Laziness & testing

Advanced functional programming - Lecture 3

Wouter Swierstra

Universiteit Utrecht

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

Laziness

A simple expression

```
square :: Integer -> Integer
square x = x * x

square (1 + 2)
= -- magic happens in the computer
9
```

How do we reach that final value?

Strict or eager or call-by-value evaluation

In most programming languages:

1. Evaluate the arguments completely
2. Evaluate the function call

```
square (1 + 2)
= -- evaluate arguments
square 3
= -- go into the function body
3 * 3
=
9
```

Non-strict or call-by-name evaluation

Arguments are replaced as-is in the function body

```
square (1 + 2)
= -- go into the function body
(1 + 2) * (1 + 2)
= -- we need the value of (1 + 2) to continue
3 * (1 + 2)
=
3 * 3
=
9
```

Does call-by-name make any sense?

In the case of `square`, non-strict evaluation is worse

Is this always the case?

Does call-by-name make any sense?

In the case of `square`, non-strict evaluation is worse

Is this always the case?

```
const x y = y  -- forget about x
```

```
-- Call-by-value
const (1 + 2) 5
=
const 3 5
=
5
```

```
-- Call-by-name
const (1 + 2) 5
=
5
```

Sharing expressions

```
square (1 + 2)
=
(1 + 2) * (1 + 2)
```

Why redo the work for `(1 + 2)`?

Sharing expressions

```
square (1 + 2)
=
(1 + 2) * (1 + 2)
```

Why redo the work for `(1 + 2)`?

We can share the evaluated result

```
square (1 + 2)
=
Δ * Δ
□___□___ (1 + 2)
         = 3
=
9
```

Universiteit Utrecht

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

Lazy evaluation

Haskell uses a **lazy** evaluation strategy

- Expressions are not evaluated *until needed*
- Duplicate expressions are *shared*

Lazy evaluation never requires more steps than call-by-value

Each of those not-evaluated expressions is called a **thunk**

Does it matter?

Is it possible to get different outcomes using different evaluation strategies?

Does it matter?

Is it possible to get different outcomes using different evaluation strategies?

Yes and no

Does it matter? - Correctness and efficiency

The *Church-Rosser Theorem* states that for *terminating* programs the result of the computation does *not* depend on the evaluation strategy

But…

1. Performance might be different
   - As `square` and `const` show
2. This applies only if the program terminates
   - What about infinite loops?
   - What about exceptions?

Termination

```
loop x = loop x
```

- This is a well-typed program
- But `loop 3` never terminates

```
-- Eager
const (loop 3) 5
=
const (loop 3) 5
=
...
```

```
-- Lazy
const (loop 3) 5
=
5
```

Lazy evaluation terminates more often than eager

Build your own control structures

```
if_ :: Bool -> a -> a -> a
if_ True  t _ = t
if_ False _ e = e
```

- In eager languages, `if_` evaluates both branches
- In lazy languages, only the one being selected

For that reason,

- In eager languages, `if` has to be *built-in*
- In lazy languages, you can build your *own control structures*

Short-circuiting

```
(&&) :: Bool -> Bool -> Bool
False && _ = False
True  && x = x
```

- In eager languages, `x && y` evaluates both conditions
  - But if the first one fails, why bother?
  - C/Java/C# include a built-in *short-circuit* conjunction
- In Haskell, `x && y` only evaluates the second argument if the first one is `True`
  - `False && (loop True)` terminates

“Until needed”

How does Haskell know *how much* to evaluate?

- By default, everything is kept in a thunk
- When we have a case distinction, we evaluate enough to distinguish which branch to follow

```
take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n-1) xs
```

- If the number is `0` we do not need the list at all
- Otherwise, we need to distinguish `[]` from `x:xs`

Weak Head Normal Form

An expression is in **weak head normal form** (WHNF) if it is:

- A constructor with (possibly non-evaluated) data inside
  - `True` or `Just (1 + 2)`
- An anonymous function
  - The body might be in any form
  - `\x -> x + 1` or `\x -> if_ True x x`
- A built-in function applied to too few arguments

Every time we need to distinguish the branch to follow the expression is evaluated until its WHNF

```
#include <LazyEval.pdf>
```

Case study: `foldl'`

From long, long time ago…

```
foldl _ v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
```

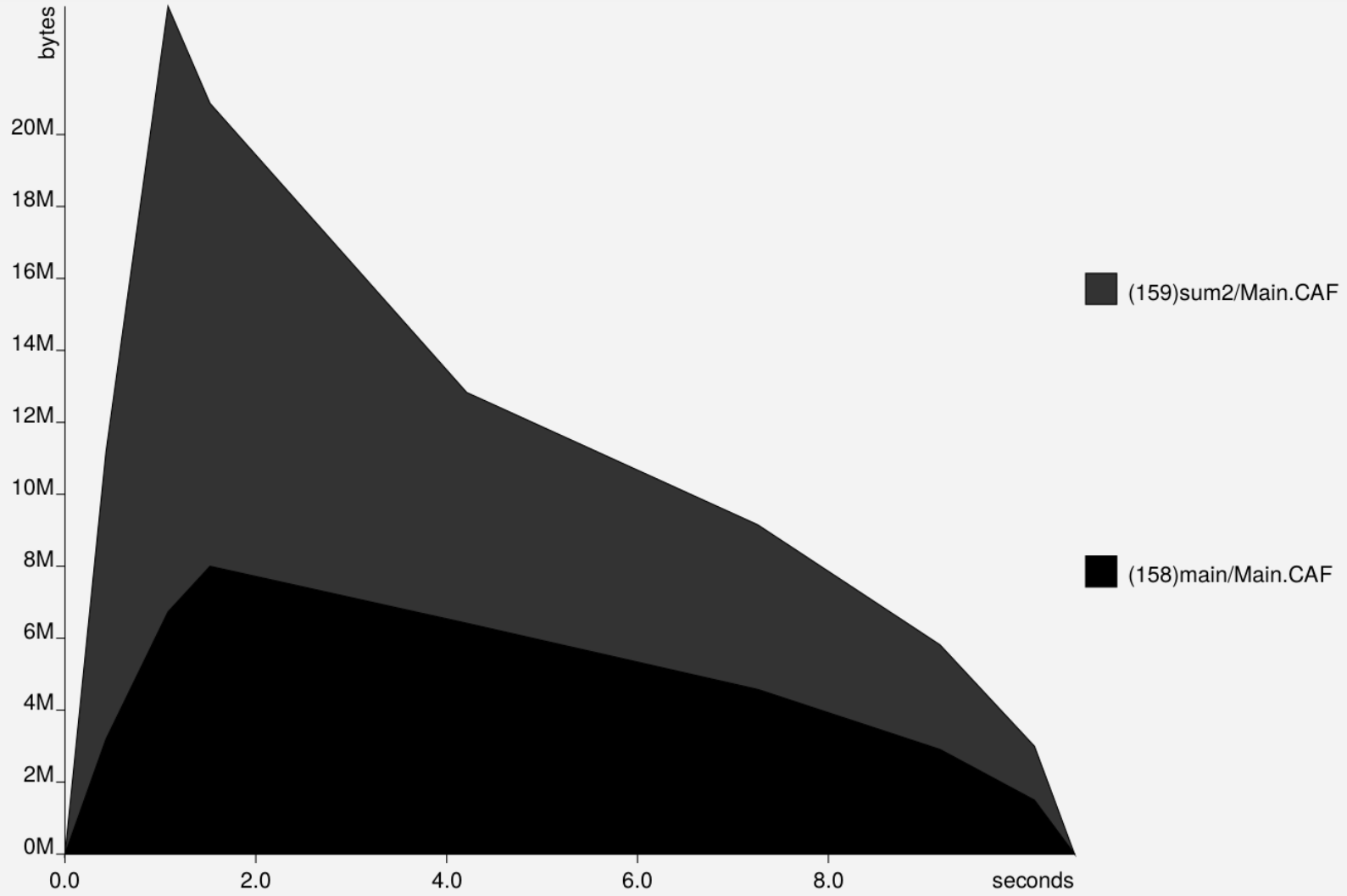
```
  foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) ((0 + 1) + 2) [3]
= foldl (+) (((0 + 1) + 2) + 3) []
= ((0 + 1) + 2) + 3
```

Case study: `foldl'`

```
  foldl (+) 0 [1,2,3]
= ((0 + 1) + 2) + 3
```

- Each of the additions is kept in a thunk
  - Some memory need to be reserved
  - They have to be GC'ed after use

Case study: `foldl'`



Case study: `foldl'`

Just performing the addition is faster!

- Computers are fast at arithmetic
- We want to *force* additions before going on

```
  foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) 1 [2,3]
= foldl (+) (1 + 2) [3]
= foldl (+) 3 [3]
= foldl (+) (3 + 3) []
= foldl (+) 6 []
= 6
```

Forcing evaluation

Haskell has a primitive operation to force

```
seq :: a -> b -> b
```

A call of the form `seq x y`

- First evaluates x up to WHNF
- Then it proceeds normally to compute y

Usually, y depends on x somehow

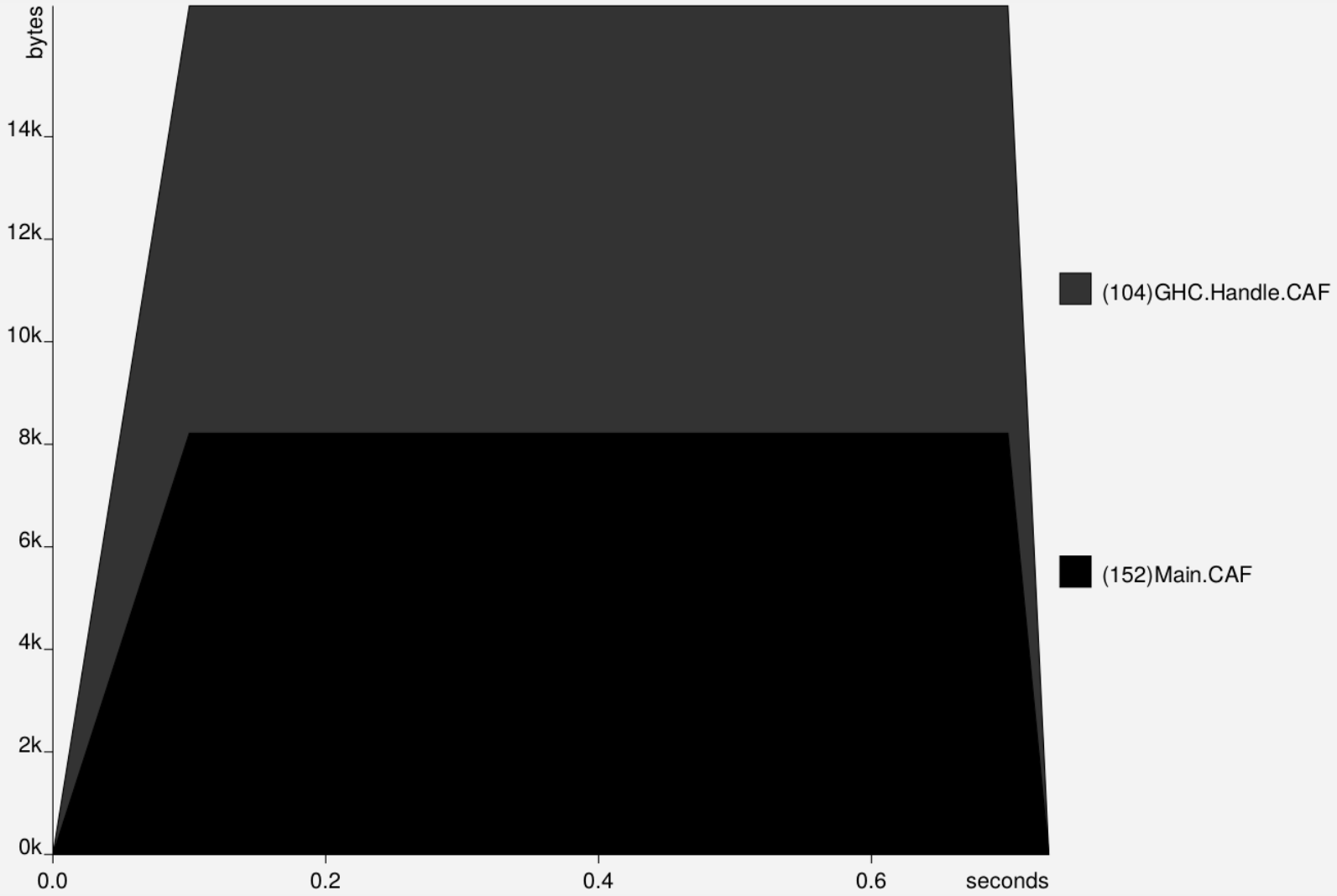
Case study: `foldl'`

We can write a new version of `foldl` which forces the accumulated value before recursion is unfolded

```
foldl' _ v []     = v
foldl' f v (x:xs) = let z = f v x
                    in z `seq` foldl' f z xs
```

This version solves the problem with addition

Case study: `foldl'`



Strict application

Most of the times we use `seq` to force an argument to a function, that is, *strict application*

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Because of sharing, x is evaluated only once

```
foldl' _ v []     = v
foldl' f v (x:xs) = ((foldl' f) $! (f v x)) xs
```

Profiling

Something about (in)efficiency

We have seen that Haskell programs:

- can be very short
- and sometimes very inefficient

Question:

How to find out where time is spent?

Something about (in)efficiency

We have seen that Haskell programs:

- can be very short
- and sometimes very inefficient

Question:

How to find out where time is spent?

Answer:

Use profiling

Laziness is a double-edged sword

- With laziness, we are sure that things are evaluated only as much as needed to get the result.
- But, being lazy means holding lots of thunks in memory:
  - Memory consumption can grow quickly.
  - Performance is not uniformly distributed.

Question:

How to find out where memory is spent?

How to find out where to sprinkle `seq`s?

Laziness is a double-edged sword

- With laziness, we are sure that things are evaluated only as much as needed to get the result.
- But, being lazy means holding lots of thunks in memory:
  - Memory consumption can grow quickly.
  - Performance is not uniformly distributed.

Question:

How to find out where memory is spent?

How to find out where to sprinkle `seq`s?

Answer:

Use profiling

Example: segs

`segs xs` computes all the consecutive sublists of `xs`.

```
segs []     = [[]]
segs (x:xs) = segs xs ++ map (x:) (inits xs)
```

```
> segs [2,3,4]
[[],[4],[3],[3,4],[2],[2, 3],[2,3,4]]
```

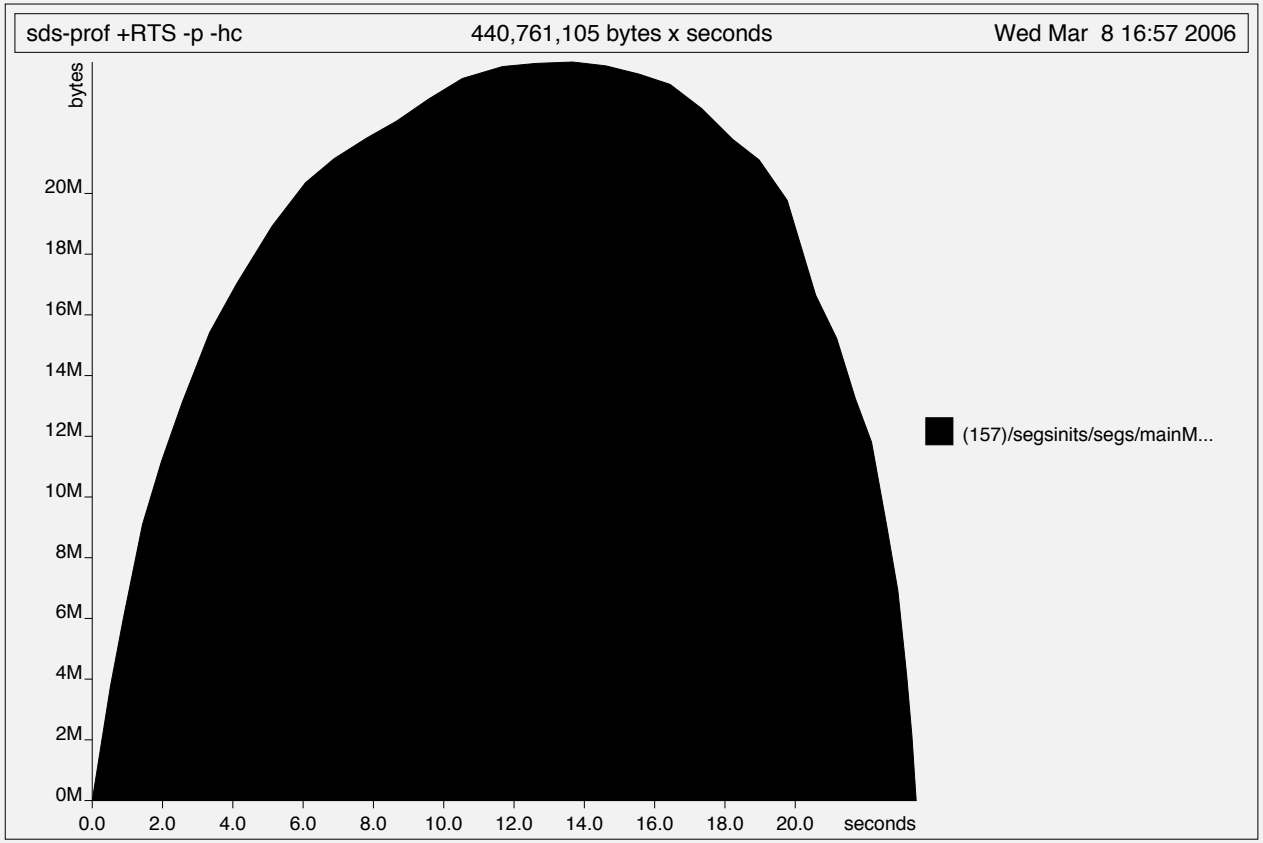
This implementation is extremely inefficient.

Example: `segsinits`

We can compute `inits` and `segs` at the same time.

```
segsinits []       =  ([[]], [[]])
segsinits (x:xs)   =
  let (segsxs, initsxs)  = segsinits xs
      newinits           = map (x:) initsxs
  in (segsxs ++ newinits, [] : newinits)
segs = fst . segsinits
```

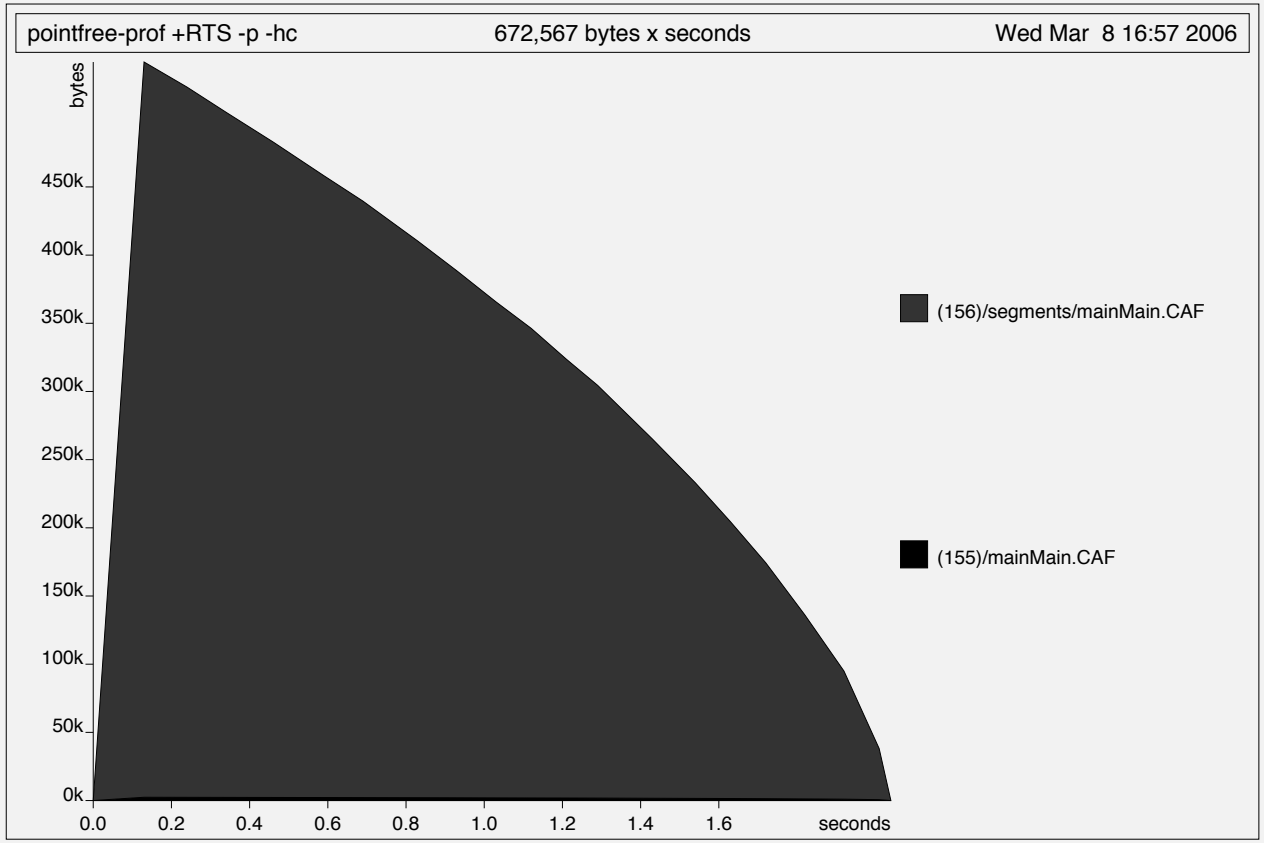
Heap profile for `segsinits`



Example: `pointfree`

```
pointfree =
  let  p    = not . null
       next = filter p . map tail . filter p
  in concat . takeWhile p . iterate next . inits
```

Heap profile for `pointfree`

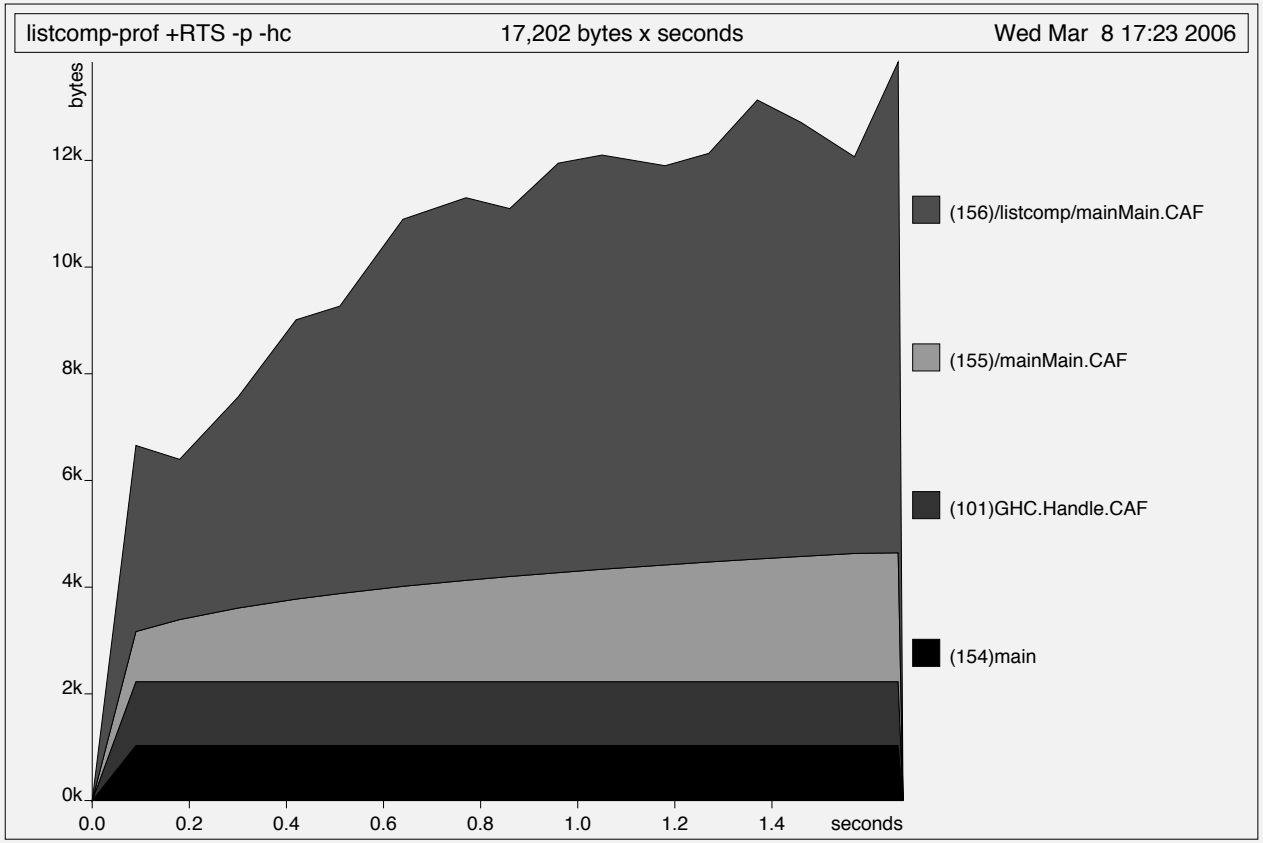


Example: `listcomp`

`segs` are just the `tails` of the `inits`!

```
listcomp xs =
  [] : [ t | i <- inits xs
           , t <- tails i
           , not (null t) ]
main =
  print (length (concat
    (listcomp [1 :: Int .. 300])))
```

Heap profile for `listcomp`



How to produce these?

```
prompt> ghc -prof -auto-all -o listcomp-prof
            -O2 Segments.hs
prompt> ./listcomp-prof +RTS -hc -p
4545100
prompt> hp2ps listcomp-prof.hp
```

Program Correctness

Testing and correctness

- When is a program correct?

Testing and 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?

Equational reasoning

- “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:

Equational reasoning

- “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:

```
ones  = 1:    ones
ones' = 1:1:  ones'
```

Referential transparency

In most functional languages like ML or OCaml, there is no referential transparency:

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

Referential transparency

In most functional languages like ML or OCaml, there is no referential transparency:

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

But we cannot replace the last line with `1 + f 2`, even though `f 1 = 1`.

Referential transparency in Haskell

- Haskell is referentially transparent – all side-effects are tracked by the IO monad.

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

Note that the type of `f` is `Int -> IO Int` – we cannot safely make the substitution we proposed previously.

Referential transparency

Because we can safely replace equals for equals, we can *reason* about our programs – this is something you already saw in the course on functional programming.

For example to prove some statement `P xs` holds for all lists `xs`, we need to show:

- `P []` – the base case;
- for all `x` and `xs`, `P xs` implies `P (x:xs)`.

Universiteit Utrecht

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

Equational reasoning

- Equational reasoning can be an elegant way to prove properties of a program.
- Equational reasoning can be used to establish a relation between an “obviously correct” Haskell program (a specification) and an efficient Haskell program.
- Equational reasoning can become quite long…
- Careful with special cases (laziness):
  - undefined values;
  - partial functions;
  - infinite values.

You can formalize such proofs in other systems such as Agda, Coq or Isabelle.

QuickCheck

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.

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.

Case study: insertion sort

```
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
```

Properties of insertion sort

We can now try to prove that for all lists `xs`,

```
length (sort xs) == length xs.
```

- The base case is trivial.
- The inductive case requires a lemma relating `insert` and `length` – suggestions?

Case study: insertion sort

Consider the following (buggy) implementation of insertion sort:

```
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
```

Let's try to debug it using QuickCheck.

How to write a specification?

A good specification is

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

A good specification for a particular problem, such as sorting, should distinguish sorting from all other operations on lists, without forcing us to use a particular sorting algorithm.

A first approximation

Certainly, sorting a list should not change its length.

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

We can test by invoking the function :

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

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
```

Which branch does not preserve the list length?

A new attempt

```
> 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
```

Properties are first-class objects

```
(f `preserves` p) x = p x == p (f x)

sortPreservesLength  =  sort `preserves` length

idPreservesLength    =  id `preserves` length
```

So `id` also preserves the lists length:

```
> quickCheck idPreservesLength
OK, passed 100 tests.
```

We need to refine our spec.

When is a list sorted?

We can define a predicate that checks if a list is sorted:

```
isSorted :: [Int] -> Bool
isSorted []         = True
isSorted [x]        = True
isSorted (x:y:xs)   = x < y && isSorted (y:xs)
```

And use this to check that sorting a list produces a list that `isSorted`.

Testing again

```
> quickCheck sortEnsuresSorted
Falsifiable, after 5 tests:
[5,0,-2]
> sort [5,0,-2]
[0,-2,5]
```

We're still not quite there…

Debugging `sort`

What's wrong now?

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

insert :: Int -> [Int] -> [Int]
```

Debugging `sort`

What's wrong now?

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

insert :: Int -> [Int] -> [Int]
```

We are not recursively sorting the tail in `sort`.

Another bug

```
> quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
[4,2,2]

> sort [4,2,2]
[2,2,4]
```

This is correct. What is wrong?

Another bug

```
> quickCheck sortEnsuresSorted
Falsifiable, after 7 tests:
[4,2,2]

> sort [4,2,2]
[2,2,4]
```

This is correct. What is wrong?

```
> isSorted [2,2,4]
False
```

Fixing the spec

The `isSorted` spec reads:

```
sorted :: [Int] -> Bool
sorted []         =  True
sorted (x:[])     =  True
sorted (x:y:ys)   =  x < y && sorted (y : ys)
```

Why does it return `False`? How can we fix it?

Are we done yet?

Is sorting specified completely by saying that

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

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) 1
```

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

```
permutes :: ([Int] -> [Int]) -> [Int] -> Bool
permutes f xs = f xs `elem` permutations xs

sortPermutes :: [Int] -> Bool
sortPermutes xs = sort `permutes` xs
```

This completely specifies sorting and our algorithm passes the corresponding tests.

How to use QuickCheck

To use QuickCheck in your program:

```
import Test.QuickCheck
```

Define properties.

Then call to test the properties.

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

The type of `quickCheck`

The type of is an *overloaded* type:

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

- The argument of is a property of type `prop`
- The only restriction on the type is that it is in the `Testable` *type class*.
- When executed, prints the results of the test to the screen – hence the result type.

Which properties are `Testable`?

So far, all our properties have been of type :

```
sortPreservesLength :: [Int] -> Bool
sortEnsuresSorted :: [Int] -> Bool
sortPermutes :: [Int] -> Bool
```

When used on such properties, QuickCheck generates random integer lists and verifies that the result is `True`.

If the result is for 100 cases, this success is reported in a message.

If the result is `False` for a test case, the input triggering the result is printed.

Other example properties

```
appendLength :: [Int] -> [Int] -> Bool
appendLength xs ys =
  length xs + length ys == length (xs ++ ys)

plusIsCommutative :: Int -> Int -> Bool
plusIsCommutative m n = m + n == n + m

takeDrop :: Int -> [Int] -> Bool
takeDrop n xs = take n xs ++ drop n xs == xs

dropTwice :: Int -> Int -> [Int] -> Bool
dropTwice m n xs =
  drop m (drop n xs) == drop (m + n) xs
```

Other forms of properties – contd.

```
> quickCheck takeDrop
OK, passed 100 tests.

> quickCheck dropTwice
Falsifiable after 7 tests.
1
-1
[0]

> drop (-1) [0]
[0]

> drop 1 (drop (-1) [0])
[]
```

Nullary properties

A property without arguments is also possible:

```
lengthEmpty :: Bool
lengthEmpty = length [] == 0

wrong :: Bool
wrong = False
```

```
> quickCheck lengthEmpty
OK, passed 100 tests.

> quickCheck wrong
Falsifiable, after 0 tests.
```

QuickCheck vs unit tests

No random test cases are involved for nullary properties.

QuickCheck subsumes unit tests.

Properties

Recall the type of `quickCheck`:

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

We can now say more about when types are `Testable`:

- testable properties usually are functions (with any number of arguments) resulting in a `Bool`

What argument types are admissible?

QuickCheck has to know how to produce random test cases of such types.

Properties – continued

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

instance Testable Bool where
      ...
instance  (Arbitrary a, Show a, Testable b) =>
          Testable (a -> b) where
```

We can test any Boolean value or any testable function for which we can generate arbitrary input.

More information about test data

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

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

```
> let sPL = sortPreservesLength
> quickCheck (\xs -> collect (null xs) (sPL xs))
OK, passed 100 tests.
96% False
4% True.
```

The result implies that not all test cases are distinct.

More information about test data – contd.

```
> quickCheck (\xs -> collect (length xs `div` 10)
                             (sPL xs))
+++ OK, passed 100 tests.
 26% 0.
 21% 1.
 15% 2.
 10% 5.
 10% 3.
 ...
```

Most lists are small in size: QuickCheck generates small test cases first, and increases the test case size for later tests.

More information about test data (contd.)

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

```
> quickCheck (\ xs -> collect xs (sPL xs))
OK, passed 100 tests:
6% []
1% [9,4,-6,7]
1% [9,-1,0,-22,25,32,32,0,9,...
...
```

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

Implications

The function insert preserves an ordered list:

```
implies :: Bool -> Bool -> Bool
implies x y = not x || y

insertPreservesOrdered :: Int -> [Int] -> Bool
insertPreservesOrdered x xs =
  sorted xs `implies` sorted (insert x xs)
```

Implications – contd.

```
> quickCheck insertPreservesOrdered
OK, passed 100 tests.
```

But:

```
> let iPO = insertPreservesOrdered
> quickCheck (\x xs -> collect (sorted xs)
                               (iPO x xs))
OK, passed 100 tests.
88% False
12% True
```

For **88** test cases, `insert` has not actually been relevant.

Implications – contd.

The solution is to use the QuickCheck implication operator:

```
(==>) :: (Testable prop) =>
           Bool -> prop -> Property

instance Testable Property
```

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

```
iPO :: Int -> [Int] -> Property
iPO x xs = sorted xs ==> sorted (insert x xs)
```

Now, lists that are not sorted are discarded and do not contribute towards the goal of 100 test cases.

Implications – contd.

We can now easily run into a new problem:

```
iPO :: Int -> [Int] -> Property
iPO x xs = length xs > 2 && sorted xs ==>
             sorted (insert x xs)
```

We try to ensure that lists are not too short, but:

```
> quickCheck (\x xs -> collect (sorted xs)
                               (iPO x xs))
Arguments exhausted after 20 tests (100% True).
```

The chance that a random list is sorted is extremely small. QuickCheck will give up after a while if too few test cases pass the precondition.

Configuring QuickCheck

```
quickCheckWith :: Testable prop =>
  Args -> prop -> IO ()

data Args where
  replay :: Maybe (StdGen, Int)
  -- should we replay a previous test?
  maxSuccess :: Int
  -- max number of successful tests
  --   before succeeding
  maxDiscardRatio :: Int
  -- max number of discarded tests
  --   per successful test
  maxSize :: Int
  --max test case size
  ...
```

Generators

- Instead of increasing the number of test cases to generate, it is usually better to write a custom random generator.
- Generators belong to an abstract data type `Gen`. Think of 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. The default generators for datatypes are specified by defining instances of class `Arbitrary`:

```
class Arbitrary a where
  arbitrary :: Gen a
  ...
```

Generator combinators

```
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
```

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]
```

Generating random numbers

- A simple possibility:

```
instance Arbitrary Int where
  arbitrary = choose (-20,20)
```

- Better:

```
instance Arbitrary Int where
  arbitrary = sized (\ n -> choose (-n,n))
```

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

How to generate sorted lists

Idea: Adapt the default generator for lists.

The following function turns a list of integers into a sorted list of integers:

```
mkSorted :: [Int] -> [Int]
mkSorted []         =  []
mkSorted [x]        =  [x]
mkSorted (x:y:ys)   =  x : mkSorted ((x + abs y : ys)
```

For example:

```
> mkSorted [1,2,-3,4]
[1,3,6,10]
```

Random generator

The generator can be adapted as follows:

```
genSorted :: Gen [Int]
genSorted =  do
               xs <- arbitrary
               return (mkSorted xs)
```

Using a custom generator

There is another function to construct properties provided by QuickCheck, passing an explicit generator:

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

This is how we use it:

```
iPO :: Int -> Property
iPO x = forAll genSorted
           (\ xs -> length xs > 2 && sorted xs ==>
                      sorted (insert x xs))
```

Loose ends: Shrinking

`Arbitrary` revisited

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
```

The other method in is

```
shrink :: (Arbitrary a) => a -> [a]
```

- Maps each value to a number of ‘structurally smaller’ values.
- When a failing test case is discovered, is applied repeatedly until no smaller failing test case can be obtained.

Program coverage

To assess the quality of your test suite, it can be very useful to use GHC's *program coverage* tool:

```
$ ghc -fhpc Suite.hs --make
$ ./Suite
$ hpc report Suite --exclude=Main --exclude=QC
   18% expressions used (30/158)
    0% boolean coverage (0/3)
         0% guards (0/3), 3 unevaluated
       100% 'if' conditions (0/0)
       100% qualifiers (0/0)
       ...
```

This also generates a `.html` file showing which code has (not) been executed.

module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module Prettify2	42%	9/21		23%	8/34		18%	30/158	
Program Coverage Total	42%	9/21		23%	8/34		18%	30/158	

screenshot

```haskell
data Doc = Empty
         | Char Char
         | Text String
         | Line
         | Concat Doc Doc
         | Union Doc Doc
         deriving (Show,Eq)

{-- /snippet Doc --}

instance Monoid Doc where
     mempty  = empty
     mappend = (<>)

{-- snippet append --}
empty :: Doc
(<>)  :: Doc -> Doc -> Doc
{-- /snippet append --}

empty = Empty

Empty <> y = y
x <> Empty = x
x <> y = x `Concat` y

char :: Char -> Doc
char c = Char c
```

screenshot

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

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.
- Use it!

However, keep in mind that writing good tests still requires training, and that tests can have bugs, too.

Further reading

Required:

- Chapter 11 of Real World Haskell
- Koen Claessen and John Hughes – QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs.

Background:

- John Hughes – Software Testing with QuickCheck
- Colin Runciman, Matthew Naylor, Fredrik Lindblad – Smallcheck and lazy smallcheck: automatic exhaustive testing for small values

Bonus: tests with HSpec and Cabal

HSpec

HSpec is a testing framework for Haskell inspired by RSpec.

- Essentially, a DSL to define tests.
- Integrates QuickCheck, SmallCheck and HUnit.

```
main = hspec $ do
  describe "insert" $ do
    it "preserves length" $
      property $ \xs ->
        length (sort xs) == length xs
    it "ensures sortedness" $
      property $ \xs -> sorted (sort xs)
```

Test suites in Cabal

In addition to `library` and `executable` stanzas, Cabal also supports test suites.

```
test-suite test
  type:             exitcode-stdio-1.0
  main-is:          Test.hs
  hs-source-dirs:   test
  build-depends:    your-library,
                    your-test-framework
```

Originally, it was planned that test frameworks would use a unified interface for tests, but this has never materialized.

HSpec automatic discovery

Just include the following line in `test/Test.hs`:

```
{-# OPTIONS_GHC -F -pgmF hspec-discover #-}
```

`hspec-discover` is a preprocessor which finds all the files in the `test` folder which end in `Spec.hs` and contain a definition for `spec :: Spec`.