Abstract Data Types and Automated Property-based Testing

George Giorgidze

Universität Tübingen

Advanced Functional Programming (Lecture 8)
(Some slides use material developed by Andres Löh)
2011 NOV 28, Tübingen, Germany

The AFP course so far

- Pure functional programming
- Typed programs with type inference
- Polymorphism
- Overloading with type classes
- Functions as first-class values
- Higher-order combinators
- Abstractions (e.g., *Functor* and *Applicative*)
- Lazy evaluation

The rest of the AFP course

- Abstract data types
- Automated property-based testing
- Monoids and monads
- Combining monads
- Functional programming for parallelism and concurrency
- Developing purely functional data structures
- Embedded Domain-specific Languages (EDSLs)
- Generic programming
- Topics for further research/professional development (more types, type-level programming, functional programming in industry and academia, etc.)

Abstract Data Types

- A data type
- Functions operating on the data type
- The data type representation is not exposed

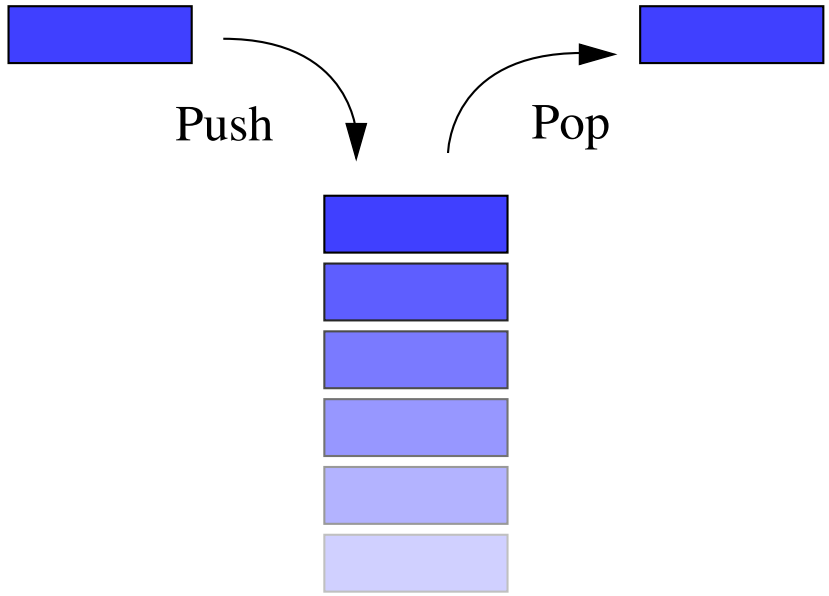
The Haskell Way

- Define the data type.
- Define the functions operating on the data type.
- Use the module system to hide the data type representation.

Live Hacking

data *Stack a*

$$
\begin{aligned}
&\mathit{empty} && :: \mathit{Stack}\ a \\
&\mathit{isEmpty} && :: \mathit{Stack}\ a \to \mathit{Bool} \\
&\mathit{push} && :: a \to \mathit{Stack}\ a \to \mathit{Stack}\ a \\
&\mathit{top} && :: \mathit{Stack}\ a \to a \\
&\mathit{pop} && :: \mathit{Stack}\ a \to (a, \mathit{Stack}\ a)
\end{aligned}
$$

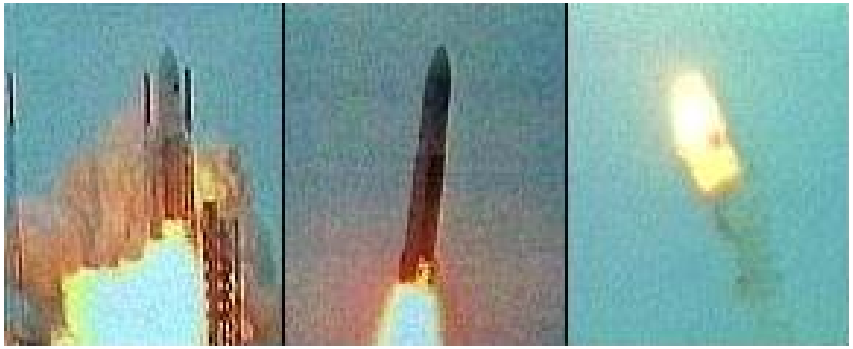


Correctness

- When is a program correct?
- What is a specification?
- How to establish a relation between the specification and the implementation?
- What about bugs in the specification?

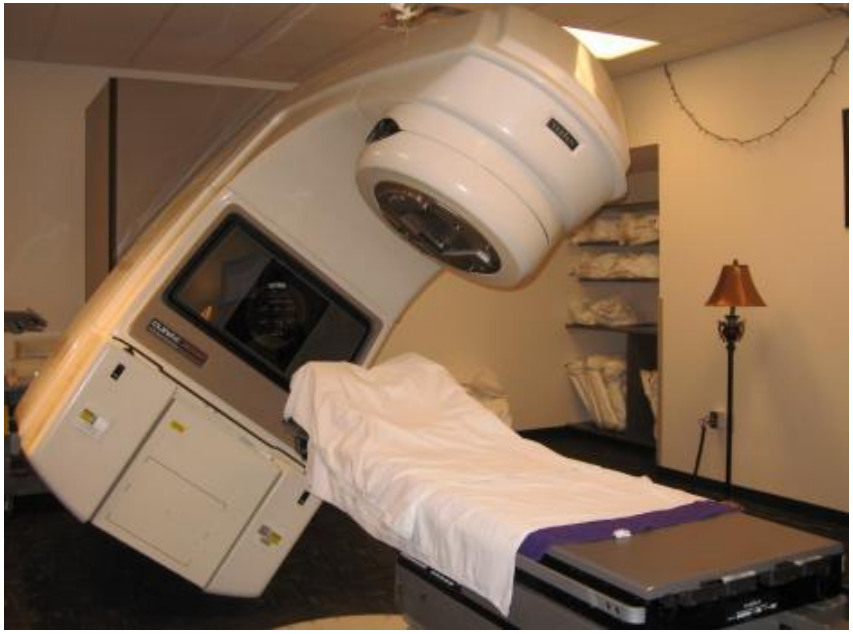
Why Correctness?

- Ariane 5 disaster
- Caused by an integer overflow bug
- Embarrassment for European Space Agency costing 370,000,000 USD



Why Correctness?

- The Therac-25 radiation therapy machine produced by Atomic Energy of Canada Limited (AECL)
- Involved in at least six accidents with massive (x100) overdoses of radiation
- Three patients died with radiation overdoses.
- Caused by a concurrency bug
- The bug was only triggered when the machine users got accustomed to it and started operating it quickly.



Establishing Correctness

- Equational reasoning
- Expressive type systems (e.g, Agda)
- Proof assistants (e.g., Coq and Isabelle)

- Widely used in the high assurance software/hardware industry
- Subject of active academic research

Testing

- Gain confidence in the correctness of your program.
- Show that common cases work correctly.
- Show that corner cases work correctly.
- Testing cannot (generally) prove the absence of bugs.

QuickCheck

- QuickCheck is a Haskell library developed by Koen Claessen and John Hughes.
- Library for defining properties.
- Automatic datatype-driven generation of random test data.
- Extensible by the user.
- Tries to shrink failing test cases.

Using QuickCheck

To use QuickCheck in your program:

$$\textbf{import } \mathit{Test.QuickCheck}$$

The simplest interface is to use

$$\mathit{quickCheck} :: \mathit{Testable\ prop} \Rightarrow \mathit{prop} \rightarrow \mathit{IO}\ ()$$

$$\textbf{class } \mathit{Testable\ prop} \textbf{ where}$$

$$\textbf{instance } \mathit{Testable\ Bool}$$

$$\textbf{instance } (\mathit{Arbitrary\ a}, \mathit{Show\ a}, \mathit{Testable\ prop}) \Rightarrow \mathit{Testable}\ (a \rightarrow \mathit{prop})$$

Quicksort

$$
\begin{aligned}
qsort\ \ & :: (Ord\ a) \Rightarrow [a] \rightarrow [a] \\
qsort\ [\,] & = [\,] \\
qsort\ (h : xs) & = qsort\ [x \mid x \leftarrow xs, x < h] \mathbin{+\!\!+} \\
& \quad\ [h] \mathbin{+\!\!+} \\
& \quad\ qsort\ [x \mid x \leftarrow xs, x \geqslant h]
\end{aligned}
$$

Nullary properties

instance *Testable Bool*

sortAscending :: *Bool*
sortAscending = *qsort* $[2,1] \equiv [1,2]$

sortDescending :: *Bool*
sortDescending = *qsort* $[2,1] \equiv [2,1]$

Running QuickCheck:

```
>>> quickCheck sortAscending
+++ OK, passed 100 tests.

>>> quickCheck sortDescending
*** Failed! Falsifiable (after 1 test):
```

Nullary properties

- Nullary properties are static properties.
- QuickCheck can be used for unit testing.
- By default, QuickCheck tests 100 times (which is wasteful for static properties, but configurable).

Functional properties

$$\textbf{instance}\ (\mathit{Arbitrary}\ a, \mathit{Show}\ a, \mathit{Testable}\ \mathit{prop}) \Rightarrow \mathit{Testable}\ (a \to \mathit{prop})$$

$$\mathit{preservesLength} :: ([\mathit{Int}] \to [\mathit{Int}]) \to [\mathit{Int}] \to \mathit{Bool}$$
$$\mathit{preservesLength}\ f\ xs = \mathit{length}\ (f\ xs) \equiv \mathit{length}\ xs$$

```
>>> quickCheck (preservesLength qsort)
+++ OK, passed 100 tests.
```

Read parameterized properties as universally quantified.
QuickCheck automatically generates lists of integers.

How to fully specify sorting

Property 1

A sorted list should be ordered:

$$
\begin{aligned}
&\mathit{sortOrders} :: [\mathit{Int}] \to \mathit{Bool} \\
&\mathit{sortOrders}\ \mathit{xs} = \mathit{ordered}\ (\mathit{qsort}\ \mathit{xs}) \\
&\mathit{ordered} :: \mathit{Ord}\ a \Rightarrow [a] \to \mathit{Bool} \\
&\mathit{ordered}\ [\,] \qquad\quad = \mathit{True} \\
&\mathit{ordered}\ [x] \qquad\quad = \mathit{True} \\
&\mathit{ordered}\ (x : y : \mathit{ys}) = x \leqslant y \wedge \mathit{ordered}\ (y : \mathit{ys})
\end{aligned}
$$

How to fully specify sorting

Property 2

A sorted list should have the same elements as the original list:

$$sortPreservesElements :: [Int] \to Bool$$
$$sortPreservesElements\ xs = sameElements\ xs\ (qsort\ xs)$$

$$sameElements :: Eq\ a \Rightarrow [a] \to [a] \to Bool$$
$$sameElements\ xs\ ys = null\ (xs \setminus\setminus ys) \wedge null\ (ys \setminus\setminus xs)$$

Implications

The function insert preserves an ordered list:

$$sortHeadMinimum :: [Int] \rightarrow Bool$$
$$sortHeadMinimum\ xs = head\ (qsort\ cs) \equiv minimum\ xs$$

```
>>> quickCheck sortHeadMinimum
*** Failed! Exception: 'Prelude.head: empty list' (after 1
[]
```

Implications

$$
\begin{aligned}
&\mathit{sortHeadMinimum} :: [\mathit{Int}] \to \mathit{Property} \\
&\mathit{sortHeadMinimum}\ \mathit{xs} = \\
&\quad \neg\ (\mathit{null}\ \mathit{xs}) ==> \mathit{head}\ (\mathit{qsort}\ \mathit{xs}) \equiv \mathit{minimum}\ \mathit{xs}
\end{aligned}
$$

```
>>> quickCheck sortHeadMinimum
+++ OK, passed 100 tests.
```

Configuring QuickCheck

$$
\begin{aligned}
&\mathit{quickCheck} &&:: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{prop} \to \mathit{IO}\ () \\
&\mathit{quickCheckWith} &&:: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{Args} \to \mathit{prop} \to \mathit{IO}\ () \\
&\mathit{quickCheckResult} &&:: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{prop} \to \mathit{IO}\ \mathit{Result} \\
&\mathit{verboseCheck} &&:: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{prop} \to \mathit{IO}\ () \\
&\mathit{verboseCheckWith} &&:: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{Args} \to \mathit{prop} \to \mathit{IO}\ () \\
&\mathit{verboseCheckResult} &&:: \mathit{Testable}\ \mathit{prop} \Rightarrow \mathit{prop} \to \mathit{IO}\ \mathit{Result}
\end{aligned}
$$

Live Hacking

- Quickchecking Quicksort
- Haskell Program Coverage (HPC)

Generators

- Generators belong to an abstract data type *Gen*. Think of *Gen* as a restricted version of *IO*. The only effect available to us is access to random numbers.
- We can define our own generators. We can define default generators for new datatypes by defining instances of class *Arbitrary*:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a → [a]
```

Combinators for generators

$$
\begin{aligned}
&\mathit{choose} && :: \mathit{Random}\ a \Rightarrow (a, a) \rightarrow \mathit{Gen}\ a \\
&\mathit{oneof} && :: [\mathit{Gen}\ a] \rightarrow \mathit{Gen}\ a \\
&\mathit{frequency} && :: [(\mathit{Int}, \mathit{Gen}\ a)] \rightarrow \mathit{Gen}\ a \\
&\mathit{elements} && :: [a] \rightarrow \mathit{Gen}\ a \\
&\mathit{sized} && :: (\mathit{Int} \rightarrow \mathit{Gen}\ a) \rightarrow \mathit{Gen}\ a
\end{aligned}
$$

Simple generators

```
instance Arbitrary Bool where
  arbitrary = choose (False, True)
instance (Arbitrary a, Arbitrary b) ⇒ Arbitrary (a, b) where
  arbitrary = do
                x ← arbitrary
                y ← arbitrary
                return (x, y)
data Dir = North | East | South | West
instance Arbitrary Dir where
  arbitrary = elements [North, East, South, West]
```

Generator for stacks

```
instance Arbitrary a ⇒ Arbitrary (Stack a) where
  arbitrary = do list ← arbitrary
                 return (Stack list)
```

Shrinking

The other method in *Arbitrary* is

$$shrink :: (Arbitrary\ a) \Rightarrow a \rightarrow [a]$$

- Maps each value to a number of structurally smaller values.
- Default definition returns [] and is always safe.
- When a failing test case is discovered, *shrink* is applied repeatedly until no smaller failing test case can be obtained.

Loose ends

- Haskell can deal with infinite values, and so can QuickCheck. However, properties must not inspect infinitely many values. For instance, we cannot compare two infinite values for equality and still expect tests to terminate. Solution: Only inspect finite parts.
- QuickCheck can generate functional values automatically, but this requires defining an instance of another class *CoArbitrary*.
- QuickCheck has facilities for testing properties that involve *IO*, but this is more difficult than testing pure properties.

Reading

- Real World Haskell - Chapter 11. Testing and quality assurance (also covers Haskell Program Coverage)
- Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00). ACM, New York, NY, USA, 268-279.

Next lecture

- Monoid and Monad
- aka, an algebraic structure with a single associative binary operation and an identity element, and an (endo-)functor, together with two natural transformations.
- aka, putting things together and a programmable semicolon