Tools and Testing

AFP Summer School

Alejandro Serrano

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

The menu for this afternoon

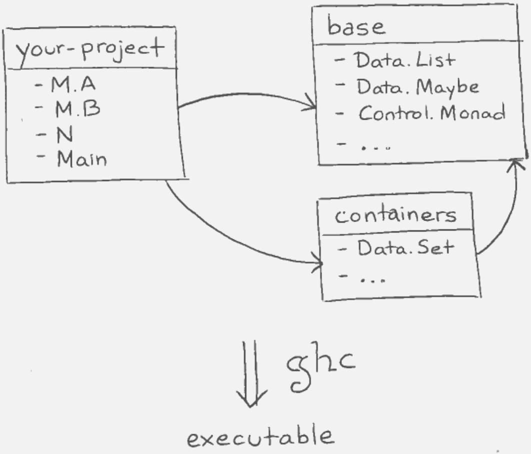
- How to build a larger Haskell project?
  - Dependencies, documentation…
- How to test that the project works correctly?
  - QuickCheck

The menu for this afternoon

- How to build a larger Haskell project?
  - Dependencies, documentation…
- How to test that the project works correctly?
  - QuickCheck
- **CODING AND FUN!!!**
  - And pizza

Project Management

Packages and modules



Packages and modules

- **Packages** are the unit of distibution of code.
  - You can *depend* on them.
  - Hackage is a repository of freely available packages.
- Each packages provides one or more **modules**.
  - Modules provide namespacing to Haskell.
  - Each module declares which functions, data types and type classes it *exports*.
  - You use elements from other modules by *importing*.
- In the presence of packages, an identifier is **no** longer **uniquely determined** by module + name, but additionally needs package name + version.

The GHC package manager

- The GHC package manager is called `ghc-pkg`.
- The set of packages GHC knows about is stored in a package configuration database, `package.conf`.
- Multiple package configuration databases:
  - one global per installation of GHC
  - one local per user
  - one per sandboxed project
  - more local databases for special purposes

Listing known packages

```
$ ghc-pkg list
/usr/lib/ghc-6.8.2/package.conf:
Cabal-1.2.3.0, GLUT-2.1.1.1, HDBC-1.1.3,
HUnit-1.2.0.0, OpenGL-2.2.1.1, QuickCheck-1.1.0.0,
array-0.1.0.0, base-3.0.1.0, binary-0.4.1,
cairo-0.9.12.1, containers-0.1.0.1, cpphs-1.5,
fgl-5.4.1.1, filepath-1.1.0.0, gconf-0.9.12.1,
(ghc-6.8.2), glade-0.9.12.1, glib-0.9.12.1,
...
/home/wouter/.ghc/i386-linux-6.8.2/package.conf:
binary-0.4.1, vty-3.0.0, zlib-0.4.0.2
```

- Parenthesized packages are hidden
- Exposed packages are usually available automatically.

The GHC package manager

Golden rule: you only use `ghc-pkg` to solve problems with your installation.

```
$ ghc-pkg check
% Empty or only warnings means
% package database in good shape
```

You use Cabal (or Stack) to manipulate the database.

Cabal: a Haskell package manager

- A unified package description format.
- A build system for Haskell applications and libraries, which is easy to use.
  - Tracks dependencies between Haskell packages.
  - Platform-independent, compiler-independent.
  - Generic support for preprocessors, inter-module dependencies, etc.
- Specifically tailored to the needs of a "normal" package.
- Integrated into the set of packages shipped with GHC.

Cabal is under *active* development, but very *stable*.

Hackage

Online Cabal package database.

- Everybody can upload their Cabal-based packages.
- Automated building of packages.
- Allows automatic online access to Haddock documentation.

`http://hackage.haskell.org/`

Project in the filesystem

```
your-project ................................ root folder
├── your-project.cabal.......info about dependencies
└── src ............................source files live here
    ├── M
    │   ├── A.hs ......................... defines module M.A
    │   └── B.hs ......................... defines module M.B
    ├── M.hs.............................defines module M
    └── N.hs.............................defines module N
```

- The project file – ending in `.cabal` – usually matches the name of the folder.
- The name of a module *matches* its place.
  - `A.B.C` lives in `src/A/B/C.hs`.

Initializing a project

1. Create a folder `your-project`.

```
$ mkdir your-project
$ cd your-project
```

2. Initialize the project file.

```
$ cabal init
Package name? [default: your-project]
...
What does the package build:
1) Library
2) Executable
Your choice? 2
...
```

Initializing a project

2. Initialize the project file (cntd.).

```
...
Source directory:
* 1) (none)
  2) src
  3) Other (specify)
Your choice? [default: (none)] 2
...
```

3. An empty project structure is created.

```
your-project
 ├── your-project.cabal
 └── src
```

The project (`.cabal`) file

```
-- General information about the package
name:    your-project
version: 0.1.0.0
author:  Alejandro Serrano
...

-- How to build an executable (program)
executable your-project
  main-is:         Main.hs
  hs-source-dirs: src
  build-depends:  base
  ...
```

Dependencies

Dependencies are declared in the `build-depends` field of a Cabal stanza such as `executable`.

- Just a comma-separated list of packages.
- Packages names as found in Hackage.
- Upper and lower bounds for version may be declared.
  - A change in the major version of a package usually involves a breakage in the library interface.

```
build-depends: base,
               transformers >= 0.5 && < 1.0
```

Executables

In an `executable` stanza you have a `main-is` field.

- Tells which file is the *entry point* of your program.

```
module Main where

import M.A
import M.B

main :: IO ()
main = -- Start running here
```

Building and running

0. Initialize a sandbox *only once*.

```
$ cabal sandbox init
```

1. Install the dependencies.

```
$ cabal update  # Obtain package information
$ cabal install --only-dependencies
```

   - Not needed if you use `cabal build`.

2. Compile and link the code.

```
$ cabal build
```

3. Run the executable.

```
$ cabal run your-project
```

Stack and Stackage

Besides cabal, there is a another package manager, *Stack*.

- Unlike Cabal, Stack manages your GHC installation.
- Uses sandboxes and local databased by default.

Stack uses *Stackage* instead of Hackage.

- Curated set of packages.
- Pro: installation plan always succeeds.
- Con: package versions lag behind Hackage.

Right now, both tools work flawlessly for normal usage.

- There are string advocates of both tools.

Using Stack

1. Create a new project.

```
$ stack new your-project && cd your-project
```

   - If you already have a Cabal file

```
$ cd your-project && stack init
```

2. Initialize the project *only once*.

   - Downloads all needed tools, including GHC.

```
$ stack setup
```

3. Compile and link the code.

```
$ stack build
```

4. Run the executable.

```
$ stack exec your-project
```

Other useful tools

`-Wall` is your friend

GHC includes a lot of warnings for suspicious code.

- Unused bindings or type variables.
- Incomplete pattern matching.
- Instance declaration without the minimal methods.

Enable this option in your Cabal stanzas.

```
library
   build-depends:  base, transformers, ...
   ghc-options:    -Wall
   ...
```

HLint

- A simple tool to improve your Haskell style.
- Developed by Neil Mitchell.
- Scans source code, provides suggestions.
- Makes use of generic programming (Uniplate).
- Suggests only correct transformations.
- New suggestions can be added, and some suggestions can be selectively disabled.
- Easy to install (via `cabal install hlint`).

HLint, simple example

Run it with `hlint path/to/your/source`.

- Source might be a file or a full folder.

```
Found:
  and (map even xs)
Why not:
  all even xs
```

HLint, larger example

```
i = (3) + 4
nm_With_Underscore = i

y = foldr (:) [] (map (+1) [3,4])

z = \x -> 5
p = \x y -> y
```

- What does HLint complain about, why?
- Would you always want such complaints?

HLint

All hints

- Error: Redundant bracket (1)
- Error: Redundant lambda (2)
- Warning: Use . (1)
- Warning: Use camelCase (1)
- Warning: Use const (1)

All files

- HLintDemo.hs (6)

Report generated by HLint v1.8.49 - a tool to suggest improvements to your Haskell code.

HLintDemo.hs:3:5: Error: Redundant bracket
Found
```
(3)
```
Why not
```
3
```

HLintDemo.hs:4:1: Warning: Use camelCase
Found
```
nm_With_Underscore = ...
```
Why not
```
nmWithUnderscore = ...
```

HLintDemo.hs:6:5: Warning: Use .
Found
```
foldr (:) [] (map (+ 1) [3, 4])
```
Why not
```
foldr ((:) . (+ 1)) [] [3, 4]
```

HLintDemo.hs:8:1: Error: Redundant lambda
Found
```
z = \ x -> 5
```
Why not
```
z x = 5
```

HLintDemo.hs:8:5: Warning: Use const
Found
```
\ x -> 5
```
Why not
```
const 5
```

HLintDemo.hs:9:1: Error: Redundant lambda
Found
```
p = \ x y -> y
```
Why not
```
p x y = y
```

Haddock

Haddock is the standard tool for documenting Haskell modules.

- Think of the Javadoc, RDoc, Sphinx… of Haskell.
- All Hackage documentation is produced by Haddock.

Haddock uses comments starting with | or ^.

```
-- | Obtains the first element.
head :: [a] -> a

tail :: [a] -> [a]
-- ^ Obtains all elements but the first one.
```

Haddock, larger example

```haskell
-- | 'filter', applied to a predicate and a list,
--   returns the list of those elements that
--   /satisfy/ the predicate.
filter :: (a -> Bool)  -- ^ Predicate over 'a'
       -> [a]          -- ^ List to be filtered
       -> [a]
```

- Single quotes as in `'filter'` indicate the name of a Haskell function, and cause automatic hyperlinking. Referring to qualified names is also possible (even if the identifier is not normally in scope).
- Emphasis with forward slashes: `/satisfy/`.

More markup

Haddock supports several more forms of markup:

- Sectioning to structure a module.
- Code blocks in documentation.
- References to whole modules.
- Itemized, enumerated, and definition lists.
- Hyperlinks.

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'
```


Universiteit Utrecht

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

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)`.

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

```haskell
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 ()
```

Which properties are `Testable`?

```
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:

```haskell
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.

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.

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.