Universiteit Utrecht

Advanced Functional Programming

2012-2013, periode 2

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.

Universiteit Utrecht

Correctness

- ▶ When is a program correct?

Universiteit Utrecht

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?

Universiteit Utrecht

This lecture

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

Universiteit Utrecht

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.

Universiteit Utrecht

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:

Universiteit Utrecht

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:

$$\text{ones } = 1: \quad \text{ones}$$
$$\text{ones}' = 1:1:\text{ones}'$$

Universiteit Utrecht

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:

```sml
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

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

evaluates to 3.

Universiteit Utrecht

Referential transparency (contd.)

Also

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

cannot be replaced by

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

Universiteit Utrecht

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.

Universiteit Utrecht

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

Universiteit Utrecht

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
```

Universiteit Utrecht

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)

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

Universiteit Utrecht

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)

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

Lemma

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

Universiteit Utrecht

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

Proof of the Lemma

Lemma

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

Proof by induction on the list.

Case $[\,]$:

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

Universiteit Utrecht

Proof of the Lemma (contd.)

Lemma

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

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

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

Universiteit Utrecht

Proof of the Lemma (contd.)

Lemma

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

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

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

Universiteit Utrecht

Proof of the Theorem

Theorem

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

Proof by induction on the list.

Case $[\,]$:

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

Universiteit Utrecht

Proof of the Theorem (contd.)

Theorem

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

Case $x : xs$:

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

Universiteit Utrecht

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 "obiously 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.

Universiteit Utrecht

Other proof methods

- Type systems.
- Proof assistants.

Universiteit Utrecht

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.

Universiteit Utrecht

Example: Sorting

An attempt at insertion sort in Haskell:

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

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

Universiteit Utrecht

How to specify sorting?

A good specification is

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

Universiteit Utrecht

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

A first approximation

Certainly, sorting a list should not change its length.

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

Universiteit Utrecht

A first approximation

Certainly, sorting a list should not change its length.

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

We can test by invoking the function quickCheck:

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

Universiteit Utrecht

Correcting the bug

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

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

Correcting the bug

$$\text{sort} :: [\text{Int}] \rightarrow [\text{Int}]$$
$$\text{sort } [\,] \quad\quad = [\,]$$
$$\text{sort } (x : xs) = \text{insert } x \text{ } xs$$

$$\text{insert} :: \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}]$$
$$\text{insert } x \text{ } [\,] \quad\quad\quad\quad\quad = [x]$$
$$\text{insert } x \text{ } (y : ys) \mid x \leqslant y \quad\quad = x : \boxed{y : \text{ys}}$$
$$\quad\quad\quad\quad\quad \mid \text{otherwise} = y : \text{insert } x \text{ ys}$$

Universiteit Utrecht

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?

```
sorted :: [Int] → Bool
sorted []        = True
sorted (x : xs)  =  ?
```

Universiteit Utrecht

When is a list sorted?

sorted :: [Int] → Bool
sorted []   = True
sorted (x : xs)   =   ?

Universiteit Utrecht

When is a list sorted?

```
sorted :: [Int] → Bool
sorted []          = True
sorted (x : [])    =  ?
sorted (x : y : ys) =  ?
```

When is a list sorted?

sorted :: [Int] → Bool
sorted [] = True
sorted (x : []) = True
sorted (x : y : ys) = ?

Universiteit Utrecht

When is a list sorted?

```
sorted :: [Int] → Bool
sorted []         = True
sorted (x : [])   = True
sorted (x : y : ys) = x < y ∧ sorted (y : ys)
```

Universiteit Utrecht

Testing again

sortEnsuresSorted :: [Int] → Bool
sortEnsuresSorted xs = sorted (sort xs)

Universiteit Utrecht

Testing again

sortEnsuresSorted :: [Int] → Bool
sortEnsuresSorted xs = sorted (sort xs)

Or:

(f 'ensures' p) x = p (f x)
sortEnsuresSorted = sort 'ensures' sorted

Testing again

```
sortEnsuresSorted :: [Int] → Bool
sortEnsuresSorted xs = sorted (sort xs)
```

Or:

```
(f 'ensures' p) x = p (f x)
sortEnsuresSorted = sort 'ensures' sorted
```

```
Main⟩ quickCheck sortEnsuresSorted
Falsifiable, after 5 tests:
[5, 0, −2]
Main⟩ sort [5, 0, −2]
[0, −2, 5]
```

Correcting again

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

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

Universiteit Utrecht

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
```

Universiteit Utrecht

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
```

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

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

Universiteit Utrecht

Specifications can have bugs, too!

```
sorted :: [Int] → Bool
sorted []          = True
sorted (x : [])     = True
sorted (x : y : ys) = x < y ∧ sorted (y : ys)
```

Universiteit Utrecht

Specifications can have bugs, too!

```
sorted :: [Int] → Bool
sorted []         = True
sorted (x : [])    = True
sorted (x : y : ys) = x ≤ y ∧ sorted (y : ys)
```

Universiteit Utrecht

Are we done yet?

Is sorting specified completely by saying that

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

Universiteit Utrecht

No, not quite

evilNoSort :: $[\mathsf{Int}] \rightarrow [\mathsf{Int}]$
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.

Universiteit Utrecht

Specifying sorting

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

Our sorting function now fulfills the specification.

Universiteit Utrecht

Using QuickCheck

To use QuickCheck in your program:

import Test.QuickCheck

The simplest interface is to use

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

class Testable prop **where**
    property :: prop $\rightarrow$ Property
instance Testable Bool
instance (Arbitrary a, Show a, Testable prop) $\Rightarrow$
                Testable (a $\rightarrow$ prop)

Universiteit Utrecht

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):

Universiteit Utrecht

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).

Universiteit Utrecht

Functional properties

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

> sortPreservesLength :: ([Int] $\rightarrow$ [Int]) $\rightarrow$ [Int] $\rightarrow$ 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.

Universiteit Utrecht

Another sorting function

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

Universiteit Utrecht

Another sorting function

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

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

Universiteit Utrecht

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:

```
sortOrders :: [Int] → Bool
sortOrders xs = ordered (sort xs)
ordered :: Ord a ⇒ [a] → Bool
ordered []        = True
ordered [x]       = True
ordered (x : y : ys) = x ⩽ y ∧ ordered (y : ys)
```

Universiteit Utrecht

How to fully specify sorting (contd.)

Property 2

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

> sortPreservesElements :: [Int] $\rightarrow$ Bool
> sortPreservesElements xs = sameElements xs (sort xs)
>
> sameElements :: Eq a $\Rightarrow$ [a] $\rightarrow$ [a] $\rightarrow$ Bool
> sameElements xs ys = null (xs $\setminus\setminus$ ys) $\wedge$ null (ys $\setminus\setminus$ xs)

Universiteit Utrecht

More information about test data

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

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

Universiteit Utrecht

More information about test data

> collect :: (Testable prop, Show a) $\Rightarrow$ a $\rightarrow$ prop $\rightarrow$ 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 $\rightarrow$ collect (null xs) (p xs))
> +++ OK, passed 100 tests:
> 92% False
> 8% True

Universiteit Utrecht

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

Universiteit Utrecht

More information about test data (contd.)

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

> Main⟩ quickCheck (λxs → 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?

Universiteit Utrecht

Implications

The function insert preserves an ordered list:

```
implies :: Bool → Bool → Bool
implies x y = not x ∨ y
```

Problematic:

```
insertPreservesOrdered :: Int → [Int] → Bool
insertPreservesOrdered x xs =
    ordered xs 'implies' ordered (insert x xs)
```

Universiteit Utrecht

Implications (contd.)

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

Universiteit Utrecht

Implications (contd.)

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

But:

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

Only 12 lists have really been tested!

Universiteit Utrecht

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.

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

Now we get:

Main⟩ quickCheck (λx xs $\rightarrow$ collect (ordered xs) (iPO x xs))
∗∗∗ Gave up! Passed only 43 tests (100% True).

Universiteit Utrecht

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).

Universiteit Utrecht

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]
```

Universiteit Utrecht

Combinators for generators

```
choose    :: Random a ⇒ (a, a) → Gen a
oneof     :: [Gen a] → Gen a
frequency :: [(Int, Gen a)] → Gen a
elements  :: [a] → Gen a
sized     :: (Int → Gen a) → Gen a
```

Universiteit Utrecht

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
    arbitrary = choose (−20, 20)
  ```

- Better:

  ```
  instance Arbitrary Int where
    arbitrary = sized (λn → choose (−n, n))
  ```

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

Universiteit Utrecht

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
```

Universiteit Utrecht

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?

Universiteit Utrecht

Shrinking

The other method in Arbitrary is

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

- ▶ 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.

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.

Universiteit Utrecht

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.

Universiteit Utrecht

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.

Universiteit Utrecht

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

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