Introduction to Functional Programming

John Harrison

University of Cambridge

Lecture 5

Proving Programs Correct

Topics covered:

- The correctness problem
- Testing and verification
- Termination and totality
- Exponential and gcd
- Appending and reversing

The correctness problem

Programs are written to perform some particular task.

However, it is often very hard to write a program that performs its intended function — as programmers know well.

In practice, most large programs have 'bugs'.

Some bugs are harmless, others merely irritating.

They can cause financial and public relations disasters (e.g. the Pentium FDIV bug).

In some situation bugs can be deadly.

Peter Neumann: 'Computer Related Risks'.

Dangerous bugs

Some situations where bugs can be deadly include:

- Heart pacemakers
- Aircraft autopilots
- Car engine management systems and antilock braking systems
- Radiation therapy machines
- Nuclear reactor controllers

These applications are said to be *safety critical*.

Testing and verification

One good way to track down bugs is through extensive testing.

But usually there are too many possible situations to try them all exhaustively, so there may still be bugs lying undetected.

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.

An alternative is *verification*, where we try to *prove* that a program behaves as required.

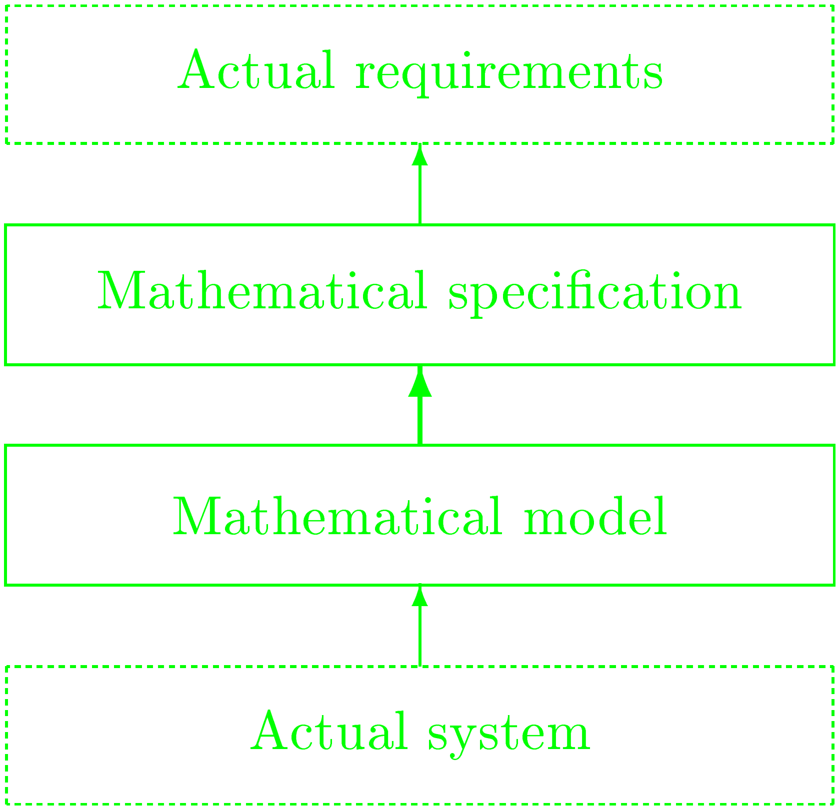
Consider ordinary mathematical theorems, like

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

We can *test* this for many particular values of N, but it is easier and more satisfactory simply to *prove* it (e.g. by induction).

The limits of verification

The enterprise of verification can be represented by this diagram:



It is only the central link that is mathematically precise. The others are still informal — all we can do is try to keep them small.

Verifying functional programs

We suggested earlier that functional programs might be easier to reason about formally, because they correspond directly to the mathematical functions that they represent.

This is arguable, but at least we will try to show that reasoning about some simple functional programs is straightforward.

We need to remember that, in general, functional programs are *partial* functions. Sometimes we need a separate argument to establish termination.

Often, the proofs proceed by induction, parallelling the definition of the functions involved by recursion.

Exponentiation (1)

Recall the following simple definition of natural number exponentiation:

```
- fun exp x n =
      if n = 0 then 1
      else x * exp x (n - 1);
```

We will prove that this satisfies the following specification:

For all $n \geq 0$ and x, exp x n terminates and exp x $n = x^n$

The function is defined by (primitive) recursion.

The proof is by (step-by-step, mathematical) induction.

Exponentiation (2)

- If $n = 0$, then by definition $\exp\ x\ n = 1$. Since for any integer x, we have $x^0 = 1$, so the desired fact is established.
- Suppose we know $\exp\ x\ n = x^n$. Because $n \geq 0$, we also know $n + 1 \neq 0$. 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.

Note that we assume $0^0 = 1$, an example of how one must state the specification precisely!

Greatest common divisor (1)

We define a function to calculate the gcd of two integers using Euclid's algorithm.

```
- fun gcd x y =
    if y = 0 then x
    else gcd y (x mod y);
```

We want to prove:

For any integers x and y, gcd x y terminates and returns a gcd of x and y.

Here we need to be even more careful about the specification. What is a gcd of two negative numbers?

Greatest common divisor (2)

We write $x|y$, pronounced 'x divides y', to mean that y is an integral multiple of x, i.e. there is some integer d with $y = dx$.

We say that d is a *common divisor* of x and y if $d|x$ and $d|y$.

We say that d is a *greatest* common divisor if:

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

Note that unless x and y are both zero, we do not specify the sign of the gcd. The specification does not constrain the implementation completely.

Greatest common divisor (3)

Now we come to the proof. The `gcd` function is no longer defined by *primitive* recursion.

In fact, `gcd x y` is defined in terms of `gcd y (x mod y)` in the step case.

We do not, therefore, proceed by step-by-step mathematical induction, but by *wellfounded* induction on $|y|$.

The idea is that this quantity (often called a *measure*) decreases with each call. We can use it to prove termination, and as a handle for wellfounded induction.

In complicated recursions, finding the right wellfounded ordering on the arguments can be tricky. But in many cases one can use this simple 'measure' approach.

Greatest common divisor (4)

Now we come to the proof. Fix some arbitrary n. We suppose that the theorem is established for all arguments x and y with $|y| < n$, and we try to prove it for all x and y with $|y| = n$. There are two cases.

First, suppose that $y = 0$. Then $\text{gcd}\ x\ y = x$ 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 immediately we get $d|x$, so x is a *greatest* common divisor.

This establishes the first part of the induction proof.

Greatest common divisor (5)

Now suppose $y \neq 0$. We want to apply the inductive hypothesis to $\mathrm{gcd}\ y\ (x \bmod y)$.

We will write $r = x \bmod y$ for short. The basic property of the `mod` function that we use 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 $d = \mathrm{gcd}\ y\ (x \bmod y)$ 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.

Append (1)

Now consider an example concerning lists rather than numbers. Define:

```
- fun append [] l = l
    | append (h::t) l = h::(append t l);
```

This is supposed to join together two lists. We want to prove that the operation is associative, i.e. for any three lists l_1, l_2 and l_3 we have:

$$\text{append}\ l_1\ (\text{append}\ l_2\ l_3) = \text{append}\ (\text{append}\ l_1\ l_2)\ l_3$$

We can proceed by induction on the length of l_1, but since the function was defined by structural recursion over lists, it is more natural to prove the theorem by *structural induction.*

The principle is: 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.

Append (2)

We proceed, then, by structural induction on l_1. There are two cases to consider. First, suppose $l_1 = []$. Then we have:

$$\begin{array}{rl} & \text{append } l_1 \ (\text{append } l_2 \ l_3) \\ = & \text{append } [] \ (\text{append } l_2 \ l_3) \\ = & \text{append } l_2 \ l_3 \\ = & \text{append } (\text{append } [] \ l_2) \ l_3 \\ = & \text{append } (\text{append } l_1 \ l_2) \ l_3 \end{array}$$

As required.

Append (3)

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

$$\text{append}\ t\ (\text{append}\ l_2\ l_3) = \text{append}\ (\text{append}\ t\ l_2)\ l_3$$

Therefore:

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

The theorem is proved.

Reverse (1)

For a final example, let us define a function to reverse a list:

```
- fun rev [] = []
    | rev (h::t) = append (rev t) [h];
> val rev = fn : 'a list -> 'a list
- rev [1,2,3];
> val it = [3, 2, 1] : int list
```

We will prove that for any list l we have:

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

This is again a structural induction. However we require two lemmas, which can also be proved by structural induction:

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

Reverse (2)

First suppose that $l = []$. Then the proof is easy:

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

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

$$\mathrm{rev}(\mathrm{rev}\ t) = t$$

Reverse (3)

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

Harder cases

Here is a difficult exercise: prove that the following terminates for $n > 0$.

```
-  fun Conway 1 = 1
     | Conway 2 = 1
     | Conway n =
         let val x = Conway(n-1)
          in Conway(x) + Conway(n-x)
         end;
> val Conway = fn : int -> int
```

Here is an unsolved problem: does the following always terminate?

```
- fun Collatz n =
    if n <= 1 then 0
    else if n mod 2 = 0 then
      Collatz(n div 2)
    else Collatz(3 * n + 1);
> val Collatz = fn : int -> int
```