Software System Design and Implementation

Property-based Testing

Gabriele Keller

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia

COMP3141 17s1

Testing in Haskell

- The language already provides many assurances:
  - Memory safety
    - no buffer overflow
    - no access to already freed memory
    - no access to uninitialised memory
    - no freeing of already free memory

Testing in Haskell

- The language already provides many assurances:
  - Strong type safety
    - type correct programs can't go wrong/lead to undefined behaviour

Testing in Haskell

- The language already provides many assurances
    - Purity except where explicitly stated
- Testing can largely focus on logic bugs
    - It is largely sufficient to test functions in isolation (thanks to purity!)
    - Focus on properties of functions

Property-based testing

Idea: specify properties formally and test the code against those properties

An example property

```haskell
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Property: reverse commutes with append

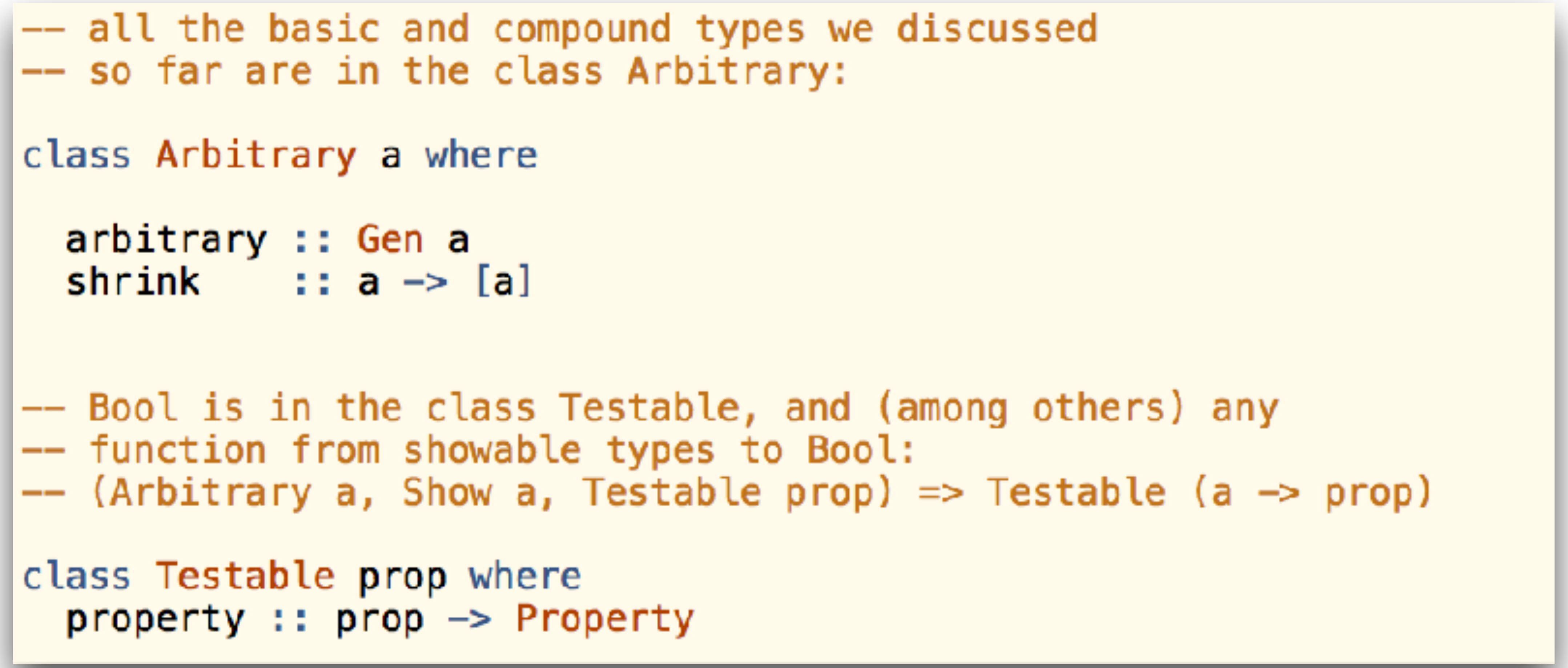
```
-- In predicate logic (more about that next week)
∀xs ys.
  reverse (xs ++ ys) ≡ reverse ys ++ reverse xs

-- In Haskell
prop_revApp xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Let's try it!

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

How can QuickCheck generate test data?

- We know that it’s not possible to generate data of any type



```haskell
-- all the basic and compound types we discussed
-- so far are in the class Arbitrary:

class Arbitrary a where

  arbitrary :: Gen a
  shrink    :: a -> [a]


-- Bool is in the class Testable, and (among others) any
-- function from showable types to Bool:
-- (Arbitrary a, Show a, Testable prop) => Testable (a -> prop)

class Testable prop where
  property :: prop -> Property
```

What is the type of `quickCheck`?

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

?

an IO action (more about this later)

Conceptual benefits of property-based testing

- Properties get specified formally
  - Encourages to think about the tested code in new ways
  - Increases understanding of tested system
  - Simple, compact test representation
- Checking is cheap
  - Checks provide feedback to debug the specification
  - Checks find bugs in the code
- This doesn't mean you should do property based testing instead of unit testing, but in addition to!

QuickCheck

- Specification of tests by properties
- Randomised test data generation
- Originally invented in the context of Haskell [Claessen & Hughes]
- Ported to Erlang, Scheme, Common Lisp, Perl, Python, Ruby, Java, Scala, F#, Standard ML, JavaScript, and C++

Core concepts of QuickCheck

- Specification of program properties
  - Using a simple specification language
  - Properties have a formal, logical meaning
  - Properties also have a well-defined operational meaning
- Property testing
  - Using random tests produced by test-data generators
  - It is cheap — so, it is done often!

Random testing in QuickCheck

- Experience shows that it works well at fine granularity
  - In purely functional code, all dependencies are explicit
  - Has been extended to cover state-based code, too
- Test-data generators
  - Type driven — that is, type-dependent generator selection
  - Built-in default generators for common types
  - Explicit user-control and custom generators are supported

Property-based versus unit testing: challenges

- Does the generated data test the cases that need to be tested?
  - Should use coverage checker in any case
- Are failures informative?
  - Try to work at a fine granularity and use shrinking
- How difficult is it to generate test data for user-defined structures?
  - QuickCheck comes with elaborate combinators for test generation

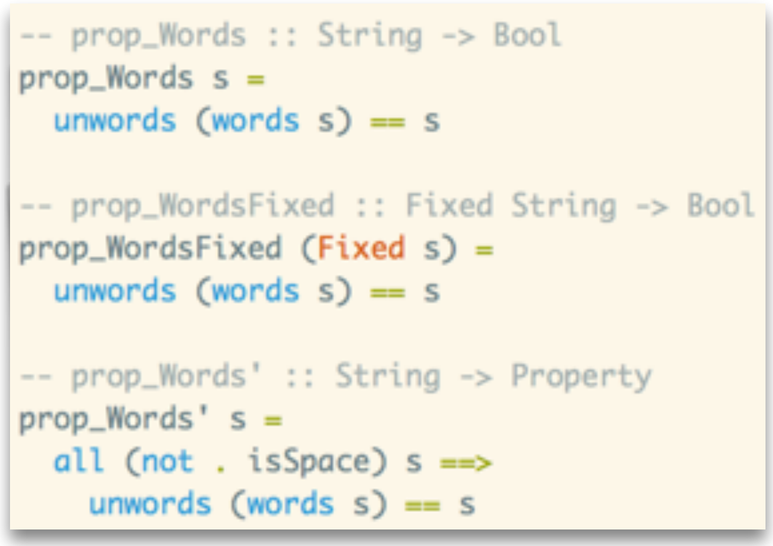
Property-based versus unit testing: advantages

- Repeated testing (as part of nightly builds and regression tests) can improve code coverage
- Properties are more compact than a set of related unit tests
    - Properties are a form of documentation checked for consistency
    - Properties cover the general case, instead of one or more examples
- Less testing code needs to be written and maintained
    - Ericsson's AXD301 ATM-switch: 1.5 million lines of Erlang code **plus** 700,000 lines of conventional testing code

A slightly larger example

```
words   :: String -> [String]   -- break into words
unwords :: [String] -> String   -- glue words together
```

- We would expect `(unwords . words)` to be the identity
- Make this into a property `prop_Words`
- Lessons:
  - The code's properties may not be what you think at first!
  - Shrinking helps to get to the bottom of bugs in properties and code



```
-- prop_Words :: String -> Bool
prop_Words s =
  unwords (words s) == s

-- prop_WordsFixed :: Fixed String -> Bool
prop_WordsFixed (Fixed s) =
  unwords (words s) == s

-- prop_Words' :: String -> Property
prop_Words' s =
  all (not . isSpace) s ==>
    unwords (words s) == s
```

Testing mergesort

A more comprehensive example

Mergesort algorithm

- Recursive divide-and-conquer algorithm

   1. If list length smaller than 2, the list is already sorted

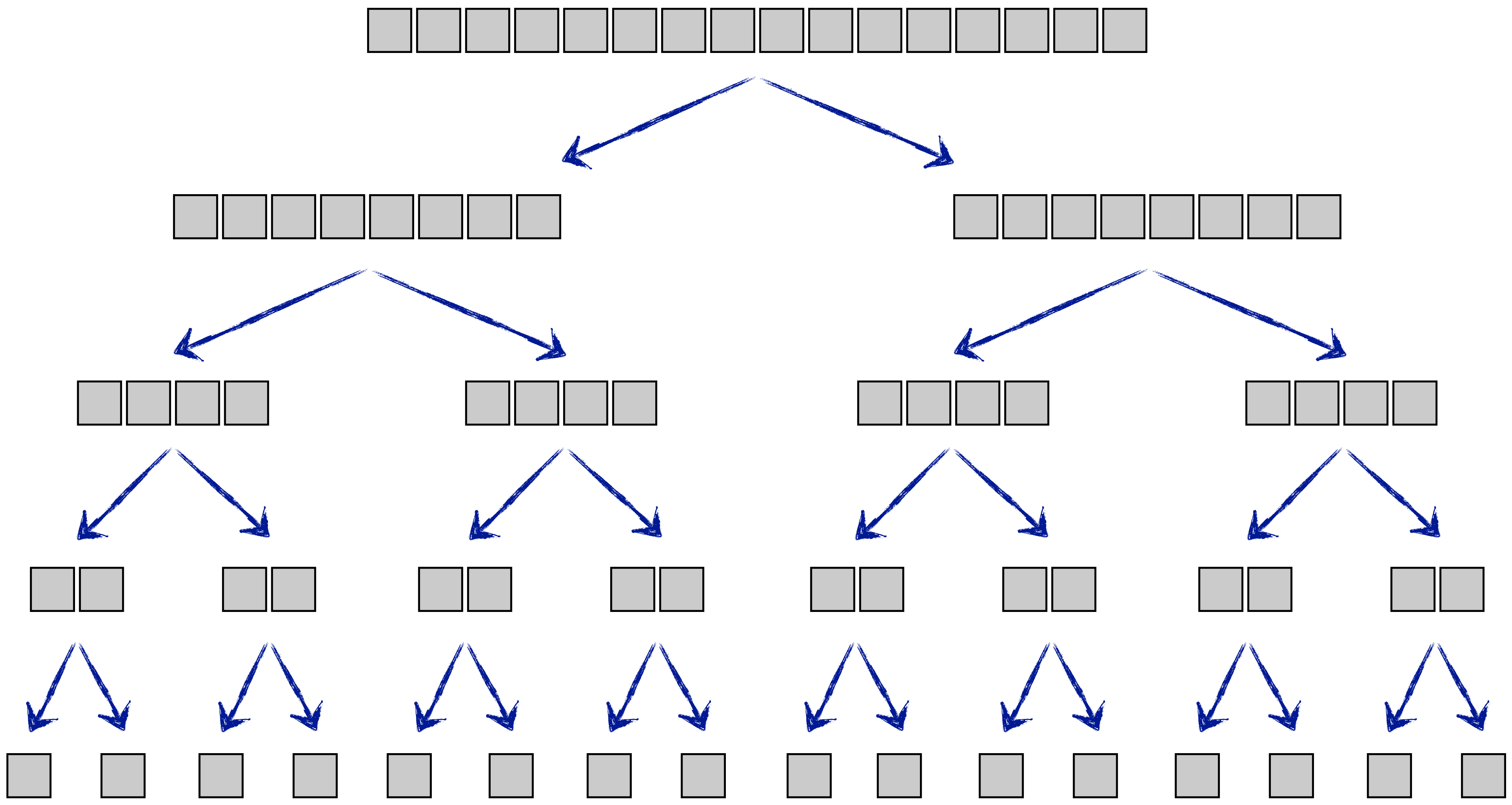
   2. Split input list into two equal halves

   3. Recursively apply mergesort to the two halves

   4. Merge the two sorted halves

  - Merging sorted lists is cheap (linear in the length of the lists)

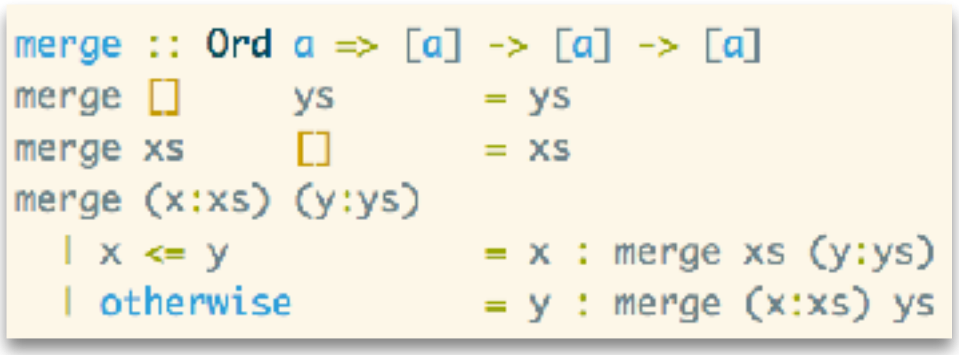
What does the call tree for mergesort look like?



Merging sorted lists

- Given two lists in ascending order
- Produce a list that combines the elements of these two lists and is also in ascending order

```
merge :: Ord a => [a] -> [a] -> [a]
```

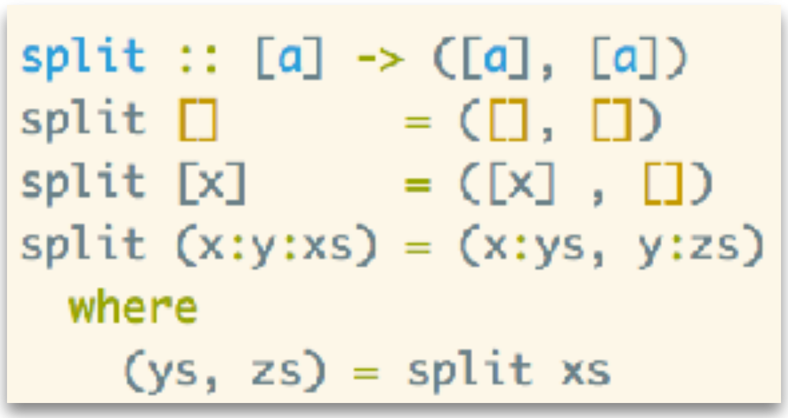


```
merge :: Ord a => [a] -> [a] -> [a]
merge []      ys        = ys
merge xs      []        = xs
merge (x:xs) (y:ys)
  | x <= y              = x : merge xs (y:ys)
  | otherwise           = y : merge (x:xs) ys
```

Splitting a list into even halves

- Partition a list into two lists, such that both have (almost) the same length

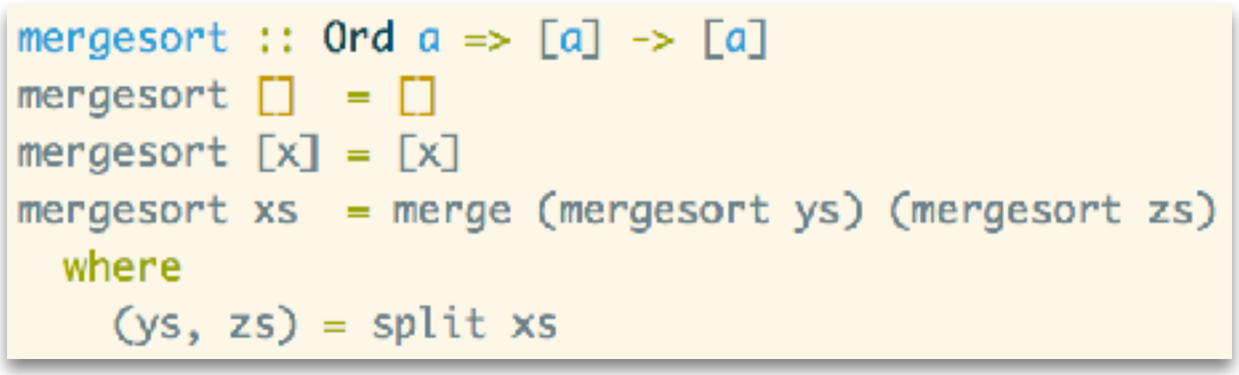
```haskell
split :: [a] -> ([a], [a])
```



```haskell
split :: [a] -> ([a], [a])
split []         = ([], [])
split [x]        = ([x] , [])
split (x:y:xs) = (x:ys, y:zs)
  where
    (ys, zs) = split xs
```

Putting it all together

```haskell
mergesort :: Ord a => [a] -> [a]
```



```haskell
mergesort :: Ord a => [a] -> [a]
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort ys) (mergesort zs)
  where
    (ys, zs) = split xs
```

```haskell
merge     :: Ord a => [a] -> [a] -> [a]
mergesort :: Ord a => [a] -> [a]
```

- `merge` must preserve the length of the sorted list: `prop_preservesLength`
- Sorting is idempotent: `prop_idempotent`
- The first element in a sorted list must be its minimum: `prop_minimum`

Let's define these properties!

- The sorted list must be ordered: `prop_ordered`
- The output must be a permutation of the input: `prop_permutation`
- The last element in a sorted list must be its maximum: `prop_maximum`
- Interaction of sorting and append: `prop_append`

Let's define these properties, too!

Testing against a model

- For many tricky (especially for optimised algorithms), there is a simpler, less efficient one
- The tricky algorithm is correct if it produces the same output as the simple one
- We call the simple (maybe even naive) implementation a model
- Test the real implementation against the model

Constraining generators

- The function merge must produce an ordered list from two ordered lists

How do we capture this in a property?

Constraining generators

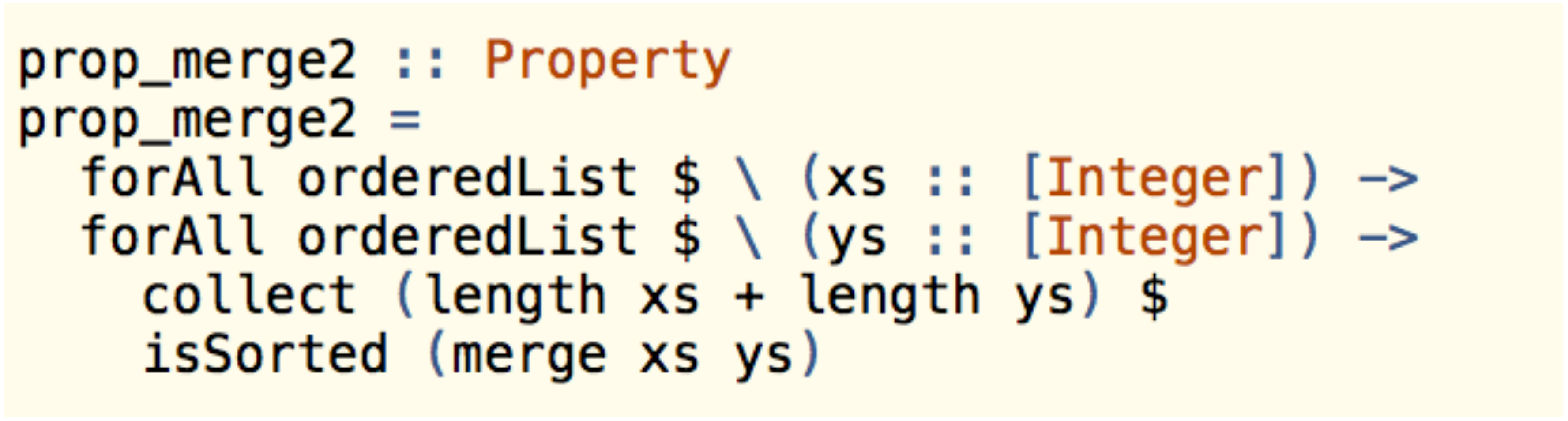
- The function merge must produce an ordered list from two ordered lists

```
prop_merge0 :: [Int] -> [Int] -> Bool
prop_merge0 xs ys =
  if (isSorted xs &&
        isSorted ys)
    then isSorted (merge xs ys)
    else True
```

```
prop_merge1 :: [Int] -> [Int] -> Property
prop_merge1 xs ys =
  isSorted xs ==>
  isSorted ys ==>
    collect (length xs, length ys) $
    isSorted (merge xs ys)
```

```
orderedList
  :: (Ord a, Arbitrary a) => Gen [a]
```

```
forAll
  :: (Show a, Testable prop) => Gen a -> (a -> prop) -> Property
```



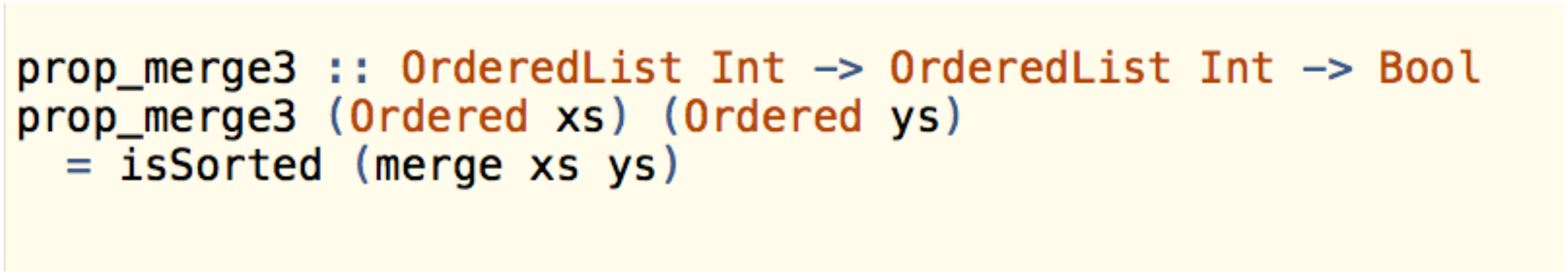
```
prop_merge2 :: Property
prop_merge2 =
  forAll orderedList $ \ (xs :: [Integer]) ->
  forAll orderedList $ \ (ys :: [Integer]) ->
    collect (length xs + length ys) $
    isSorted (merge xs ys)
```

- Alternative: type-level modifier:

```
newtype OrderedList a
```

- Constructor:

```
Ordered :: [a] -> OrderedList a
```



```
prop_merge3 :: OrderedList Int -> OrderedList Int -> Bool
prop_merge3 (Ordered xs) (Ordered ys)
  = isSorted (merge xs ys)
```

Quality of tests

- How good are your tests?
    - Have you checked that every special case works correctly?
    - Is all code exercised in the tests?
    - Even if all code is exercised, is it exercised in all contexts?

Code coverage

- A whole family of measures is used to judge the degree of code coverage
- The various measures have varying precision
- They vary in the rigour required of tests to achieve full coverage
- They differ in their suitability for tool support
- May be combined

Let's look at some of the more important ones.

Function coverage

- Has every function been called?
- Easy to measure
- Easy to provide tool support
- Rather coarse grain — i.e., misses many possible execution paths

Entry/exit coverage

- Has every possible call and return of the function been executed?
- Easy to measure
- Easy to provide tools support
- Somewhat more comprehensive than plain function coverage

Statement/expression coverage

- Has each statement been executed?
- Somewhat harder to measure
- Easy to provide tool support
- Still fairly coarse grain, but substantially more comprehensive than function coverage or entry/exit coverage

Branch/decision coverage

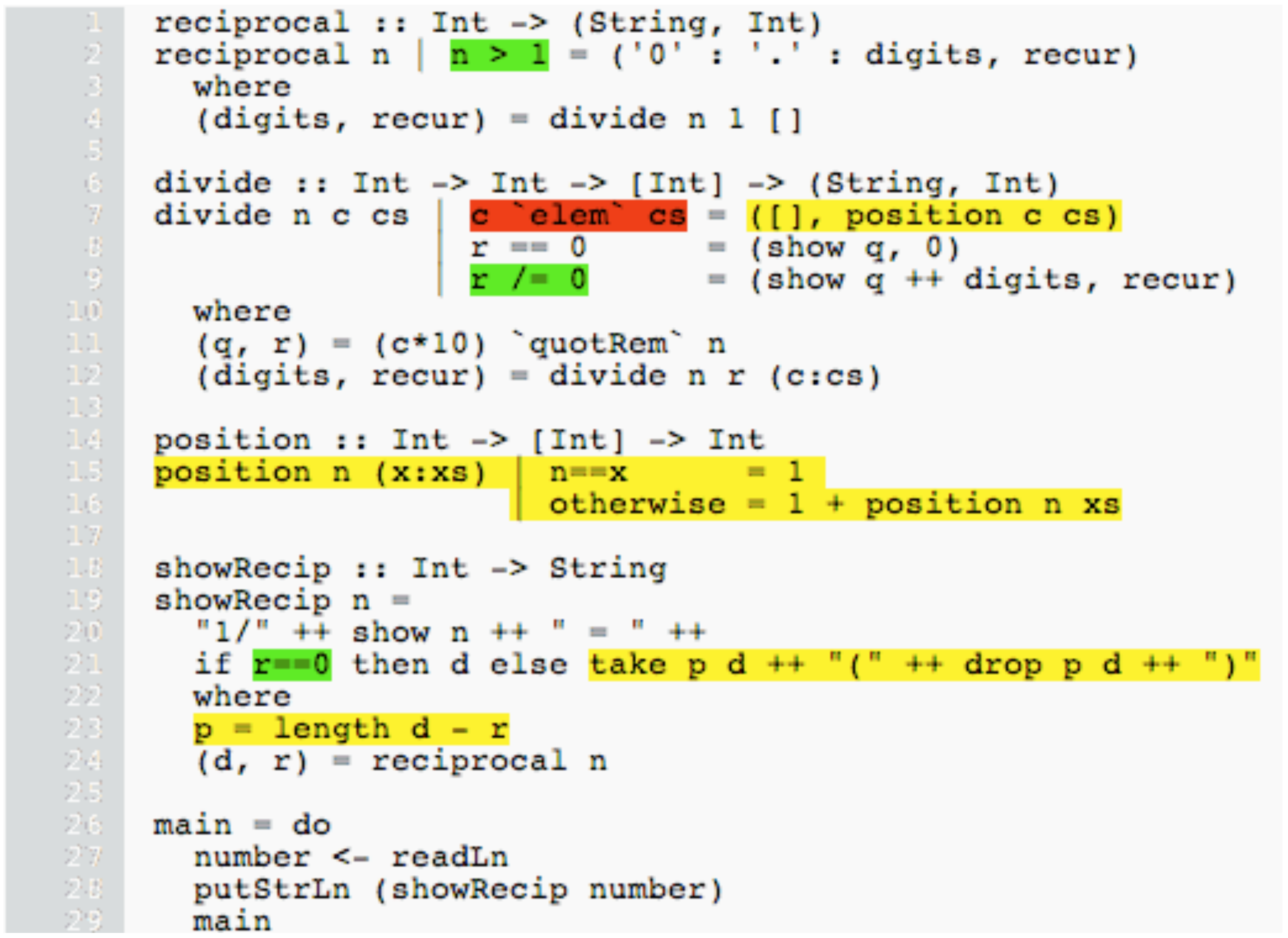
- Has every control flow alternative been executed?
- Measuring and tool support as for statement coverage
- In an imperative language, more comprehensive than statement coverage
    - There may be more than one control flow edge that leads to a given statement
    - In branch/decision coverage, they must all have been taken
- Related condition coverage: have all conditions been `True` and `False`

Path coverage

- Has every possible route through a program been executed?
- This is much harder to measure
    - Requires that each combination of branches which leads to a unique path to be executed
- Full path coverage is infeasible
    - Loops may lead to infinite numbers of paths
- Even when disregarding repetitions, path coverage is still very expensive

Haskell Program Coverage (HPC)

- Coverage checker integrated with GHC
- Instruments a compiled program to track and log code execution
- Produces coverage statistics and annotations of source listings
- Implements function coverage, condition coverage & expression coverage



```
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
  where
  (digits, recur) = divide n 1 []

divide :: Int -> Int -> [Int] -> (String, Int)
divide n c cs | c `elem` cs = ([], position c cs)
              | r == 0      = (show q, 0)
              | r /= 0      = (show q ++ digits, recur)
  where
  (q, r) = (c*10) `quotRem` n
  (digits, recur) = divide n r (c:cs)

position :: Int -> [Int] -> Int
position n (x:xs) | n==x      = 1
                  | otherwise = 1 + position n xs

showRecip :: Int -> String
showRecip n =
  "1/" ++ show n ++ " = " ++
  if r==0 then d else take p d ++ "(" ++ drop p d ++ ")"
  where
  p = length d - r
  (d, r) = reciprocal n

main = do
  number <- readLn
  putStrLn (showRecip number)
  main
```

- Yellow: functions and expression that were not executed
- Red: conditions that never evaluated to True
- Green: conditions that never evaluated to False