

Lecture 6: Proving and Testing Program Properties

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

November 28, 2016

Levels of Correctness Assurance (1/2)

1. **Manual testing** while developing.

- As the software evolves, things that worked earlier might break.

2. **Unit testing with hard-coded test cases.** The classic way of testing software.

- If a software system has many unit tests, it is likely to catch unexpected consequences of new changes to the code base.
- Recall the **Polynomials project**. Given a function $\text{diff} :: \text{Poly } a \rightarrow \text{Poly } a$ (differentiate a polynomial) we could write the unit tests
 - $\text{diff } [] = []$
 - $\text{diff } [1] \equiv []$
 - $\text{diff } [1, 2, 3] \equiv [2, 6]$
 - ...

Levels of Correctness Assurance (2/2)

3. Property-based unit-testing with randomly generated test-cases.

- The system will generate test cases the programmer might not have thought about.
- The intention of the tests is clearer to the reader, since they are specified in terms of properties, instead of test cases:
 - $\lambda p \rightarrow \text{deg} (\text{diff } p) \equiv \text{deg } p - 1$
 - $\lambda p \rightarrow (\text{diff} \circ \text{int}) p \equiv p$
 - $\lambda p q \rightarrow \text{diff } (p \text{ 'compose' } q) \equiv ((\text{diff } p) \text{ 'compose' } q) \text{ 'mul' } (\text{diff } q)$
 - ...

4. Formal proofs of properties - the program is correct for *any* input!

- Proofs can be implemented in a “proof management system”, and verification can be run whenever the programmer wants, just like unit testing
- It is a **lot** of work to formally prove correctness. Usually only used by compiler writers etc.

How to formally prove program properties?

Even though it is a lot of work to formally prove correctness of bigger systems, the basic principles are quite simple. This is the topic of today.

Consider the boolean function

$$(\wedge) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$$

$$\text{True} \wedge x = x$$

$$\text{False} \wedge x = \text{False}$$

How do we prove the following?

- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
- $x \wedge y \equiv y \wedge x$

First example

Recall the *map* function for the list data type $[a]$:

$$\text{map } f [] = []$$

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

It should hold that

- for any list xs , $\text{map } id \ xs = xs$
- for any list xs , $\text{map } (f \circ g) \ xs = \text{map } f \ (\text{map } g \ xs)$

Let's prove the first law:

- **Base case:** We want to show

$$\text{map } id [] = []$$

- **Inductive case:** Assume that $\text{map } id \ xs = xs$.

Then we want to show

$$\text{map } id (x : xs) = (x : xs)$$

The second law

- **Base step:** Prove $\text{map } (f \circ g) [] = \text{map } f (\text{map } g [])$
- **Induction hypothesis:** Assume
 $\text{map } (f \circ g) xs = \text{map } f (\text{map } g xs)$
- **Induction step:** Prove
 $\text{map } (f \circ g) (x : xs) = \text{map } f (\text{map } g (x : xs))$

Base step: Left hand side (LHS) and right hand side (RHS) match:

$$\begin{aligned}\text{map } (f \circ g) [] &= [] \\ \text{map } f (\text{map } g []) &= \text{map } f [] = []\end{aligned}$$

The second law

- **Base step:** Prove $\text{map } (f \circ g) [] = \text{map } f (\text{map } g [])$
- **Induction hypothesis:** Assume
 $\text{map } (f \circ g) xs = \text{map } f (\text{map } g xs)$
- **Inductive step:** Prove
 $\text{map } (f \circ g) (x : xs) = \text{map } f (\text{map } g (x : xs))$

Induction step:

$$\begin{aligned}\text{map } (f \circ g) (x : xs) &= (f \circ g) x : \text{map } (f \circ g) xs \\ &= f (g x) : \text{map } (f \circ g) xs \\ &= f (g x) : \text{map } f (\text{map } g xs) \\ &= \text{map } f (g x : \text{map } g xs) \\ &= \text{map } f (\text{map } g (x : xs))\end{aligned}$$

Another example: *reverse*

Recall the definitions

$$(\text{++}) :: [a] \rightarrow [a] \rightarrow [a]$$

$$[] \text{ ++ } ys = ys$$

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

$$\text{rev} :: [a] \rightarrow [a]$$

$$\text{rev} [] = []$$

$$\text{rev} (x : xs) = \text{rev } xs \text{ ++ } [x]$$

(just above is the (inefficient) definition of *reverse* (here: shortened to *rev*). We want to show the following properties:

$$(xs \text{ ++ } ys) \text{ ++ } zs \equiv xs \text{ ++ } (ys \text{ ++ } zs)$$

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

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

We proceed by induction.

Recall that there are two versions of *reverse*

- $rev1 = foldr (\lambda x rxs \rightarrow rxs \# [x]) []$
- $rev2 = foldl (flip (:)) []$

These are equal due to the **Second Duality Theorem**, stated here:

Suppose \oplus , \otimes and e are such that for all x , y , and z we have

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

In other words, \oplus and \otimes associate with each other, and e on the right of \oplus is equivalent to e on the left of \otimes . Then

$$foldr (\oplus) e xs = foldl (\otimes) e xs$$

for all finite lists xs .

Show that the preconditions of the theorem applies

Let

$$(\oplus) = (\lambda x \text{ rxs} \rightarrow \text{rxs} \dashv\vdash [x])$$

$$(\otimes) = (\textit{flip} \, (:))$$

$$e = []$$

Show that \oplus , \otimes and e are such that for all x, y , and z we have

$$x \oplus (y \otimes z) = (x \oplus y) \otimes z$$

$$x \oplus e = e \otimes x$$

- LHS1: $x \oplus (y \otimes z) = (y \otimes z) \uplus [x] = (z : y) \uplus x$
- RHS1: $(x \oplus y) \otimes z = z : (x \oplus y) = z : (y \uplus x) = (z : y) \uplus x$
- LHS2: $x \oplus e = [] \uplus [x] = [x]$
- RHS2: $e \otimes x = x : [] = [x]$

Showing the theorem itself...

Recall

$$\text{foldr } f \ z \ [] = z$$

$$\text{foldr } f \ z \ (x : xs) = f \ x \ (\text{foldr } f \ z \ xs)$$

$$\text{foldl } f \ z \ [] = z$$

$$\text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs$$

Theorem. Assume \oplus and \otimes associate with each other, and e on the right of \oplus is equivalent to e on the left of \otimes . Then

$$\text{foldr } (\oplus) \ e \ xs = \text{foldl } (\otimes) \ e \ xs$$

for all finite lists xs .

We need an auxillary lemma:

$$x \oplus \text{foldl } (\otimes) \ z \ xs = \text{foldl } (\otimes) \ (x \oplus z) \ xs$$

The First Duality Theorem

The Second Duality Theorem is a special case of “the first duality theorem”:

Suppose \oplus is associative with unit e . Then

$$\textit{foldr} (\oplus) e xs = \textit{foldl} (\oplus) e xs$$

for all finite lists xs .

Monoids

A type t is a Monoid, if there exists an operation $(\oplus) :: t \rightarrow t \rightarrow t$ and an identity element $e :: t$, s.t.

- for all $x :: t$, $x \oplus e = x = e \oplus x$
- for all $x, y, z :: t$, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$

In Haskell, there is the type class **Data.Monoid**. Here,

- the operation is called *mappend*,
- the identity element is called *mempty*
- the fold is called *mconcat*

Can you come up with examples of Monoids?

A few monoids

instance *Monoid* [a] **where**

mempty = []

mappend = (++)

[1,2,3] 'mappend' [4,5,6] = [1,2,3,4,5,6]

instance (*Monoid* a, *Monoid* b) \Rightarrow *Monoid* (a, b) **where**

mempty = (*mempty*, *mempty*)

(a1, b1) 'mappend' (a2, b2) =

(a1 'mappend' a2, b1 'mappend' b2)

([1,3], "le") 'mappend' ([3,7], "et") = ([1,3,3,7], "leet")

A few monoids

-- lexicographical ordering

instance *Monoid Ordering* **where**

mempty = EQ

LT 'mappend' _ = LT

EQ 'mappend' y = y

GT 'mappend' _ = GT

compare 1 1 'mappend' *compare* 2 2 'mappend' *compare* 1 3 = LT

What about infinite lists?

To summarize the proof structure we have seen so far:

To show that a property P holds for every finite list xs , we need to show

1. $P ([])$
2. For any finite list xs , $P (xs) \Rightarrow P (x : xs)$

We would now like to extend this to infinite lists.

What about infinite lists?

Studies of denotational semantics shows that infinite lists can be regarded as the limit of an infinite sequence of partial lists

$$\perp$$
$$x0 : \perp$$
$$x0 : x1 : \perp$$
$$x0 : x1 : x2 : \perp$$
$$x0 : x1 : x2 : x3 : \perp$$
$$x0 : x1 : x2 : x3 : x4 : \perp$$

Proving properties for infinite lists

To show that a property P holds for every infinite list xs , we need to show

1. $P (\perp)$
2. For any partial list xs , $P (xs) \Rightarrow P (x : xs)$

If P is a chain complete property, then this suffices to show that the property also hold for any infinite list xs .

We can now show that

$$xs \mathrel{++} ys = xs$$

for any infinite list xs .

What about equality of infinite lists?

$$\text{iterate } f \ x = x : \text{iterate } f \ (f \ x)$$

$$\text{iterate}' f \ x = x : \text{map } f \ (\text{iterate } f \ x)$$

How do we approach the proof of:

$$\text{iterate } f \ x = \text{iterate}' f \ x$$

for any f, x ?

A more practical example

What about more advanced functions - like insertion sort...?

How to we describe that a list is *sorted*? I.e. a function

$$\text{sorted} :: \text{Ord } a \Rightarrow [a] \rightarrow \text{Bool}$$

which returns *True* if and only if a list is sorted. Any ideas?

- $\text{sorted } xs = \text{and } (\text{zipWith } (\leq) xs (\text{tail } xs))$
 - Turns out to be cumbersome to use in proofs
- Instead, we stick with the following definition

$$\text{sorted} :: \text{Ord } a \Rightarrow [a] \rightarrow \text{Bool}$$

$$\text{sorted } [] = \text{True}$$

$$\text{sorted } (x : xs) = \text{lower } x \ xs \wedge \text{sorted } xs$$

$$\text{lower} :: \text{Ord } a \rightarrow a \rightarrow [a] \rightarrow \text{Bool}$$

$$\text{lower } n \ [] = \text{True}$$

$$\text{lower } n \ (x : xs) = n \leq x \wedge \text{lower } n \ xs$$

The structure of the proof

We want to prove

$$\textit{sorted} (\textit{isort} \textit{xs})$$

To help us, we proceed in 4 different induction proofs:

1. $n \leq m \Rightarrow (\textit{lower } m \textit{ xs} \Rightarrow \textit{lower } n \textit{ xs})$
2. $\textit{lower } y (\textit{insert } x \textit{ as}) = (y \leq x \wedge \textit{lower } y \textit{ as})$
3. $\textit{sorted } ys = \textit{sorted } (\textit{insert } x \textit{ ys})$
4. $\textit{sorted} (\textit{isort} \textit{xs})$

DEMO: Isabelle Proof Assistant

Show the Insertion sort proof

Demonstrate the Sledgehammer

Mention limitations (restricted to total functions)

Are we done?

We now know that

sorted (*isort* *xs*)

for any list *xs*. Is this property enough to know that *isort* is correct?

What if I come up with the function

isort _ = []

Are we done?

We need to specify the fact that the sorted list is a permutation of the original list, i.e. a predicate

$$\text{perm} :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$$

should be defined, such that $\text{perm}\ xs\ ys = True$ iff ys is a permutation of xs .

$$\text{perm}\ []\ [] = True$$

$$\text{perm}\ []\ _ = False$$

$$\text{perm}\ (x : xs)\ ys = \text{elem}\ x\ ys \wedge \text{perm}\ xs\ (\text{delete}\ x\ ys)$$

There is another way...

So, induction proofs takes a lot of work.

But we can still get great testing only by write down the *properties* of our program, i.e. predicates

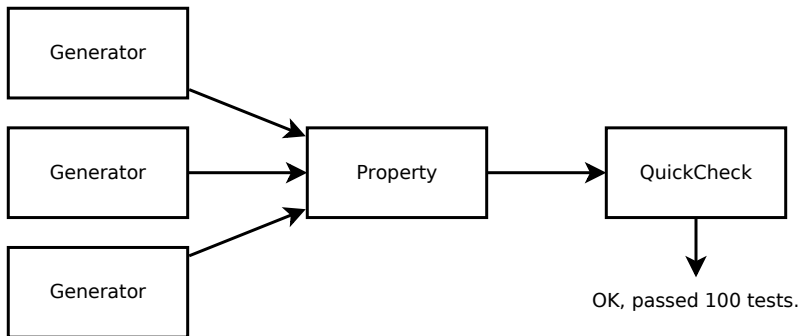
- $sorted :: Ord\ a \Rightarrow [a] \rightarrow Bool$
- $perm :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$

QuickCheck, an automated testing library/tool for Haskell

Features:

- Describe properties as Haskell programs using an embedded domain-specific language (EDSL).
- Automatic datatype-driven random test case generation.
- Extensible, e.g. test case generators can be adapted.

Overview



History

- Developed in 2000 by Koen Claessen and John Hughes.
- Copied to other programming languages: Common Lisp, Scheme, Erlang, Python, Ruby, SML, Clean, Java, Scala, F#
- Erlang version is sold by a company, QuviQ, founded by the authors of QuickCheck.

```

{-# LANGUAGE TemplateHaskell #-}

import Data.List
import Test.QuickCheck
import Test.QuickCheck.All (quickCheckAll)

    -- (... definitions of isort etc ...)

prop_lowerle n m xs    = n ≤ m ==> (lower m xs ==> lower n xs)
prop_lowerinsert x y as = lower y (insert x as) ≡ (y ≤ x ∧ lower y as)
prop_sortedinsert x ys  = sorted ys ≡ sorted (insert x ys)
prop_sortedisort xs     = sorted (isort xs)
prop_permisort xs       = perm xs (isort xs)

return []
main = $quickCheckAll

```


DEMO: QuickCheck for PolyList project

Summary

QuickCheck is a great tool:

- A domain-specific language for writing properties.
- Test data is generated automatically and randomly.
- Another domain-specific language to write custom generators.
- You should use it.

However, keep in mind that writing good tests still requires training, and that tests can have bugs, too.