Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Advanced Functional Programming

2010-2011, periode 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

November 19, 2011

2. Correctness and Testing

What is testing about?

- 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.

Correctness

- When is a program correct?

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?

This lecture

- Equational reasoning with Haskell programs
- QuickCheck, an automated testing library/tool for Haskell

Goals

- Understand how to prove simple properties using equational reasoning.
- Understand how to define QuickCheck properties and how to use QuickCheck.
- Understand how QuickCheck works and how to make QuickCheck usable for your own larger programs.

2.1 Equational reasoning

Referential transparency

- "Equals can be substituted for equals"
- In other words: if an expression has a value in a context, we can replace it with any other expression that has the same value in the context without affecting the meaning of the program.
- When we deal with infinite structures: two things are equivalent if we cannot find out about their difference:

Referential transparency

- "Equals can be substituted for equals"
- In other words: if an expression has a value in a context, we can replace it with any other expression that has the same value in the context without affecting the meaning of the program.
- When we deal with infinite structures: two things are equivalent if we cannot find out about their difference:

$$\begin{aligned} \mathsf{ones} &= 1 : \mathsf{ones} \\ \mathsf{ones}' &= 1 : 1 : \mathsf{ones}' \end{aligned}$$

Universiteit Utrecht
[Faculty of **Science**
Information and Computing Sciences]

Referential transparency (contd.)

SML is (like most languages) not referentially transparent:

```
let val x   = ref 0
    fun f n = (x := !x + n; !x)
in  f 1 + f 2
end
```

The expression evaluates to 4.

Referential transparency (contd.)

SML is (like most languages) not referentially transparent:

```
let val x   = ref 0
    fun f n = (x := !x + n; !x)
in  f 1 + f 2
end
```

The expression evaluates to 4. The value of f 1 is 1. But

```
let val x   = ref 0
    fun f n = (x := !x + n; !x)
in  1 + f 2
end
```

evaluates to 3.

Referential transparency (contd.)

Also

$$\begin{array}{l} \textbf{let val } \mathsf{x} \quad = \textbf{ref } 0 \\ \quad \textbf{fun } \mathsf{f\ n} = (\mathsf{x} := !\mathsf{x} + \mathsf{n}; !\mathsf{x}) \\ \textbf{in } \ \mathsf{f}\ 1 + \mathsf{f}\ 1 \end{array}$$

cannot be replaced by

$$\begin{array}{l} \textbf{let val } \mathsf{x} \quad = \textbf{ref } 0 \\ \quad \textbf{fun } \mathsf{f\ n} = (\mathsf{x} := !\mathsf{x} + \mathsf{n}; !\mathsf{x}) \\ \quad \textbf{val } \mathsf{r} \quad = \mathsf{f}\ 1 \\ \textbf{in } \ \mathsf{r} + \mathsf{r} \end{array}$$

Referential transparency in Haskell

- Haskell is referentially transparent.
- The SML example breaks down because Haskell has no untracked side-effects.

```
do
  x ← newIORef 0
  let f n = do modifyIORef x (+n); readIORef x
  r ← f 1
  s ← f 2
  return (r + s)
```

The type of f is Int → IO Int, not Int → Int as in SML.

Referential transparency in Haskell (contd.)

- **Because of referential transparency**, the definitions of functions give us rules for reasoning about Haskell programs.
- Properties regarding datatypes can be proved using induction:

> **data** $[a] = [\,] \mid a : [a]$

To prove $\forall (xs :: [a]).P\ xs$, we prove

- $P\ [\,]$
- $\forall (x :: a)\ (xs :: [a]).P\ xs \rightarrow P\ (x : xs)$

Equational reasoning example

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

isort :: Ord a ⇒ [a] → [a]
isort []       = []
isort (x : xs) = insert x (isort xs)
```

```
insert :: Ord a ⇒ a → [a] → [a]
insert x []       = [x]
insert x (y : ys)
  | x ⩽ y         = x : y : ys
  | otherwise     = y : insert x ys
```

Equational reasoning example

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

isort :: Ord a ⇒ [a] → [a]
isort []       = []
isort (x : xs) = insert x (isort xs)
```

```
insert :: Ord a ⇒ a → [a] → [a]
insert x []       = [x]
insert x (y : ys)
  | x ⩽ y         = x : y : ys
  | otherwise     = y : insert x ys
```

Theorem (Sorting preserves length)

$\forall(xs :: [a]).\mathrm{length}\ (\mathrm{isort}\ xs) \equiv \mathrm{length}\ xs$

Equational reasoning example

$$\begin{aligned}
&\mathsf{length} :: [\mathsf{a}] \to \mathsf{Int}\\
&\mathsf{length}\ [\,] \qquad\quad = 0\\
&\mathsf{length}\ (\mathsf{x} : \mathsf{xs}) = 1 + \mathsf{length}\ \mathsf{xs}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{isort} :: \mathsf{Ord}\ \mathsf{a} \Rightarrow [\mathsf{a}] \to [\mathsf{a}]\\
&\mathsf{isort}\ [\,] \qquad\quad = [\,]\\
&\mathsf{isort}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{insert}\ \mathsf{x}\ (\mathsf{isort}\ \mathsf{xs})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{insert} :: \mathsf{Ord}\ \mathsf{a} \Rightarrow \mathsf{a} \to [\mathsf{a}] \to [\mathsf{a}]\\
&\mathsf{insert}\ \mathsf{x}\ [\,] \qquad\quad = [\mathsf{x}]\\
&\mathsf{insert}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys})\\
&\qquad \mid \mathsf{x} \leqslant \mathsf{y} \qquad\quad = \mathsf{x} : \mathsf{y} : \mathsf{ys}\\
&\qquad \mid \mathsf{otherwise} = \mathsf{y} : \mathsf{insert}\ \mathsf{x}\ \mathsf{ys}
\end{aligned}$$

Theorem (Sorting preserves length)

$$\forall(\mathsf{xs} :: [\mathsf{a}]).\mathsf{length}\ (\mathsf{isort}\ \mathsf{xs}) \equiv \mathsf{length}\ \mathsf{xs}$$

Lemma

$$\forall(\mathsf{x} :: \mathsf{a})\ (\mathsf{ys} :: [\mathsf{a}]).\mathsf{length}\ (\mathsf{insert}\ \mathsf{x}\ \mathsf{ys}) \equiv 1 + \mathsf{length}\ \mathsf{ys}$$

Proof of the Lemma

Lemma

$\forall (\mathsf{x} :: \mathsf{a})\ (\mathsf{ys} :: [\mathsf{a}]). \mathsf{length}\ (\mathsf{insert}\ \mathsf{x}\ \mathsf{ys}) \equiv 1 + \mathsf{length}\ \mathsf{ys}$

Proof by induction on the list.

Case $[\,]$:

$$
\begin{aligned}
& \mathsf{length}\ (\mathsf{insert}\ \mathsf{x}\ [\,]) \\
\equiv \quad & \{\ \text{Definition of insert}\ \} \\
& \mathsf{length}\ [\mathsf{x}] \\
\equiv \quad & \{\ \text{Definition of length}\ \} \\
& 1 + \mathsf{length}\ [\,]
\end{aligned}
$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Proof of the Lemma (contd.)

Lemma

$\forall (x :: a)\ (ys :: [a]).\mathsf{length}\ (\mathsf{insert}\ x\ ys) \equiv 1 + \mathsf{length}\ ys$

Case $y : ys$, case $x \leqslant y$:

$$
\begin{array}{ll}
 & \mathsf{length}\ (\mathsf{insert}\ x\ (y : ys)) \\
\equiv & \quad \{\ \text{Definition of insert}\ \} \\
 & \mathsf{length}\ (x : y : ys) \\
\equiv & \quad \{\ \text{Definition of length}\ \} \\
 & 1 + \mathsf{length}\ (y : ys)
\end{array}
$$

Proof of the Lemma (contd.)

Lemma

$\forall (x :: a)\ (ys :: [a]).\text{length}\ (\text{insert}\ x\ ys) \equiv 1 + \text{length}\ ys$

Case y : ys, case $x > y$:

$$
\begin{array}{ll}
 & \text{length}\ (\text{insert}\ x\ (y : ys)) \\
\equiv & \quad \{\ \text{Definition of insert}\ \} \\
 & \text{length}\ (y : \text{insert}\ x\ ys) \\
\equiv & \quad \{\ \text{Definition of length}\ \} \\
 & 1 + \text{length}\ (\text{insert}\ x\ ys) \\
\equiv & \quad \{\ \text{Induction hypothesis}\ \} \\
 & 1 + (1 + \text{length}\ ys) \\
\equiv & \quad \{\ \text{Definition of length}\ \} \\
 & 1 + \text{length}\ (y : ys)
\end{array}
$$

Proof of the Theorem

Theorem

$\forall (xs :: [a]).\text{length}\ (\text{isort}\ xs) \equiv \text{length}\ xs$

Proof by induction on the list.

Case $[\,]$:

$$
\begin{array}{ll}
 & \text{length}\ (\text{isort}\ [\,]) \\
\equiv & \quad \{\ \text{Definition of isort}\ \} \\
 & \text{length}\ [\,]
\end{array}
$$

Proof of the Theorem (contd.)

Theorem

$\forall (xs :: [a]).\text{length (isort xs)} \equiv \text{length xs}$

Case x : xs:

$$
\begin{aligned}
& \text{length (isort (x : xs))} \\
\equiv \quad & \{ \text{ Definition of isort } \} \\
& \text{length (insert x (isort xs))} \\
\equiv \quad & \{ \text{ Lemma } \} \\
& 1 + \text{length (isort xs)} \\
\equiv \quad & \{ \text{ Induction hypothesis } \} \\
& 1 + \text{length xs} \\
\equiv \quad & \{ \text{ Definition of length } \} \\
& \text{length (x : xs)}
\end{aligned}
$$

Equational reasoning summary

- Equational reasoning can be an elegant way to prove properties of a program.
- Equational reasoning can be used to establish a relation between an "obivously correct" Haskell program (a specification) and an efficient Haskell program.
- Equational reasoning is usually quite lengthy.
- Careful with special cases (laziness):
  - undefined values;
  - infinite values
- It is infeasible to prove properties about every Haskell program using equational reasoning.

Other proof methods

- Type systems.
- Proof assistants.

2.2 QuickCheck

QuickCheck

- QuickCheck is a Haskell library developed by Koen Claessen and John Hughes in 2000.
- An embedded domain-specific language (EDSL) for defining properties.
- Automatic datatype-driven generation of random test data.
- Extensible by the user.
- Shrinks failing test cases.

Current sitation

- 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.

Example: Sorting

An attempt at insertion sort in Haskell:

$$
\begin{array}{l}
\mathsf{sort} :: [\mathsf{Int}] \to [\mathsf{Int}] \\
\mathsf{sort}\ [\,] \qquad\quad = [\,] \\
\mathsf{sort}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{insert}\ \mathsf{x}\ \mathsf{xs} \\[4pt]
\mathsf{insert} :: \mathsf{Int} \to [\mathsf{Int}] \to [\mathsf{Int}] \\
\mathsf{insert}\ \mathsf{x}\ [\,] \qquad\qquad\qquad\qquad\quad = [\mathsf{x}] \\
\mathsf{insert}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys}) \mid \mathsf{x} \leqslant \mathsf{y} \qquad\; = \mathsf{x} : \mathsf{ys} \\
\qquad\qquad\qquad\qquad\;\; \mid \mathsf{otherwise} = \mathsf{y} : \mathsf{insert}\ \mathsf{x}\ \mathsf{ys}
\end{array}
$$

How to specify sorting?

A good specification is

- as precise as necessary,
- no more precise than necessary.

How to specify sorting?

A good specification is

- as precise as necessary,
- no more precise than necessary.

If we want to specify sorting, we should give a specification that distinguishes sorting from all other operations, but does not force us to use a particular sorting algorithm.

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

A first approximation

Certainly, sorting a list should not change its length.

```
sortPreservesLength :: [Int] → Bool
sortPreservesLength xs = length xs == length (sort xs)
```

A first approximation

Certainly, sorting a list should not change its length.

$$\text{sortPreservesLength} :: [\text{Int}] \to \text{Bool}$$
$$\text{sortPreservesLength xs} = \text{length xs} == \text{length (sort xs)}$$

We can test by invoking the function quickCheck:

Main⟩ quickCheck sortPreservesLength
Failed! Falsifiable, after 4 tests:
$[0, 3]$

Correcting the bug

$$
\begin{aligned}
&\mathsf{sort} :: [\mathsf{Int}] \to [\mathsf{Int}]\\
&\mathsf{sort}\ [\,] = [\,]\\
&\mathsf{sort}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{insert}\ \mathsf{x}\ \mathsf{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{insert} :: \mathsf{Int} \to [\mathsf{Int}] \to [\mathsf{Int}]\\
&\mathsf{insert}\ \mathsf{x}\ [\,] = [\mathsf{x}]\\
&\mathsf{insert}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys})\ \mid \mathsf{x} \leqslant \mathsf{y} = \mathsf{x} : \mathsf{ys}\\
&\qquad\qquad\qquad\quad\ \mid \mathsf{otherwise} = \mathsf{y} : \mathsf{insert}\ \mathsf{x}\ \mathsf{ys}
\end{aligned}
$$

Correcting the bug

$$
\begin{array}{l}
\mathsf{sort} :: [\mathsf{Int}] \rightarrow [\mathsf{Int}] \\
\mathsf{sort}\ [\,] \qquad\quad = [\,] \\
\mathsf{sort}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{insert}\ \mathsf{x}\ \mathsf{xs}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{insert} :: \mathsf{Int} \rightarrow [\mathsf{Int}] \rightarrow [\mathsf{Int}] & & \\
\mathsf{insert}\ \mathsf{x}\ [\,] & & = [\mathsf{x}] \\
\mathsf{insert}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys}) & \mid \mathsf{x} \leqslant \mathsf{y} & = \mathsf{x} : \boxed{\mathsf{y} :} \mathsf{ys} \\
 & \mid \mathsf{otherwise} & = \mathsf{y} : \mathsf{insert}\ \mathsf{x}\ \mathsf{ys}
\end{array}
$$

A new attempt

```
Main⟩ quickCheck sortPreservesLength
OK, passed 100 tests.
```

Looks better. But have we tested enough?

Properties are first-class objects

```
(f `preserves` p) x = p x == p (f x)
sortPreservesLength = sort `preserves` length
idPreservesLength   = id `preserves` length
```

```
Main⟩ quickCheck idPreservesLength
OK, passed 100 tests.
```

Clearly, the identity function does not sort the list.

When is a list sorted?

$$\begin{aligned}
&\mathsf{sorted} :: [\mathsf{Int}] \rightarrow \mathsf{Bool} \\
&\mathsf{sorted}\ [\,] \qquad\qquad = \mathsf{True} \\
&\mathsf{sorted}\ (\mathsf{x} : \mathsf{xs}) \quad = \ ?
\end{aligned}$$

When is a list sorted?

$$\begin{aligned}
&\text{sorted} :: [\text{Int}] \rightarrow \text{Bool} \\
&\text{sorted } [\,] \qquad\qquad = \text{True} \\
&\text{sorted } (x : \boxed{xs}) \quad = \; ?
\end{aligned}$$

When is a list sorted?

$$
\begin{aligned}
&\mathsf{sorted} :: [\mathsf{Int}] \to \mathsf{Bool}\\
&\mathsf{sorted}\ [\,] &&= \mathsf{True}\\
&\mathsf{sorted}\ (\mathsf{x} : \boxed{[\,]}) &&= \ ?\\
&\mathsf{sorted}\ (\mathsf{x} : \boxed{\mathsf{y} : \mathsf{ys}}) &&= \ ?
\end{aligned}
$$

When is a list sorted?

$$
\begin{aligned}
&\mathsf{sorted} :: [\mathsf{Int}] \to \mathsf{Bool}\\
&\mathsf{sorted}\ [\,] &&= \mathsf{True}\\
&\mathsf{sorted}\ (\mathsf{x} : [\,]) &&= \mathsf{True}\\
&\mathsf{sorted}\ (\mathsf{x} : \mathsf{y} : \mathsf{ys}) &&= \ ?
\end{aligned}
$$

When is a list sorted?

$$
\begin{aligned}
&\mathsf{sorted} :: [\mathsf{Int}] \to \mathsf{Bool}\\
&\mathsf{sorted}\ [\,] &&= \mathsf{True}\\
&\mathsf{sorted}\ (\mathsf{x} : [\,]) &&= \mathsf{True}\\
&\mathsf{sorted}\ (\mathsf{x} : \mathsf{y} : \mathsf{ys}) &&= \mathsf{x} < \mathsf{y} \wedge \mathsf{sorted}\ (\mathsf{y} : \mathsf{ys})
\end{aligned}
$$

Testing again

$$\text{sortEnsuresSorted} :: [\text{Int}] \to \text{Bool}$$
$$\text{sortEnsuresSorted}\ xs = \text{sorted}\ (\text{sort}\ xs)$$

Testing again

$$\text{sortEnsuresSorted} :: [\text{Int}] \rightarrow \text{Bool}$$
$$\text{sortEnsuresSorted xs} = \text{sorted (sort xs)}$$

Or:

$$(\text{f `ensures` p})\ \text{x} = \text{p (f x)}$$
$$\text{sortEnsuresSorted} = \text{sort `ensures` sorted}$$

Testing again

$$\text{sortEnsuresSorted} :: [\text{Int}] \rightarrow \text{Bool}$$
$$\text{sortEnsuresSorted xs} = \text{sorted (sort xs)}$$

Or:

$$(\text{f `ensures` p})\ \text{x} = \text{p (f x)}$$
$$\text{sortEnsuresSorted} = \text{sort `ensures` sorted}$$

Main⟩ quickCheck sortEnsuresSorted
Falsifiable, after 5 tests:
$[5, 0, -2]$
Main⟩ sort $[5, 0, -2]$
$[0, -2, 5]$

Correcting again

$$
\begin{aligned}
&\mathsf{sort} :: [\mathsf{Int}] \to [\mathsf{Int}]\\
&\mathsf{sort}\ [\,] \qquad\quad = [\,]\\
&\mathsf{sort}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{insert}\ \mathsf{x}\ \mathsf{xs}\\
&\mathsf{insert} :: \mathsf{Int} \to [\mathsf{Int}] \to [\mathsf{Int}]\\
&\mathsf{insert}\ \mathsf{x}\ [\,] \qquad\qquad\qquad\qquad\quad = [\mathsf{x}]\\
&\mathsf{insert}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys})\ \mid \mathsf{x} \leqslant \mathsf{y} \qquad\quad = \mathsf{x} : \mathsf{y} : \mathsf{ys}\\
&\qquad\qquad\qquad\qquad\ \ \mid \mathsf{otherwise} = \mathsf{y} : \mathsf{insert}\ \mathsf{x}\ \mathsf{ys}
\end{aligned}
$$

Correcting again

```
sort :: [Int] → [Int]
sort []       = []
sort (x : xs) = insert x (sort xs)

insert :: Int → [Int] → [Int]
insert x []                     = [x]
insert x (y : ys) | x ⩽ y       = x : y : ys
                  | otherwise   = y : insert x ys
```

Correcting again

$$
\begin{aligned}
&\mathsf{sort} :: [\mathsf{Int}] \to [\mathsf{Int}]\\
&\mathsf{sort}\ [\,] \qquad\quad = [\,]\\
&\mathsf{sort}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{insert}\ \mathsf{x}\ (\mathsf{sort}\ \mathsf{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{insert} :: \mathsf{Int} \to [\mathsf{Int}] \to [\mathsf{Int}]\\
&\mathsf{insert}\ \mathsf{x}\ [\,] \qquad\qquad\qquad\qquad = [\mathsf{x}]\\
&\mathsf{insert}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys}) \mid \mathsf{x} \leqslant \mathsf{y} \qquad\ \ = \mathsf{x} : \mathsf{y} : \mathsf{ys}\\
&\qquad\qquad\qquad\quad\ \ \mid \mathsf{otherwise} = \mathsf{y} : \mathsf{insert}\ \mathsf{x}\ \mathsf{ys}
\end{aligned}
$$

Main⟩ quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
$[4, 2, 2]$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Another bug?

> Main⟩ quickCheck sortEnsuresSorted
> Falsifiable, after 7 tests:
> $[4, 2, 2]$
> Main⟩ sort $[4, 2, 2]$
> $[2, 2, 4]$

But this is correct. So what went wrong?

Another bug?

Main⟩ quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
$[4,2,2]$
Main⟩ sort $[4,2,2]$
$[2,2,4]$

But this is correct. So what went wrong?

Main⟩ sorted $[2,2,4]$
False

Specifications can have bugs, too!

$$
\begin{aligned}
&\mathsf{sorted} :: [\mathsf{Int}] \to \mathsf{Bool}\\
&\mathsf{sorted}\ [\,] &&= \mathsf{True}\\
&\mathsf{sorted}\ (\mathsf{x} : [\,]) &&= \mathsf{True}\\
&\mathsf{sorted}\ (\mathsf{x} : \mathsf{y} : \mathsf{ys}) &&= \mathsf{x} < \mathsf{y} \wedge \mathsf{sorted}\ (\mathsf{y} : \mathsf{ys})
\end{aligned}
$$

Specifications can have bugs, too!

$$
\begin{aligned}
&\mathsf{sorted} :: [\mathsf{Int}] \to \mathsf{Bool} \\
&\mathsf{sorted}\ [\,] = \mathsf{True} \\
&\mathsf{sorted}\ (x : [\,]) = \mathsf{True} \\
&\mathsf{sorted}\ (x : y : ys) = x \boxed{\leqslant} y \wedge \mathsf{sorted}\ (y : ys)
\end{aligned}
$$

Are we done yet?

Is sorting specified completely by saying that

- sorting preserves the length of the input list,
- the resulting list is sorted?

No, not quite

```
evilNoSort :: [Int] → [Int]
evilNoSort xs = replicate (length xs) 0
```

This function fulfills both specifications, but still does not sort.

We need to make the relation between the input and output lists precise: both should contain the same elements – or one should be a permutation of the other.

Specifying sorting

```
f `permutes` xs    = f xs `elem` permutations xs
sortPermutes xs = sort `permutes` xs
```

Our sorting function now fulfills the specification.

Using QuickCheck

To use QuickCheck in your program:

```haskell
import Test.QuickCheck
```

The simplest interface is to use

```haskell
quickCheck :: Testable prop => prop -> IO ()
```

```haskell
class Testable prop where
  property :: prop -> Property
instance Testable Bool
instance (Arbitrary a, Show a, Testable prop) =>
           Testable (a -> prop)
```

Recap: Classes and instances

- Classes declare predicates on types.

  ```
  class Testable prop where
    property :: prop → Property
  ```

  Here, any type can either be Testable or not.

- If a predicate holds for a type, this implies that the class methods are supported by the type.
  For any type prop such that Testable prop, there is a method property :: prop → Property.
  Outside of a class declaration, Haskell denotes this type as

  ```
  property :: Testable prop ⇒ prop → Property
  ```

Recap: Classes and instances (contd.)

- Instances declare which types belong to a predicate.

  > **instance** Testable Bool
  > **instance** (Arbitrary a, Show a, Testable prop) $\Rightarrow$
  > Testable (a $\rightarrow$ prop)

  Booleans are in Testable.
  Functions, i.e., values of type a $\rightarrow$ prop, are in Testable if prop is Testable and a is in Arbitrary and in Show.
- Instance declarations have to provide implementations of the class methods (in this case, of property), as a proof that the predicate does indeed hold for the type.
- Other functions that use class methods inherit the class constraints:

  > quickCheck :: Testable prop $\Rightarrow$ prop $\rightarrow$ IO ()

Nullary properties

instance Testable Bool

sortAscending :: Bool
sortAscending $=$ sort $[2,1]$ == $[1,2]$

sortDescending :: Bool
sortDescending $=$ sort $[2,1]$ == $[2,1]$

Running QuickCheck:

```
Main⟩ quickCheck sortAscending
+++ OK, passed 100 tests.
Main⟩ quickCheck sortDescending
*** Failed! Falsifiable (after 1 test):
```

Nullary properties (contd.)

- 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

```
instance (Arbitrary a, Show a, Testable prop) ⇒
            Testable (a → prop)
```

```
sortPreservesLength :: ([Int] → [Int]) → [Int] → Bool
sortPreservesLength isort xs = length (isort xs) == length xs
```

```
Main⟩ quickCheck (sortPreservesLength isort)
+++ OK, passed 100 tests.
```

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

Another sorting function

```
import Data.Set
setSort = toList ∘ fromList
```

Another sorting function

import Data.Set

$\text{setSort} = \text{toList} \circ \text{fromList}$

Main⟩ quickCheck (sortPreservesLength setSort)
*** Failed! Falsifiable (after 6 tests and 2 shrinks):
$[1, 1]$

Another sorting function

```
import Data.Set
setSort = toList ∘ fromList
```

```
Main⟩ quickCheck (sortPreservesLength setSort)
*** Failed! Falsifiable (after 6 tests and 2 shrinks):
[1, 1]
```

- The function setSort eliminates duplicate elements, therefore a list with duplicate elements causes the test to fail.
- QuickCheck shows evidence of the failure, and tries to present minimal test cases that fail (shrinking).

How to fully specify sorting

Property 1

A sorted list should be ordered:

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

How to fully specify sorting (contd.)

Property 2

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

$$\mathsf{sortPreservesElements} :: [\mathsf{Int}] \to \mathsf{Bool}$$
$$\mathsf{sortPreservesElements}\ \mathsf{xs} = \mathsf{sameElements}\ \mathsf{xs}\ (\mathsf{sort}\ \mathsf{xs})$$

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

More information about test data

$\text{collect} :: (\text{Testable prop}, \text{Show a}) \Rightarrow \text{a} \rightarrow \text{prop} \rightarrow \text{Property}$

The function collect gathers statistics about test cases. This information is displayed when a test passes:

More information about test data

$$\text{collect} :: (\text{Testable prop}, \text{Show a}) \Rightarrow \text{a} \to \text{prop} \to \text{Property}$$

The function collect gathers statistics about test cases. This information is displayed when a test passes:

```
Main⟩ let p = sortPreservesLength isort
Main⟩ quickCheck (λxs → collect (null xs) (p xs))
+++ OK, passed 100 tests:
92% False
 8% True
```

More information about test data (contd.)

```
Main⟩ quickCheck (λxs → collect (length xs `div` 10) (p xs))
+++ OK, passed 100 tests:
31% 0
24% 1
16% 2
 9% 4
 9% 3
 4% 8
 4% 6
 2% 5
 1% 7
```

More information about test data (contd.)

In the extreme case, we can show the actual data that is tested:

Main⟩ quickCheck (λxs $\rightarrow$ collect xs (p xs))
+++ OK, passed 100 tests:
6% $[\,]$
1% $[9, 4, -6, 7]$
1% $[9, -1, 0, -22, 25, 32, 32, 0, 9, \ldots$
$\ldots$

Question

Why is it important to have access to the test data?

Implications

The function insert preserves an ordered list:

$$\begin{array}{l} \text{implies} :: \text{Bool} \to \text{Bool} \to \text{Bool} \\ \text{implies}\ x\ y = \text{not}\ x \lor y \end{array}$$

Problematic:

$$\begin{array}{l} \text{insertPreservesOrdered} :: \text{Int} \to [\text{Int}] \to \text{Bool} \\ \text{insertPreservesOrdered}\ x\ xs = \\ \quad \text{ordered}\ xs\ \text{`implies`}\ \text{ordered}\ (\text{insert}\ x\ xs) \end{array}$$

Implications (contd.)

```
Main⟩ quickCheck insertPreservesOrdered
+++ OK, passed 100 tests.
```

Implications (contd.)

```
Main⟩ quickCheck insertPreservesOrdered
+++ OK, passed 100 tests.
```

But:

```
Main⟩ let iPO = insertPreservesOrdered
Main⟩ quickCheck (λx xs → collect (ordered xs) (iPO x xs))
+++ OK, passed 100 tests.
88% False
12% True
```

Only 12 lists have really been tested!

Implications (contd.)

The solution is to use the QuickCheck implication operator:

$(\Longrightarrow) :: (\text{Testable prop}) \Rightarrow \text{Bool} \rightarrow \text{prop} \rightarrow \text{Property}$

instance Testable Property

The type Property allows to encode not only True or False, but also to reject the test case.

$\text{iPO} :: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Property}$
$\text{iPO } x \text{ xs} = \text{ordered xs} \Longrightarrow \text{ordered (insert } x \text{ xs)}$

Now we get:

```
Main⟩ quickCheck (λx xs → collect (ordered xs) (iPO x xs))
*** Gave up! Passed only 43 tests (100% True).
```

Configuring QuickCheck

```
quickCheckWith ::
  (Testable prop) ⇒ Int       -- maximum number tests
                  → Int       -- maximum number attempts
                  → Int       -- maximum size
                  → prop
                  → IO Bool

quickCheck p =
  do
    quickCheckWith 100 500 100 p
    return ()
```

- Increasing the number of attempts might work.
- Better solution: use a custom generator (discussed next).

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 using another domain-specific language. 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

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

Simple generators

```
instance Arbitrary Bool where
  arbitrary = elements [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 deriving Ord
instance Arbitrary Dir where
  arbitrary = choose (North, West)
```

Generating numbers

- A simple possibility:

  instance Arbitrary Int **where**
  $\quad$ arbitrary $=$ choose $(-20, 20)$

- Better:

  instance Arbitrary Int **where**
  $\quad$ arbitrary $=$ sized $(\lambda n \rightarrow$ choose $(-n, n))$

- QuickCheck automatically increases the size gradually, up to the configured maximum value.

Generating trees

A bad approach to generating more complex values is a frequency table:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary =
    frequency [(1, liftM   Leaf  arbitrary),
               (2, liftM2 Node arbitrary arbitrary)]
```

Here:

```
liftM  :: (a -> b)      -> Gen a -> Gen b
liftM2 :: (a -> b -> c) -> Gen a -> Gen b -> Gen c
```

Generating trees

A bad approach to generating more complex values is a frequency table:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
instance Arbitrary a ⇒ Arbitrary (Tree a) where
  arbitrary =
    frequency [(1, liftM   Leaf  arbitrary),
               (2, liftM2 Node arbitrary arbitrary)]
```

Here:

```
liftM  :: (a → b)      → Gen a → Gen b
liftM2 :: (a → b → c) → Gen a → Gen b → Gen c
```

Termination is unlikely!

Generating trees (contd.)

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbitraryTree

arbitraryTree :: Arbitrary a => Int -> Gen (Tree a)
arbitraryTree 0 = liftM Leaf arbitrary
arbitraryTree n = frequency [(1, liftM   Leaf  arbitrary),
                             (4, liftM2 Node t t)]
  where t = arbitraryTree (n `div` 2)
```

Why a non-zero probability for Leaf in the second case of arbitraryTree?

Shrinking

The other method in Arbitrary is

$$\text{shrink} :: (\text{Arbitrary a}) \Rightarrow \text{a} \to [\text{a}]$$

- 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.

Defining Arbitrary generically

- Both arbitrary and shrink are examples of datatype-generic functions – they can be defined for (almost) any Haskell datatype in a systematic way.
- Haskell does not provide any way to denote such an algorithm.
- Many extensions and tools do (cf. course on Generic Programming in block 4).

GHCi pitfall

All lists are ordered?

> Main⟩ quickCheck ordered
> +++ OK, passed 100 tests.

GHCi pitfall

All lists are ordered?

```
Main⟩ quickCheck ordered
+++ OK, passed 100 tests.
```

Use type signatures in GHCi to make sure a sensible type is used!

```
Main⟩ quickCheck (ordered :: [Int] → Bool)
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
[0, −1]
```

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.

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. Also, showing functional values is problematic.

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. Also, showing functional values is problematic.
- QuickCheck has facilities for testing properties that involve IO, but this is more difficult than testing pure properties.

Next lecture

- Today (maybe late): First set of weekly assignments.
- Tuesday: Parsing paper, since many projects depend on it (read at least first half)
- Thursday wc: Programming project discussions, work on exercises