal experience, it can be very difficult to write a program that is *correct*, i.e. performs its intended function. Most large programs have bugs. The consequences of these bugs can vary widely. Some bugs are harmless, some merely irritating. Some are deadly. For example the programs inside heart pacemakers, aircraft autopilots, car engine management systems and antilock braking systems, radiation therapy machines and nuclear reactor controllers are *safety critical*. An error in one of them can lead directly to loss of life, possibly on a large scale. As computers become ever more pervasive in society, the number of ways in which program bugs can kill people increases.[1]

How can we make sure a program is correct? One very useful method is simply to test it in a variety of situations. One tries the program out on inputs designed to exercise many different parts of it and reveal possible bugs. But usually there are too many possible situations to try them all exhaustively, so there may still be bugs lying undetected. As repeatedly emphasized by various writers, notably Dijkstra, program testing can be very useful for demonstrating the presence of bugs, but it is only in a few unusual cases where it can demonstrate their absence.

Instead of testing a program, one can try to *verify* it mathematically. A program in a sufficiently precisely defined programming language has a definite mathematical meaning. Similarly, the requirements on a program can be expressed using mathematics and logic as a precise *specification*. One can then try to perform a mathematical *proof* that the program meets its specification.

One can contrast testing and verification of programs by considering the following simple mathematical assertion:

$$\Sigma_{n=0}^{N} n = \frac{N(N+1)}{2}$$

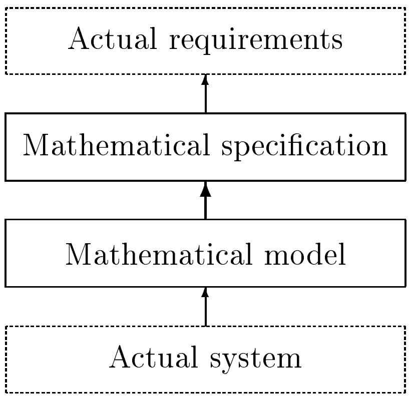
It is claimed that this holds for any N. We can certainly test it for any number of particular values of N. This may lead to our intuitive confidence in the formula, such that we can hardly believe it could ever fail. However in

[1]Similar remarks apply to hardware, but we will focus on software.

general a preponderance of numerical evidence can be deceptive; there are well known cases in number theory where things turned out to be false against the weight of many particular cases.[2] A more reliable procedure is to *prove* the above formula mathematically. This can easily be done using induction on N. In the same way, we hope to replace testing a program on a range of inputs by some completely general proof that it always works correctly. It is important, however, to appreciate two limitations of program proofs.

- One has no guarantee that the execution of the program on the computer corresponds exactly to the abstract mathematical model. One must be on the lookout for discrepancies, caused perhaps by bugs in compilers and operating systems, or physical defects in the underlying technology. This is, of course, a general problem in applications of mathematics to science. For example, assuming that arithmetic operations correspond exactly to their mathematical counterparts is a natural simplifying assumption, just as one might neglect air resistance in the analysis of a simple dynamical system. But both can be invalid in certain cases.
- The program is to be verified by proof against a mathematical specification. It is possible that this specification does not capture accurately what is actually wanted in the real world. In fact, it can be remarkably difficult to arrive at a precise mathematical version of informal requirements. There is evidence that many of the serious problems in computer systems are caused not by coding errors in the usual sense, but by a mistaken impression of what is actually wanted. Just formalizing the requirements is often valuable in itself.

We can represent this situation by the following diagram:



We are trying to establish a link between the bottom and top boxes, i.e. the actual system and the actual requirements in real life. To do this, we proceed by

[2] For example, Littlewood proved in 1914 that $\pi(n) - li(n)$ changes sign infinitely often, where $\pi(n)$ is the number of primes $\leq n$ and $li(n) = \int_0^n du/ln(u)$. This came as a surprise since not a single sign change had been found despite extensive testing up to 10^{10}.

producing a mathematical version of each. It is only the central link, between the mathematical model of the system and the mathematical version of the specification, that is mathematically precise. The other links remain informal. All we can do is try to keep them small by using a realistic model of the system and a high-level mathematical specification that is reasonably readable.

These reservations notwithstanding, program proving has considerable merit. Compared with testing, it establishes correctness once and for all, possibly for a whole class of programs at once (e.g. controlled by the setting of some parameter). Moreover because it is a more principled analytical approach, the process of proving a program (or failing to do so) can lead to a deeper appreciation of the program and of the task in hand.

7.1 Functional programs as mathematical objects

In the introduction, we remarked that (pure) functional programs really correspond directly to functions in the mathematical sense. For this reason, it is often suggested that they are more amenable to formal proof than imperative programs. Even if this is true, and many would dispute it, it is worth realizing that the gap between this mathematical abstraction and the final execution in hardware is greater than for typical imperative languages. In particular, there can be substantial storage requirements hidden in the eventual realization on the computer. So one may just be getting easier proofs because they prove less about what really matters. We won't settle this argument here, but we want to show that reasoning about simple functional programs *is* often fairly straightforward.

The example at the end of the previous chapter showed that a naive association of function spaces of ML and function spaces of mathematics is not quite accurate. However, if we stay away from the higher reaches of recursive types and are not concerned with the space of *all* functions, only with some particular ones, we can safely identify the objects of ML with those of abstract mathematical realms.[3]

We intend to model functional programs as mathematical functions. Given the equations — typically recursive — that define an ML function, we want to interpret them as 'axioms' about the corresponding mathematical objects. For example, we would assume from the definition of the factorial function:

```
#let rec fact = fun 0 -> 1
                  | n -> n * fact(n - 1);;
```

that $\mathrm{fact}(0) = 1$ and that for all $n \neq 0$ we have $\mathrm{fact}(n) = n * \mathrm{fact}(n-1)$. This is right, but we need to tread carefully because of the question of termination. For

[3] We will neglect arithmetic overflow whenever we use arithmetic. There is a facility in CAML for arbitrary precision arithmetic, at the cost of, in principle, unlimited storage requirements.

negative n, the program fails to terminate, so the second equation is true only in the vacuous sense that both sides are 'undefined'. It is much easier to reason about functions when we know they are total, so sometimes one uses a separate argument to establish termination. In the examples that follow, this is done in parallel with the correctness proof, i.e. we prove together that for each argument, the function terminates and the result satisfies our specification.

As for the actual proof that the function obeys the specification, this can be a mathematical proof in the most general sense. However it often happens that when functions are defined by recursion, properties of them, dually, can be proved by induction; this includes termination. Moreover, the exact form of induction (arithmetical, structural, wellfounded) usually corresponds to the mode of recursion used in the definition. This should become clearer when we have seen a few examples.

7.2 Exponentiation

Recall our simple definition of exponentiation:

```
  #let rec exp x n =
    if n = 0 then 1
    else x * exp x (n - 1);;
```

We will prove the following property of `exp`:

Theorem 7.1 *For all $n \geq 0$ and x, $exp\ x\ n$ is defined and $exp\ x\ n = x^n$.*
Proof: *The ML function* `exp` *was defined by primitive, step-by-step, recursion. It is therefore no surprise that our proof proceeds by ordinary, step-by-step induction. We prove that it holds for $n = 0$ and then that if it holds for any $n \geq 0$ it also holds for $n + 1$.*

1. *If $n = 0$, then by definition $exp\ x\ n = 1$. By definition, for any integer x, we have $x^0 = 1$, so the desired fact is established. Note that we assume $0^0 = 1$, an example of how one must state the specification precisely — what does x^n mean in general? — and how it might surprise some people.*

2. *Suppose that for $n \geq 0$ we have $exp\ x\ n = x^n$. Since $n \geq 0$, we have $n + 1 \neq 0$ and so by definition $exp\ x\ (n + 1) = x * exp\ x\ ((n + 1) - 1)$. Therefore:*

$$\begin{aligned} exp\ x\ (n+1) &= x * exp\ x\ ((n+1) - 1) \\ &= x * exp\ x\ n \\ &= x * x^n \\ &= x^{n+1} \end{aligned}$$

Q.E.D.

7.3 Greatest common divisor

Recall our function to calculate the greatest common divisor $gcd(m, n)$ of two natural numbers m and n:

```
#let rec gcd x y =
     if y = 0 then x else gcd y (x mod y);;
```

In fact, we claim this works for any integers, not just positive ones. However, one needs to understand the precise definition of gcd in that case. Let us define the relation '$u|v$', or 'u divides v' for any two integers u and v to mean 'v is an integral multiple of u', i.e. there is some integer d with $v = du$. For example, $0|0$, $1|11$, $-2|4$ but $0 \not| 1$, $3 \not| 5$. We will say that *d is a greatest common divisor of x and y* precisely if:

- $d|x$ and $d|y$
- For any other integer d', if $d'|x$ and $d'|y$ then $d'|d$.

Note that we say $d'|d$ not $d' \leq d$. This belies the use of 'greatest', but it is in fact the standard definition in algebra books. Observe that any pair of numbers (except 0 and 0) has two gcds, since if d is a gcd of x and y, so is $-d$.

Our specification is now: *for any integers x and y, $d = gcd\ x\ y$ is a gcd of x and y.* This is another good example of how providing a precise specification, even for such a simple function, is harder than it looks. This specification also exemplifies the common property that it does not completely specify the answer, merely places certain constraints on it. If we defined:

```
#let rec ngcd x y = -(gcd x y);;
```

then the function `ngcd` would satisfy the specification just as well as `gcd`. Of course, we are free to tighten the specification if we wish, e.g. insist that if x and y are positive, so is the gcd returned.

The `gcd` function is not defined simply by primitive recursion. In fact, `gcd x y` is defined in terms of `gcd y (x mod y)` in the step case. Correspondingly, we do not use step-by-step induction, but wellfounded induction. We want a wellfounded relation that decreases over recursive calls; this will guarantee termination and act as a handle for proofs by wellfounded induction. In general, complicated relations on the arguments are required. But often it is easy to concoct a *measure* that maps the arguments to natural numbers such that the measure decreases over recursive calls. Such is the case here: the measure is $|y|$.

Theorem 7.2 *For any integers x and y, $gcd\ x\ y$ terminates in a gcd of x and y.*
Proof: *Take some arbitrary n, and suppose that for all x and y with $|y| < n$ we have that $gcd\ x\ y$ terminates with a gcd of x and y, and try to prove that the same holds for all x and y with $|y| = n$. This suffices for the main result, since any y will have* some *n with $|y| = n$. So, let us consider an x and y with $|y| = n$. There are two cases to consider, according to the case split used in the definition.*

- *Suppose $y = 0$. Then $gcd\ x\ y = x$ by definition. Now trivially $x|x$ and $x|0$, so it is a common divisor. Suppose d is another common divisor, i.e. $d|x$ and $d|0$. Then indeed we have $d|x$ immediately, so x must be a greatest common divisor.*

- *Suppose $y \neq 0$. We want to apply the inductive hypothesis to $gcd\ y\ (x\ mod\ y)$. We will write $r = x\ mod\ y$ for short. The basic property of the* `mod` *function is that, since $y \neq 0$, for some integer q we have $x = qy + r$ and $|r| < |y|$. Since $|r| < |y|$ the inductive hypothesis tells us that $d = gcd\ y\ (x\ mod\ y)$ is a gcd of y and r. We just need to show that it is a gcd of x and y. It is certainly a common divisor, since if $d|y$ and $d|r$ we have $d|x$, as $x = qy+r$. Now suppose $d'|x$ and $d'|y$. By the same equation, we find that $d'|r$. Thus d' is a common divisor of y and r, but then by the inductive hypothesis, $d'|d$ as required.*

Q.E.D.

Note that the basic property of the modulus operation that we used needs careful checking against the specification of `mod` in the CAML manual. There are well known differences between languages, and between implementations of the same language, when moduli involving negative numbers are concerned. If ever one is in doubt over the validity of a necessary assumption, it can be made explicit in the theorem: 'if ... then ...'.

7.4 Appending

We will now have an example of a function defined over lists. The function `append` is intended, not surprisingly, to append, or join together, two lists. For example, if we append $[3; 2; 5]$ and $[6; 3; 1]$ we get $[3; 2; 5; 6; 3; 1]$.

```
#let rec append l1 l2 =
   match l1 with
     [] -> l2
   | (h::t) -> h::(append t l2);;
```

This is defined by primitive recursion over the type of lists. It is defined on the empty list, and then for $h :: t$ in terms of the value for t. Consequently, when proving theorems about it, it is natural to use the corresponding principle of structural induction for lists: if a property holds for the empty list, and whenever it holds for t it holds for any $h :: t$, then it holds for any list. However this is not obligatory, and if preferred, we could proceed by mathematical induction on the length of the list. We will aim at proving that the append operation is associative.

Theorem 7.3 *For any three lists l_1, l_2 and l_3 we have:*

$$\mathit{append}\, l_1\ (\mathit{append}\, l_2\ l_3) = \mathit{append}\ (\mathit{append}\, l_1\ l_2)\ l_3$$

Proof: *By structural induction on l_1, we prove this holds for any l_2 and l_3.*

- *If $l_1 = []$ then:*

$$\begin{aligned} \mathit{append}\, l_1\ (\mathit{append}\, l_2\ l_3) &= \mathit{append}\, []\ (\mathit{append}\, l_2\ l_3) \\ &= \mathit{append}\, l_2\ l_3 \\ &= \mathit{append}\ (\mathit{append}\, []\ l_2)\ l_3 \\ &= \mathit{append}\ (\mathit{append}\, l_1\ l_2)\ l_3 \end{aligned}$$

- *Now suppose $l_1 = h :: t$. We may assume that for any l_2 and l_3 we have:*

$$\mathit{append}\, t\ (\mathit{append}\, l_2\ l_3) = \mathit{append}\ (\mathit{append}\, t\ l_2)\ l_3$$

  Therefore:

$$\begin{aligned} \mathit{append}\, l_1\ (\mathit{append}\, l_2\ l_3) &= \mathit{append}\ (h :: t)\ (\mathit{append}\, l_2\ l_3) \\ &= h :: (\mathit{append}\, t\ (\mathit{append}\, l_2\ l_3)) \\ &= h :: (\mathit{append}\ (\mathit{append}\, t\ l_2)\ l_3) \\ &= \mathit{append}\ (h :: (\mathit{append}\, t\ l_2))\ l_3) \\ &= \mathit{append}\ (\mathit{append}\ (h :: t)\ l_2)\ l_3) \\ &= \mathit{append}\ (\mathit{append}\, l_1\ l_2)\ l_3) \end{aligned}$$

Q.E.D.

7.5 Reversing

It is not difficult to define a function to reverse a list:

```
#let rec rev =
   fun [] -> []
     | (h::t) -> append (rev t) [h];;
rev : 'a list -> 'a list = <fun>
#rev [1;2;3];;
- : int list = [3; 2; 1]
```

We will prove that `rev` is an involution, i.e. that

$$\text{rev}(\text{rev } l) = l$$

However, if we try to tackle this directly by list induction, we find that we need a couple of additional lemmas. We will prove these first.

Lemma 7.4 *For any list l we have $append\, l\, [] = l$.*
Proof: *Structural induction on l.*

- *If $l = []$ we have:*

$$\begin{aligned} append\, l\, [] &= append\, []\, [] \\ &= [] \\ &= l \end{aligned}$$

- *Now suppose $l = h :: t$ and we know that $append\, t\, [] = t$. We find:*

$$\begin{aligned} append\, l\, [] &= append\, (h :: t)\, [] \\ &= h :: (append\, t\, []) \\ &= h :: t \\ &= l \end{aligned}$$

Q.E.D.

Lemma 7.5 *For any lists l_1 and l_2 we have*

$$rev(append\, l_1\, l_2) = append\, (rev\, l_2)\, (rev\, l_1)$$

Proof: *Structural induction on l_1.*

- *If $l_1 = []$ we have:*

$$\begin{aligned} rev(append\, l_1\, l_2) &= rev(append\, []\, l_2) \\ &= rev\, l_2 \\ &= append\, (rev\, l_2)[] \\ &= append\, (rev\, l_2)\, (rev\, []) \end{aligned}$$

- *Now suppose* $l_1 = h :: t$ *and we know that*

$$rev(append\, t\ l_2) = append\,(rev\ l_2)\ (rev\, t)$$

then we find:

$$\begin{aligned}
rev(append\, l_1\ l_2) &= rev(append\ (h :: t)\ l_2)\\
&= rev(h :: (append\, t\ l_2))\\
&= append\,(rev(append\, t\ l_2))\ [h]\\
&= append\,(append\,(rev\ l_2)\ (rev\, t))\ [h]\\
&= append\,(rev\ l_2)\ (append\,(rev\, t)\ [h])\\
&= append\,(rev\ l_2)\ (rev\ (h :: t))\\
&= append\,(rev\ l_2)\ (rev\ l_1)
\end{aligned}$$

Q.E.D.

Theorem 7.6 *For any list* l *we have* $rev(rev\ l) = l$.
Proof: *Structural induction on* l.

- *If* $l = []$ *we have:*

$$\begin{aligned}
rev(rev\ l) &= rev(rev\ [])\\
&= rev\ []\\
&= []\\
&= l
\end{aligned}$$

- *Now suppose* $l = h :: t$ *and we know that*

$$rev(rev\, t) = t$$

then we find:

$$\begin{aligned}
rev(rev\ l) &= rev(rev\ (h :: t))\\
&= rev(append\,(rev\, t)\ [h])\\
&= append\,(rev\ [h])\ (rev(rev\, t))\\
&= append\,(rev\ [h])\ t\\
&= append\,(rev\ (h :: []))\ t
\end{aligned}$$

$$
\begin{aligned}
&= \mathit{append}\,(\mathit{append}\,(\mathit{rev}\,[])\,[h])\,t \\
&= \mathit{append}\,(\mathit{append}\,[]\,[h])\,t \\
&= \mathit{append}\,[h]\,t \\
&= \mathit{append}\,(h :: [])\,t \\
&= h :: (\mathit{append}\,[]\,t) \\
&= h :: t \\
&= l
\end{aligned}
$$

Q.E.D.

There are lots of theorems relating the list operations that can be proved in a similar style. Generally, one proceeds by list induction. In subtler cases, one has to split off lemmas which are themselves proved inductively, and sometimes to generalize the theorem before proceeding by induction. There are plenty of examples in the exercises for readers to try their hand at.

Further reading

Neumann (1995) catalogues the dangers arising from the use of computers in society, including those arising from software bugs. A very readable discussion of the issues, with extensive discussion of some interesting examples, is given by Peterson (1996). The general issue of software verification was at one time controversial, with DeMillo, Lipton, and Perlis (1979) among others arguing against it. A cogent discussion is given by Barwise (1989). There is now a large literature in software verification, though much of it is concerned with imperative programs. Many introductory functional programming books such as Paulson (1991) and Reade (1989) give some basic examples like the ones here. It is also worth looking at the work of Boyer and Moore (1979) on verifying properties of pure LISP functions expressed inside their theorem prover. One of the largest real proofs of the type we consider here is the verification by Aagaard and Leeser (1994) of a boolean simplifier used in VLSI logic synthesis.

Exercises

1. Prove the correctness of the more efficient program we gave for performing exponentiation:

```
#let square x = x * x;;
#let rec exp x n =
  if n = 0 then 1
  else if n mod 2 = 0 then square(exp x (n / 2))
  else x * square(exp x (n / 2));;
```

2. Recall the definition of `length`:

```
#let rec length =
   fun [] -> 0
     | (h::t) -> 1 + length t;;
```

Prove that $\mathrm{length}(\mathrm{rev}\ l) = \mathrm{length}\ l$ and that $\mathrm{length}(\mathrm{append}\ l_1\ l_2) = \mathrm{length}\ l_1 + \mathrm{length}\ l_2$.

3. Define the `map` function, which applies a function to each element of a list, as follows:

```
#let rec map f =
   fun [] -> []
     | (h::t) -> (f h)::(map f t);;
```

Prove that if $l \neq []$ then $\mathrm{hd}(\mathrm{map}\ f\ l) = f(\mathrm{hd}\ l)$, and that $\mathrm{map}\ f\ (\mathrm{rev}\ l) = \mathrm{rev}\ (\mathrm{map}\ f\ l)$. Recall the definition of function composition:

```
#let o f g = fun x -> f(g x);;
#infix "o";;
```

Prove that $\mathrm{map}\ f\ (\mathrm{map}\ g\ l) = \mathrm{map}\ (f \circ g)\ l$.

4. McCarthy's '91 function' can be defined by:

```
#let rec f x = if x > 100 then x - 10
               else f(f(x + 11));;
```

Prove that for $n \leq 101$, we have $f(n) = 91$. Pay careful attention to establishing termination. (Hint: a possible measure is $101 - x$.)

5. The problem of the Dutch National Flag is to sort a list of 'colours' (red, white and blue) so that the reds come first, then the whites and then the blues. However, the only method permitted is to swap adjacent elements. Here is an ML solution. The function `dnf` returns 'true' iff it has made a change, and this function is then repeated until no change occurs.

```
type colour = Red | White | Blue;;

let rec dnf =
   fun [] -> [],false
     | (White::Red::rest) -> Red::White::rest,true
     | (Blue::Red::rest) -> Red::Blue::rest,true
     | (Blue::White::rest) -> White::Blue::rest,true
     | (x::rest) -> let fl,ch = dnf rest in x::fl,ch;;

let rec flag l =
  let l',changed = dnf l in
  if changed then flag l' else l';;
```

For example:

```
#flag [White; Red; Blue; Blue; Red; White; White; Blue; Red];;
- : colour list =
  [Red; Red; Red; White; White; White; Blue; Blue; Blue]
```

Prove that the function `flag` always terminates with a correctly sorted list.

6. (*) Define the following functions:

```
#let rec sorted =
   fun [] -> true
     | [h] -> true
     | (h1::h2::t) -> h1 <= h2 & sorted(h2::t);;

#let rec filter p =
   fun [] -> []
     | (h::t) -> let t' = filter p t in
                 if p h then h::t' else t';;

#let sameprop p l1 l2 =
   length(filter p l1) = length(filter p l2);;

#let rec permutes l1 l2 =
   fun [] -> true
     | (h::t) -> sameprop (fun x -> x = h) l1 l2 &
                 permutes l1 l2 t;;

#let permuted l1 l2 = permutes l1 l2 l1;;
```

What do they all do? Implement a sorting function `sort` and prove that for all lists l one has $\text{sorted}(\text{sort } l) = \text{true}$ and $\text{permuted } l \ (\text{sort } l) = \text{true}$.