Lecture 6: Proving and Testing Program Properties

Søren Haagerup

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

October 10, 2017

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 $diff :: Poly\ a \to Poly\ a$ (differentiate a polynomial) we could write the unit tests
     - $diff\ [\,] = [\,]$
     - $diff\ [1] \equiv [\,]$
     - $diff\ [1,2,3] \equiv [2,6]$
     - ...

Purely functional programming is *great* for testing!

- Every function is a "mathematical function" (no side-effects)
  - Testing is just a matter of constructing some input to a function, and seeing if the output is as expected
- In non-pure languages, the behaviour of methods depend both on input, but also on *external state* of the system.
  - You generally need "set up" and "tear down" methods, to build up the context a test should run within.

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:
        - $\lambda p \rightarrow deg\ (diff\ p) \equiv deg\ p - 1$
        - $\lambda p \rightarrow (diff \circ int)\ p \equiv p$
        - $\lambda p\ q \rightarrow diff\ (p\ \text{`}compose\text{`}\ q) \equiv ((diff\ p)\ \text{`}compose\text{`}\ q)\ \text{`}mul\text{`}\ (diff\ q)$
        - ...
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.

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*

Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and Reiner Hähnle[4]

[1] CWI, Amsterdam, The Netherlands
[2] SDL, Amsterdam, The Netherlands
[3] Leiden University, The Netherlands
[4] Technische Universität Darmstadt, Germany

Abstract. We investigate the correctness of TimSort, which is the main sorting algorithm provided by the Java standard library. The goal is functional verification with mechanical proofs. During our verification attempt we discovered a bug which causes the implementation to crash. We characterize the conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise the performance. We formally specify the new version and mechanically verify the absence of this bug with KeY, a state-of-the-art verification tool for Java.

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

$$
\begin{aligned}
&(\wedge) :: \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool} \\
&\mathit{True} \wedge x \ = x \\
&\mathit{False} \wedge x = \mathit{False}
\end{aligned}
$$

How do we prove the following?

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

Proving properties involving Algebraic Data Types

We saw that properties involving

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

are proved by splitting up in cases, corresponding to each of the definitons of the function.
What about properties involving recursive data types?

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ (\mathit{List}\ a)$$

These are also proved by cases - namely the base case and inductive case (we will use proof by *induction*).

First example

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

$$map\ f\ [\,] = [\,]$$
$$map\ f\ (x : xs) = f\ x : map\ f\ xs$$

It should hold that

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

Let's prove the first law:

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

  $$map\ id\ [\,] = [\,]$$

- **Inductive case:** Assume that $map\ id\ xs = xs$.
  Then we want to show

  $$map\ id\ (x : xs) = (x : xs)$$

The second law

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

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

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

The second law

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

Induction step:

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

Another example: *reverse*

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] +\!\!+ ys \qquad\quad = ys \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

$$
\begin{aligned}
&rev :: [a] \to [a] \\
&rev\ [\,] \qquad\quad = [\,] \\
&rev\ (x : xs) = rev\ xs +\!\!+ [x]
\end{aligned}
$$

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

$$
\begin{aligned}
&(xs +\!\!+ ys) +\!\!+ zs \equiv xs +\!\!+ (ys +\!\!+ zs) \\
&xs +\!\!+ [\,] \equiv xs \\
&rev\ (xs +\!\!+ ys) = rev\ ys +\!\!+ rev\ xs \\
&rev\ (rev\ xs) = xs
\end{aligned}
$$

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

$$
\begin{aligned}
(\oplus) &= (\lambda x \; rxs \rightarrow rxs +\!\!+ [x]) \\
(\otimes) &= (\mathit{flip}\ (:)) \\
e &= [\,]
\end{aligned}
$$

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

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

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

Showing the theorem itself...

Recall

$$\begin{aligned} \mathit{foldr}\ f\ z\ [\,] &= z \\ \mathit{foldr}\ f\ z\ (x : xs) &= f\ x\ (\mathit{foldr}\ f\ z\ xs) \end{aligned}$$

$$\begin{aligned} \mathit{foldl}\ f\ z\ [\,] &= z \\ \mathit{foldl}\ f\ z\ (x : xs) &= \mathit{foldl}\ f\ (f\ z\ x)\ xs \end{aligned}$$

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

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

for all finite lists xs.

We need an auxillary lemma:

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

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

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

for all finite lists xs.

Monoids

A type t is a Monoid, if there exists an operation $(\oplus) :: t \to t \to t$ 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

$$\textbf{instance}\ \mathit{Monoid}\ [a]\ \textbf{where}$$
$$\quad \mathit{mempty} = [\,]$$
$$\quad \mathit{mappend} = (+\!\!+)$$

$$[1,2,3]\ \text{`}\mathit{mappend}\text{`}\ [4,5,6] = [1,2,3,4,5,6]$$

$$\textbf{instance}\ (\mathit{Monoid}\ a, \mathit{Monoid}\ b) \Rightarrow \mathit{Monoid}\ (a,b)\ \textbf{where}$$
$$\quad \mathit{mempty} = (\mathit{mempty}, \mathit{mempty})$$
$$\quad (a1,b1)\ \text{`}\mathit{mappend}\text{`}\ (a2,b2) =$$
$$\qquad (a1\ \text{`}\mathit{mappend}\text{`}\ a2, b1\ \text{`}\mathit{mappend}\text{`}\ b2)$$

$$([1,3],\texttt{"le"})\ \text{`}\mathit{mappend}\text{`}\ ([3,7],\texttt{"et"}) = ([1,3,3,7],\texttt{"leet"})$$

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. $P\ (\bot)$
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

$$xs \mathbin{+\!\!+} ys = xs$$

for any infinite list xs.

Proving properties for both finite and infinite lists

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

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

Where do we run into trouble when trying to prove

$$rev\ (rev\ xs) = xs$$

for infinite lists also?

What about equality of infinite lists?

$$
\begin{aligned}
\mathit{iterate}\, f\ x &= x : \mathit{iterate}\, f\ (f\ x) \\
\mathit{iterate}'\, f\ x &= x : \mathit{map}\, f\ (\mathit{iterate}\, f\ x)
\end{aligned}
$$

How do we approach the proof of:

$$\mathit{iterate}\, f\ x = \mathit{iterate}'\, f\ x$$

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

$$sorted :: Ord\ a \Rightarrow [a] \rightarrow Bool$$

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

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

$$\begin{aligned}
&sorted :: Ord\ a \Rightarrow [a] \rightarrow Bool \\
&sorted\ [\,] \qquad\qquad = True \\
&sorted\ (x : xs) \quad = lower\ x\ xs \land sorted\ xs
\end{aligned}$$

$$\begin{aligned}
&lower :: Ord\ a \rightarrow a \rightarrow [a] \rightarrow Bool \\
&lower\ n\ [\,] \qquad\quad = True \\
&lower\ n\ (x : xs) = n \leqslant x \land lower\ n\ xs
\end{aligned}$$

The structure of the proof

We want to prove

$$sorted\ (isort\ xs)$$

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

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

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

$$perm :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$$

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

$$\begin{aligned}
&perm\ []\ [] &&= True \\
&perm\ []\ _ &&= False \\
&perm\ (x : xs)\ ys &&= elem\ x\ ys \wedge perm\ xs\ (delete\ x\ ys)
\end{aligned}$$

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

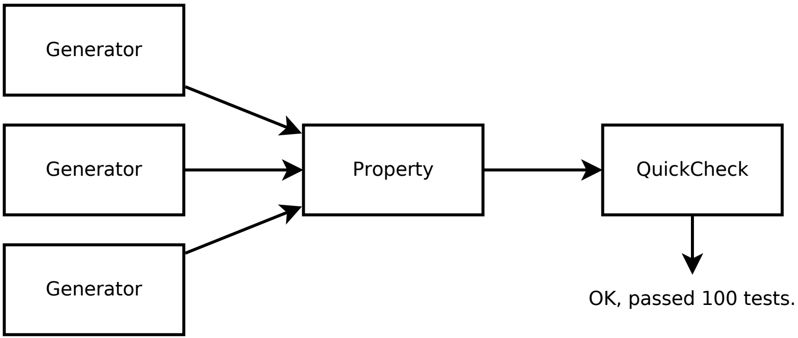
- $sorted :: Ord\ a \Rightarrow [a] \rightarrow Bool$
- $perm :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$

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


Generator
Generator
Generator
Property
QuickCheck
OK, passed 100 tests.

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.

```haskell
{-# 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.