# Chapter 9
# Reasoning about programs



We gave an introduction to proof in Section 1.14, where we said that a *proof* is an argument that a proposition holds. Often a proposition will be general in saying that something holds *for all* things of a certain sort. In mathematics we might give a proof of Pythagoras' theorem, which states that there is a relationship $a^2 = b^2 + c^2$ between the sides of all right-angled triangles.

In programming we can prove that programs have a particular property for all input values. Having a proof means that we can be certain that the program will behave as we require whatever the conditions. Compare this with program testing: a test assures us that a program behaves as it should on a particular collection of input values; it can only be an act of faith to infer from this that the program behaves as expected on every possible input, and no mathematician would accept a proposition as valid simply because it holds for a limited set of test data.

Property-based testing – as given by QuickCheck – provides much better coverage, but it is still possible that the places where an error occurs could be missed by randomly generating data; proof avoids this, and gives an error 'nowhere to hide': Figure 1.5 on page 24 illustrates this. Even when we are sure that using QuickCheck shows that a particular property holds, a proof can tell us *why* that property holds, not just that it is true.

Central to applying proof within functional programming is the insight that we can read function definitions as *logical descriptions* of what they do; we discuss this in depth at the start of the chapter. After looking again at the relationship of reasoning, testing and property-based testing, we look at some background topics in programming and logic, before introducing the central idea of proof by induction over finite lists.

Proofs by induction follow a pattern, and we illustrate this by giving a sequence of examples. We also supply advice on how to go about finding induction proofs. Finally we look at the QuickCheck properties that correspond to the propositions we prove, and show how to check them using QuickCheck. The chapter concludes with a more challenging example of proof, which you can omit on a first reading.

## 9.1 Understanding definitions

Suppose that we ask ourselves the seemingly obvious question: 'how do we understand what a function does?' There are various ways of answering this.

- We can evaluate what the function does on particular inputs, using an implementation like GHCi.
- We can do the same thing by hand, performing a line-by-line calculation. This has the advantage of letting us see how the program gets to its result, but the disadvantage of being slow and impractical for all but the smallest of programs.
- We can try to argue about how the program behaves in general.

The third answer, in which we reason about the behaviour of our programs, is the subject of this chapter, which builds on the introduction of Section 1.14.

Consider a simple functional program like

```
length []     = 0                                    (length.1)
length (x:xs) = 1 + length xs                        (length.2)
```

Using the definition we can calculate the length of any particular list like [2,3,1]

```
length [2,3,1]
⇝  1 + length [3,1]                                  by (length.2)
⇝  1 + (1 + length [1])                              by (length.2)
⇝  1 + (1 + (1 + length []))                         by (length.2)
⇝  1 + (1 + (1 + 0))                                 by (length.1)
⇝  3
```

We can also read `(length.1)` and `(length.2)` as **descriptions** of how `length` behaves in general.

- `(length.1)` says what `length []` is;
- `(length.2)` says that **whatever values** of `x` and `xs` we choose, `length (x:xs)` will be equal to `1 + length xs`.

In the second case we have a **general property** of `length`: it states something about how length behaves on all non-empty lists. On the basis of these equations we can conclude that

```
length [x] = 1                                       (length.3)
```

How do we do that? We know that `(length.2)` holds for **all** values of `x` and `xs`, and so it will hold in particular when `xs` is replaced by `[]`, so

```
length [x]
  = length (x:[])                                    by defn of [x]
  = 1 + length []                                    by (length.2)
  = 1 + 0                                            by (length.1)
  = 1
```

The lesson of this discussion is that we can read a function definition in (at least) two different ways.

- We can take the definition as describing how to compute particular results, such as `length [2,3,1]`.
- We can also take the definition as a **general description** of the behaviour of the function in question.

From this general description we are able to deduce other facts, some like `(length.3)` being utterly straightforward, and others like

```
length (xs ++ ys) = length xs + length ys                    (length.4)
```

expressing more complicated interactions between two or more functions. We will prove `(length.4)` in Section 9.6.

Another way of looking at the proof of `(length.3)` above is that we are doing **symbolic evaluation**; rather than evaluating `length` at a particular value like `[2]` we have replaced the number 2 with a variable `x`, but used the evaluation rules in exactly the way that we used them earlier. We will find that symbolic evaluation forms an important part of our proofs, but we will need to use another principle – induction – to do most proofs for recursive functions.

To conclude this introduction, we have seen that functional programs 'describe themselves' in a direct way. If you are familiar with an imperative language like Pascal, C or Java, think how you might convince yourself of the analogues of `(length.3)` or `(length.4)` for programs written in that language. It's very difficult to see how you might state these properties, and even more difficult to work out how to prove them valid.

## 9.2 Testing and proof

When we introduced program testing in Section 4.8 we looked at the example

```
mysteryMax :: Integer -> Integer -> Integer
mysteryMax x y z
  | x > y && x > z      = x
  | y > x && y > z      = y
  | otherwise           = z
```

which was an attempted solution to the problem of finding the maximum of three integers.

If I asked you to give me five sets of test data for the function, and for you to test the function at those points, I would guess that you would conclude that the implementation works: try it!

We can write a **property** that expresses that the function does as it should, like this:

```
prop_mystery :: Integer -> Integer -> Integer -> Bool

prop_mystery x y z =
    mysteryMax x y z == (x `max` y) `max` z
```

Let's see what happens when we check this property using QuickCheck:

```
*Chapter8> quickCheck prop_mystery
*** Failed! Falsifiable (after 91 tests and 2 shrinks):
75
75
0
*Chapter8 Test.QuickCheck> quickCheck prop_mystery
*** Failed! Falsifiable (after 4 tests and 1 shrink):
3
3
0
*Chapter8> quickCheck prop_mystery
+++ OK, passed 100 tests.
```

The first time we check it, it takes 91 tests to find the error; in the second case we find it much more quickly, but in the third we don't catch it at all!

Now let's try to **prove** that the function behaves as it should. We need to look at various *cases* of the ordering of the values. If we first look at the cases

```
x > y && x > z
y > x && y > z
z > x && z > y
```

then in each of these `mysteryMax` will produce the correct solution. In the other cases, at least two of the three arguments are equal. If all three are equal,

```
x == y && y == z
```

the function also operates correctly. Finally, we start to look at the cases where precisely two elements are equal. The function behaves correctly when

```
y == z && z > x
```

but in the case of

```
x == y && y > z
```

we can see that the result will, erroneously, be z.

Now, we can see this process of attempting to prove a result as a general way of testing the function – it is a form of **symbolic testing** which will consider all cases in turn, at least until an error is found. We can therefore see that reasoning can give us a powerful way of debugging programs by focusing on the reason why we cannot complete a proof of correctness, as well as the more traditional view that a proof shows that a program meets the requirements put upon it, if it does.

On the other hand, as we mentioned in Section 4.8, finding a proof is a difficult enterprise, and so there are clearly roles for proof, property-based testing and traditional testing in the development of reliable software.

## 9.3 Definedness, termination and finiteness

Before we say anything more about proof, we need to talk about two aspects of programming upon which we have only touched so far.

### Definedness and termination

Evaluating an expression can have one of two outcomes:

- the evaluation can halt, or **terminate**, to give an answer; or
- the evaluation can go on forever.

If we make the definition

```
fact :: Integer -> Integer
fact n
  | n==0          = 1
  | otherwise     = n * fact (n-1)
```

then examples of the two are given by the expressions

```
fact 2                    fact (-2)
```

since in the latter case

```
fact (-2)
⇝  (-2) * fact (-3)
⇝  (-2) * ((-3) * fact (-4))
⇝  ...
```

In the case that evaluation goes on for ever, we say that the value of the expression is **undefined**, since no defined result is reached. In writing proofs we often have to confine our attention to cases where a value is **defined**, since it is only for defined values that many familiar properties hold. One of the simplest examples is given by the expression

```
0*e
```

which we expect to be `0` irrespective of the value of `e`. That is certainly so if `e` has a defined value, but if `e` is `fact (-2)`, the value of

```
0 * fact (-2)
```

will be undefined and *not* zero.

In many of the proofs we give, we state that results hold for all defined values. This restriction does not cause problems in practice, since the defined cases will be exactly those which interest us the vast majority of the time. An undefined value *is* of interest when a function does not give a defined value when it is expected to – a case of symbolic debugging.

### Finiteness

We have said nothing so far about the order in which expressions are evaluated in Haskell. In fact, Haskell evaluation is **lazy**, so that arguments to functions are only evaluated if their values are actually needed. This gives some Haskell programs a distinctive flavour, which we explore in depth in Chapter 17. What is important for us here is that lazy evaluation allows the definition and use of **infinite** lists like

```
[1,2,3, ... ]
```

and **partially defined** lists. In what follows we will mainly confine our attention to **finite** lists, by which we mean lists which have a defined, finite length and defined elements. Examples are

```
[]          [1,2,3]          [[4,5],[3,2,1],[]]
```

Reasoning about lazy programs is discussed explicitly in Section 17.9 below.

### Exercises

**9.1** Given the definition of `fact` above, what are the results of evaluating the following expressions?

```
(4 > 2) || (fact (-1) == 17)
(4 > 2) && (fact (-1) == 17)
```

Discuss the reasons why you think that you obtained these answers.

**9.2** Give a definition of a multiplication function

```
mult :: Integer -> Integer -> Integer
```

so that `mult 0 (fact (-2))` evaluates to 0.
What is the result of `mult (fact (-2)) 0` for your function? Explain why.

## 9.4 A little logic

In order to appreciate how to reason about functional programs we need not have a background in formal logic. Nevertheless, it is worth discussing two aspects of logic before we proceed with our proofs.

### Assumptions in proofs

First, we look at the idea of proofs which contain **assumptions**. Taking a particular example, it follows from elementary arithmetic that if we *assume* that petrol costs 27 pence per litre, then we can prove that four litres will cost £1.08.

What does this tell us? It does *not* tell us outright how much four litres will cost; it only tells us the cost *if the assumption is valid*. To be sure that the cost will be £1.08, we need to supply some evidence that the assumption is justified: this might be another proof – perhaps based on petrol costing £1.20 per gallon – or direct evidence.

We can write what we have proved as a formula,

1 litre costs 27 pence $\Rightarrow$ 4 litres cost £1.08

where the arrow, $\Rightarrow$, which is the logical symbol for **implication**, says that the second proposition **follows from** the first.

As we have seen, we prove an implication like $A \Rightarrow B$ by assuming $A$ in proving $B$. If we then find a proof of $A$, then knowing the implication will guarantee that $B$ is also valid.

Yet another way of looking at this is to see a proof of $A \Rightarrow B$ as a *process* for turning a proof of $A$ into a proof of $B$. We use this idea in proof by induction, as one of the tasks in building an induction proof is the induction step, where we prove that one property holds assuming another.

### Free variables and quantifiers

When we write an equation like

```
square x = x*x
```

it is usually our intention to say that this holds **for all** (defined) values of the free variable `x`. If we want to make this 'for all' explicit we can use a **quantifier** like this

```
∀x (square x = x*x)
```

where we read the universal quantifier, '`∀x`', as saying 'for all `x`...'.

We now turn to induction, the main technique we use for proving properties of programs.

## 9.5 Induction

In Chapter 7 we saw that a general method for defining lists was primitive recursion, as exemplified by

```
sum :: [Integer] -> Integer
sum []     = 0                    (sum.1)
sum (x:xs) = x + sum xs           (sum.2)
```

Here we give a value outright at `[]`, and define the value of `sum (x:xs)` using the value `sum xs`. Structural induction is a proof principle which states:

**Definition 9.1** *Principle of structural induction for lists*

*In order to prove that a logical property* `P(xs)` *holds for all* ***finite*** *lists* `xs` *we have to do two things.*

- **Base case.** *Prove* `P([])` *outright.*
- **Induction step.** *Prove* `P(x:xs)` *on the* assumption *that* `P(xs)` *holds.*
  *In other words* `P(xs)` $\Rightarrow$ `P(x:xs)` *has to be proved.*
  *The* `P(xs)` *here is called the* ***induction hypothesis*** *since it is assumed in proving* `P(x:xs)`.

It is interesting to see that this is just like primitive recursion, except that instead of building the values of a function, we are building up the parts of a proof. In both cases we deal with `[]` as a basis, and then build the general thing by showing how to go from `xs` to `(x:xs)`. In a function definition we define `fun (x:xs)` using `fun xs`; in the proof of `P(x:xs)` we are allowed to use `P(xs)`.

### Justification

Just as we argued that recursion was not circular, so we can see proof by induction building up the proof for all finite lists in stages. Suppose that we are given proofs of `P([])` and `P(xs)` $\Rightarrow$ `P(x:xs)` for all `x` and `xs` and we want to show that `P([1,2,3])`. The list `[1,2,3]` is built up from `[]` using cons like this,

```
1:2:3:[]
```

and we can construct the proof of `P([1,2,3])` in a way which mirrors this step-by-step construction,

- `P([])` holds;
- `P([])` $\Rightarrow$ `P([3])` holds, since it is a case of `P(xs)` $\Rightarrow$ `P(x:xs)`;
- Recall our discussion of '$\Rightarrow$' above; if we know that both `P([])` $\Rightarrow$ `P([3])` and `P([])` hold, then we can infer that `P([3])` holds.
- `P([3])` $\Rightarrow$ `P([2,3])` holds, and so for similar reasons we get `P([2,3])`.
- Finally, because `P([2,3])` $\Rightarrow$ `P([1,2,3])` holds, we see that `P([1,2,3])` holds.

This explanation is for a particular finite list, but will work for any finite list: if the list has `n` elements, then we will have `n+1` steps like the four above. To conclude, this shows that we get `P(xs)` for every possible **finite** list `xs` if we know that both requirements of the induction principle hold.

### A first example

We have mentioned the definition of `sum`; recall also the function to double all elements of a list

```
doubleAll []     = []                                    (doubleAll.1)
doubleAll (z:zs) = 2*z : doubleAll zs                    (doubleAll.2)
```

Now, how would we expect `doubleAll` and `sum` to interact? If we `sum` a list after doubling all its elements, we would expect to get the same result as by doubling the sum of the original list:

```
sum (doubleAll xs) = 2 * sum xs                          (sum+dblAll)
```

We can make this a QuickCheck property, like this:

```
prop_SumDoubleAll :: [Integer] -> Bool
```

```
prop_SumDoubleAll xs =
    sum (doubleAll xs) == 2 * sum xs
```

which is identical except that we use the Boolean equality '==' rather than the mathematical equality '='; the property repeatedly passes 100 tests when we QuickCheck it.

### Setting up the induction

How are we to prove this for all `xs`? According to the principle of structural induction we get two induction goals. The first is the base case

```
sum (doubleAll []) = 2 * sum []                                    (base)
```

The second is the induction step, in which we have to prove

```
sum (doubleAll (x:xs)) = 2 * sum (x:xs)                            (ind)
```

using the induction hypothesis

```
sum (doubleAll xs) = 2 * sum xs                                    (hyp)
```

In all proofs that follow we will label the cases by `(base)`, `(ind)` and `(hyp)`.

### The base case

We are required to prove `(base)`: how do we start? The only resources we have are the equations `(sum.1)`, `(sum.2)`, `(doubleAll.1)` and `(doubleAll.2)`, so we have to concentrate on using these. As we are trying to prove an equation, we can think of simplifying the two sides separately, so working with the left-hand side first,

```
sum (doubleAll [])
  = sum []                                          by (doubleAll.1)
  = 0                                                     by (sum.1)
```

Looking at the right-hand side, we have

```
2 * sum []
  = 2 * 0                                                 by (sum.1)
  = 0                                                           by *
```

This shows that the two sides are the same, and so completes the proof of the base case.

### The induction step

Here we are required to prove `(ind)`. As in the base case we have the defining equations of `doubleAll` and `sum`, but we also can – and usually *should* – use the induction hypothesis `(hyp)`.

We work as we did in the base case, simplifying each side as much as we can using the defining equations. First the left-hand side,

```
sum (doubleAll (x:xs))
  = sum (2*x : doubleAll xs)                        by (doubleAll.2)
  = 2*x + sum (doubleAll xs)                              by (sum.2)
```

and then the right

```
2 * sum (x:xs)
  = 2 * (x + sum xs)                    by (sum.2)
  = 2*x + 2 * sum xs                    by arith.
```

Now, we have simplified each side using the defining equations. The last step equating the two is given by the induction hypothesis (`hyp`), which can be used to carry on the simplification of the left-hand side, giving

```
sum (doubleAll (x:xs))
  = sum (2*x : doubleAll xs)
  = 2*x + sum (doubleAll xs)
  = 2*x + 2 * sum xs                    by (hyp)
```

and so this final step makes the left- and right-hand sides equal, on the assumption that the induction hypothesis holds. This completes the induction step, and therefore the proof itself. ■

We use the box, ■, to signify the end of a proof.

### Finding induction proofs

Looking at the previous example, we can glean a number of pieces of advice about how to find proofs of properties of recursively defined functions.

- As a first step, it is a good idea to **state the goal as a QuickCheck property**. Once we have that we can check that it should be possible to prove it: if we find a counterexample then we'd better try to reformulate the goal, and not waste our time trying to prove something that doesn't hold!
- State clearly the goal of the induction and the two sub-goals of the induction proof: (`base`) and (`hyp`) ⇒ (`ind`).
- If any confusion is possible, change the names of the variables in the relevant definitions so that they are different from the variable(s) over which you are doing the induction.
- The only resources available are the definitions of the functions involved and the general rules of arithmetic. Use these to simplify the sub-goals. If the sub-goal is an equation, then simplify each side separately.
- In the case of the induction step, (`ind`), you should expect to use the induction hypothesis (`hyp`) in your proof; if you do not, then it is most likely that your proof is incorrect.
- Label each step of your proof with its justification: this is usually one of the defining equations of a function.

In the next section we look at a series of examples.