

Reasoning About Programs

Reading: Thompson Ch.8

COMP1100 — Introduction to Programming and Algorithms

Malcolm Newey

Australian National University
Semester 1, 2006

The Need for Proof

- Who can program without making errors?
(Errors can be subtle. .. and they can be expensive.)
- You need to be able to demonstrate correctness.
(All cases must be covered correctly.)
- Properties of a program must be validated.
But tests can rarely be exhaustive.

You must remember this:

Testing can only show the presence of bugs.

Before we Start:-

Standard Postmortem Questions

- Did I pass the mid-semester exam?
Your marks are available through STREAMS
- Where does my score place me in the class?
Vital statistics are:
Mean = 69;
Distribution: 42% HD, 13% D, 15% Cr, 9% P, 22% Fail
- Did I do well?
Marks must be interpreted by YOU
- Are the results for the class acceptable?
Yes! The staff are very happy.

Proof and Haskell

- Proof is easier in functional languages.
- Clauses in a function definition specify computations
- They can also be used as mathematical properties.
- Symbolic execution can be used to show general properties.
- Induction is flip-side of Recursion.
This week is mostly about Induction. It is a powerful technique .

Typical Proof by Induction

Theorem: For all natural numbers, n , the number $n^2 + n$ is even.

Proof:

- **Base Case**

$0^2 + 0$ is clearly even (by evaluation).

- **Step Case**

Assume $n^2 + n$ is even.

$$(n+1)^2 + (n+1) = n^2 + 2n + 1 + n + 1 = (n^2 + n) + 2(n+1)$$

which is the sum of two even numbers and so is even.

(We know the first is even by the induction hypothesis.)

In summary, $n^2 + n$ is even implies $(n+1)^2 + (n+1)$ is too.

Proving Properties of Recursive Functions

Suppose that a function t is defined recursively on positive numbers as follows

$$t(1) = 1$$

$$t(n) = t(n-1) + n*n!$$

Theorem: $t(n) = (n+1)! - 1$

Proof:

- **The Base Case:**

$$\text{LHS} = t(1) = 1$$

$$\text{RHS} = 2! - 1 = 1 = \text{LHS}$$

Why is $n=1$ in the base case (and not 0)?

Because t is only defined for +ve integers.

- **The Step Case:**

The induction hypothesis is $t(n) = (n+1)! - 1$

The goal for proof is $t(n+1) = ((n+1)+1)! - 1$

$$\begin{aligned} \text{LHS} &= t(n+1) \\ &= t((n+1)-1) + (n+1)*(n+1)! \\ &= t(n) + (n+1)*(n+1)! \\ &= (n+1)! - 1 + (n+1)*(n+1)! \\ &= (n+2)! - 1 \end{aligned}$$

$$\text{RHS} = ((n+1)+1)! - 1 = \text{LHS}$$

QED

Induction for Natural Numbers

To prove a property $P(n)$ for all natural numbers n we must do these two things:

- **The Base Case:** Prove $P(0)$.
- **The Step Case:** Prove $P(n+1)$ on the *assumption* that $P(n)$ holds.

In the step case $P(n)$ is referred to as the **induction hypothesis**.

- **The Step Case:**

The induction hypothesis is $t(n) = (n+1)! - 1$

The goal for proof is $t(n+1) = ((n+1)+1)! - 1$

$$\begin{aligned} \text{LHS} &= t(n+1) \\ &= t((n+1)-1) + (n+1)*(n+1)! \\ &= t(n) + (n+1)*(n+1)! \\ &= (n+1)! - 1 + (n+1)*(n+1)! \\ &= (n+2)! - 1 \end{aligned}$$

$$\text{RHS} = ((n+1)+1)! - 1 = \text{LHS}$$

QED

Proving Properties of Recursive Functions

Suppose we were to define multiplication in terms of addition, coding it as a Haskell function:

```
(*) :: Int -> Int -> Int
m*n
  n==0 = 0                (base case)
  n>0  = m*(n-1) + m     (recursive case)
```

Lemmas:

```
m*0 = 0                prop. 1
m*(n+1) = m*n + m     (for n>0) prop. 2
```

(These re-express the clauses of the function definition.)

Theorem: $0*m = 0$

Proving Properties of List Functions

The length of a list can be calculated using this function definition:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length(xs)
```

The most basic property of the function is that the value returned can never be negative. This can be *stated formally* and *proven* as follows:

Theorem: $\text{length}(xs) \geq 0$

Proof:

- **The Base Case:**

$$\text{LHS} = 0*0 = 0 = \text{RHS}$$

- **The Step Case:**

The induction hypothesis is $0*n=0$ for $n>0$

The goal for proof is $0*(n+1) = 0$

$$\begin{aligned} \text{LHS} &= 0*(n+1) \\ &= 0*n + 0 && \text{(using prop. 2)} \\ &= 0 + 0 && \text{(using ind. hyp.)} \\ &= 0 = \text{RHS} \end{aligned}$$

QED

Proof:

- **The Base Case:**

$$\begin{aligned} (\text{length } []) &\geq 0 \\ &= 0 \geq 0 \\ &= \text{True} \end{aligned}$$

- **The Step Case:**

The induction hypothesis is $\text{length}(xs) \geq 0$

The goal to be proved is $\text{length}(x:xs) \geq 0$
and its proof is this:

$$\begin{aligned} \text{length}(x:xs) &\geq 0 \\ &= 1 + \text{length}(xs) \geq 0 \\ &= \text{True} \end{aligned}$$

QED

Structural Induction for Lists

To prove a property $P(xs)$ for all *finite* lists xs we must do these two things:

- **The Base Case:** Prove $P([])$.
- **The Step Case:** Prove $P(x:xs)$ on the *assumption* that $P(xs)$ holds.

In the step case $P(xs)$ is referred to as the **induction hypothesis**.

Another Example

- If we append a list of length 9 to one of length 7, the result will be a list of length 16.
- That illustrates a fundamental characteristic of the length function:
$$\text{length}(xs++ys) = (\text{length } xs) + (\text{length } ys)$$
- It doesn't depend on the type of the elements.

Induction on Lists

- Note that the induction principle for lists has identical structure to that for natural numbers.
- As with mathematical induction, structural induction for lists is a good proof technique for recursively defined functions.
- In later courses you will encounter other structural induction principles - induction trees, for example

Proof of 'length-of-append' Theorem

Here is the definition of append:

$$\begin{aligned} (++) &:: [a] \rightarrow [a] \rightarrow [a] \\ [] ++ ys &= ys \\ (x:xs) ++ ys &= x:(xs ++ ys) \end{aligned}$$

Theorem: $\text{length}(xs ++ ys) = \text{length}(xs) + \text{length}(ys)$

Proof:

- **The Base Case:**
$$\begin{aligned} \text{LHS} &= \text{length}([] ++ ys) \\ &= \text{length}(ys) \\ \text{RHS} &= \text{length}([]) ++ \text{length}(ys) \\ &= 0 + \text{length}(ys) = \text{LHS} \end{aligned}$$

- **The Step Case:**

The induction hypothesis is

$$\text{length}(xs ++ ys) = \text{length}(xs) + \text{length}(ys)$$

The goal for proof is

$$\text{length}((x:xs) ++ ys) = \text{length}(x:xs) + \text{length}(ys)$$

$$\begin{aligned} \text{LHS} &= \text{length}((x:xs) ++ ys) \\ &= 1 + \text{length}(xs ++ ys) \\ &= 1 + \text{length}(xs) + \text{length}(ys) \end{aligned}$$

$$\begin{aligned} \text{RHS} &= \text{length}(x:xs) + \text{length}(ys) \\ &= 1 + \text{length}(xs) + \text{length}(ys) = \text{LHS} \end{aligned}$$

QED

Yet Another Example

We now make use of the even-ness predicate which is standard in Haskell and also the map function.

- $(\text{map even } [1,2,3,4]) = [\text{False}, \text{True}, \text{False}, \text{True}]$
- $(\text{map even } [1,2]) = [\text{False}, \text{True}]$ and $(\text{map even } [3,4]) = [\text{False}, \text{True}]$
- $\text{map even } ([1,2] ++ [3,4]) = (\text{map even } [1,2]) ++ (\text{map even } [3,4])$

A generalization should be obvious. It is:

$$\text{map even } (xs ++ ys) = (\text{map even } xs) ++ (\text{map even } ys)$$

What is the Proof Strategy

See page 144 of Thompson's text.

- State clearly the goal of the induction and the main subgoals, namely the base case and the step case.
- If any confusion is possible, change variable name in the definitions to be used.
- Use clauses from the function definitions to simplify the subgoals. If a subgoal is an equation, simplify its left and right hand sides separately.
- You should expect to use the induction hypothesis in your proof.

Proof of map Example

Here is the definition of map:

$$\begin{aligned} \text{map} &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{map } f \ [] &= [] \\ \text{map } f \ (x:xs) &= (f x) : (\text{map } f \ xs) \end{aligned}$$

Theorem: $\text{map } f \ (xs ++ ys) = (\text{map } f \ xs) ++ (\text{map } f \ ys)$

Proof:

- **The Base Case:**

$$\begin{aligned} \text{LHS} &= \text{map } f \ ([] ++ ys) \\ &= \text{map } f \ ys \\ \text{RHS} &= (\text{map } f \ []) ++ (\text{map } f \ ys) \\ &= [] ++ (\text{map } f \ ys) = \text{LHS} \end{aligned}$$

- **The Step Case:**

The induction hypothesis is

$$\text{map } f \text{ (xs ++ ys)} = (\text{map } f \text{ xs}) ++ (\text{map } f \text{ ys})$$

The goal for proof is

$$\text{map } f \text{ ((x:xs) ++ ys)} = (\text{map } f \text{ (x:xs)}) ++ (\text{map } f \text{ ys})$$

$$\begin{aligned} \text{LHS} &= \text{map } f \text{ ((x:xs) ++ ys)} \\ &= \text{map } f \text{ (x:(xs ++ ys))} \\ &= (f \text{ x}):(\text{map } f \text{ (xs ++ ys)}) \\ &= (f \text{ x}):((\text{map } f \text{ xs}) ++ (\text{map } f \text{ ys})) \end{aligned}$$

$$\begin{aligned} \text{RHS} &= (\text{map } f \text{ (x:xs)}) ++ (\text{map } f \text{ ys}) \\ &= ((f \text{ x}):(\text{map } f \text{ xs})) ++ (\text{map } f \text{ ys}) \\ &= (f \text{ x}):((\text{map } f \text{ xs}) ++ (\text{map } f \text{ ys})) = \text{LHS} \end{aligned}$$

QED

Time for Perspective

- The use of induction is clearly a very solid basis for making an argument that some property holds.
- We have seen some examples that you might say are ‘almost automatic’.
- There are decision procedures which will produce many such proofs completely mechanically.
- They can't prove everything.

Surely not *Another Example*

- We've done three simple ones quite successfully
- What about harder ones?
No. They make for boring lecture material.
- The Thompson text book (page 146) proves that:
$$\text{reverse(xs++ys)} = (\text{reverse ys}) ++ (\text{reverse xs})$$

A Hard Problem

- It is a theorem about the reverse function that you may well say is ‘obvious’.
$$\text{reverse}(\text{reverse}(xs)) = xs$$
- The next slide shows why it is hard.
- The text book gives a way of proving it (page 149) but it relies on redefining the reverse function and using another mathematical technique - *generalization*.
- This section of the text is not examinable in COMP1100.

Attempted Proof of the Rev-Rev Property

Here is the definition of reverse:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

Theorem to be proved: $\text{reverse}(\text{reverse}(xs)) = xs$

Proof: The base case is easy;

$$\text{reverse}(\text{reverse } []) = \text{reverse } [] = []$$

so we focus on the step case:

- **The Induction Hypothesis:**

$$\text{reverse}(\text{reverse}(xs)) = xs$$

- **The Goal in the Step Case:**

$$\text{reverse}(\text{reverse}(x:xs)) = x:xs$$

- **The attempt at simplification:**

$$\begin{aligned} \text{LHS} &= \text{reverse}(\text{reverse}(x:xs)) \\ &= \text{reverse}((\text{reverse } xs) ++ [x]) \\ &= \text{????} \end{aligned}$$

- So we find the RHS is in its simplest form and there are no definitions that simplify the left side.

We are apparently at a dead end.