Haskell Program Coverage Toolkit

Andy Gill

Colin Runciman

galois

Haskell as an Implementation Language

How do you debug and test Haskell programs?

- The program compiles!
  - Type checking does catch many, many errors, but clearly not all
- Run program over a handful of inputs, examine output
  - Incremental testing; other features can break
- Write unit tests
  - In Haskell, we can use a specification style for function level unit tests

```
prop_goodSort1 xs = isSorted (sort xs) == True
prop_goodSort2 xs = length xs == length (sort xs)
prop_goodSort3 xs = all [ x `isMember` sort xs | x <- xs ]
…
```

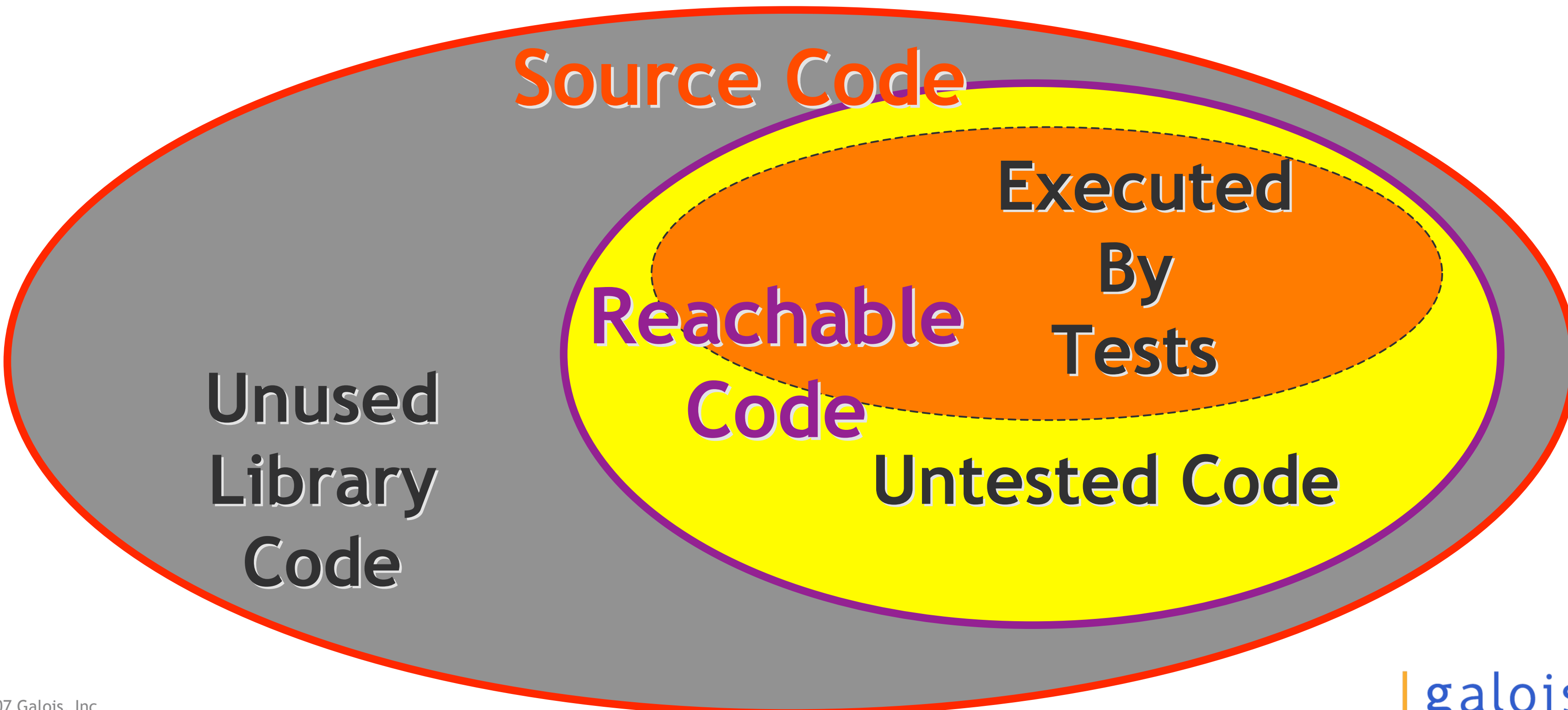
- How much testing is enough?
- **What assessments about test quality can we make automatically?**

We are going to use code coverage to help make assessments…

|galois|

Why Study Code Coverage?

If your program contains reachable code that has not been executed by your tests, then your program is insufficiently tested. <u>This reachable, unexecuted code could do **anything.**</u>



Principal Classes of Code Coverage

- Function Based Coverage
  - List of functions (or classes) that are never executed
  - Coarse grain functionality
- Decision Coverage
  - What branches have been taken?
  - What boolean expressions inside control structures were always true or always false?

```
if (x == 4) {
  y = 5;
}
```

- Line (or Statement) Coverage
  - What lines have never been executed?
  - Typically displayed as color listings
- Path Coverage
  - Capturing combinations of assignments and control flow

Mapping Traditional Code Coverage to Haskell Code Coverage

Coverage Class	Traditional Code Coverage	Haskell Code Coverage
Function	Yes	Yes
Decision	Conditionals	Conditionals, Guards, Qualifiers
	Switch	Case, Pattern Matching
Line	Yes	?
Path	Yes*	(Sub) Expressions

* Only found in high-end coverage tools

| galois |

Classes of Haskell Code Coverage

- Function Based Coverage
  - List of functions that are never evaluated
  - Course grain functionality
- Alternative Coverage
  - How many alternatives were never evaluated?
- Control Boolean Coverage
  - What boolean expressions inside control structures were always true or always false?
- Expression Level coverage
  - What expression has never been evaluated?
  - Critical for complete coverage of non-strict language
  - Comparable to path coverage in traditional coverage tools

Example Haskell Program

```
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
             | otherwise = error
                  "attempting to compute reciprocal of number <= 1"
  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
```


galois

Reciprocal Program Executing

```
$ reciprocal
2
1/2 = 0.5
3
1/3 = 0.(3)
4
1/4 = 0.25
```

Example Markup from Haskell Coverage Tool

```haskell
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
             | otherwise = error
                 "attempting to compute reciprocal of number <= 1"
  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
```

Example Output from Textual Report

```
$ hpc-report a.out
-----<module Main>-----
 90% expressions used (88/97)
 37% boolean coverage (3/8)
      28% guards (2/7),
          3 always True, 2 unevaluated
     100% 'if' conditions (1/1)
     100% qualifiers (0/0)
 77% alternatives used (7/9)
100% local declarations used (1/1)
100% top-level declarations used (5/5)
```

Reciprocal program executing new cases

```
$ reciprocal
1
reciprocal: attempting to compute
   reciprocal of number <= 1
$ reciprocal
33
1/33 = 0.(03)
```

Example of 100% Coverage

```
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
             | otherwise = error
                  "attempting to compute reciprocal of number <= 1"
  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
```


galois

Instrumented code

- Ticks are added to each “interesting” sub-expression

```
f 99 (g n)   -->   tick 1 (f (tick 2 99) (tick 3 (g (tick 4 n))))
```

- Ticks
  - are numbered
  - are omitted on obviously strict sub-expression
  - work by benign side-effect

The Haskell Program Coverage Toolkit

- Hpc consists of
  - Compiler option inside the Glasgow Haskell compiler
  - Command line tools for processing coverage data
- Hpc can output code and summary tables for viewing in any browser
  - Intermediate formats are simple and open
  - Other tools can use the coverage data
- Scales to large Haskell programs
  - Handles Haskell programs with 100s of modules and 100k+ lines of code
  - Can interoperate with pre-compiled libraries
  - Runtime overhead of around 2
- We are now going to quickly survey three Hpc features that go beyond simple code markup
  - Dashboard of coverage
  - DSL for coverage exclusions
  - Dynamic Code Coverage

Haskell Program Coverage Dashboard

module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module CSG	-	0/0		-	0/0		-	0/0	
module Construct	100%	25/25		100%	12/12		100%	569/569	
module Data	91%	22/24		60%	24/40		90%	527/585	
module Eval	95%	19/20		93%	59/63		96%	541/561	
module Geometry	100%	45/45		60%	6/10		95%	335/351	
module Illumination	100%	15/15		72%	26/36		96%	415/428	
module Intersections	100%	22/22		81%	68/83		87%	879/1001	
module Interval	70%	12/17		73%	17/23		78%	129/165	
module Main	100%	1/1		-	0/0		100%	5/5	
module Misc	100%	1/1		-	0/0		100%	10/10	
module Parse	100%	17/17		100%	8/8		100%	234/234	
module Primitives	33%	2/6		-	0/0		41%	10/24	
module Surface	55%	5/9		85%	17/20		92%	205/221	
Program Coverage Total	92%	186/202		80%	237/295		92%	3859/4154	

Why is code not executed?

- Dead Code (unreachable from main)
  - If in a core module, should be removed
  - If in a library, not using a specific function is completely reasonable
- Asserts, Preconditions and Impossible cases
  - Asserts catch cases that we consider impossible to ever happen (inconsistent data, bad precondition, etc)
  - Should be impossible to reach this code!
- Token values
  - () : the empty tuple is a type of token we use in Haskell
- Code specifically for testing code not executed in a system binary
  - A type of dead code
  - Perhaps reachable through BIST

Hpc includes a script for specifying exclusions

```
tick every expression "()"                          [idiom];

module "Parse" {
  tick function "test_number"                       [testing];
  function "rayParse" {
    tick expression “\“error (show err)\””          [impossible];
  }
}
…
```

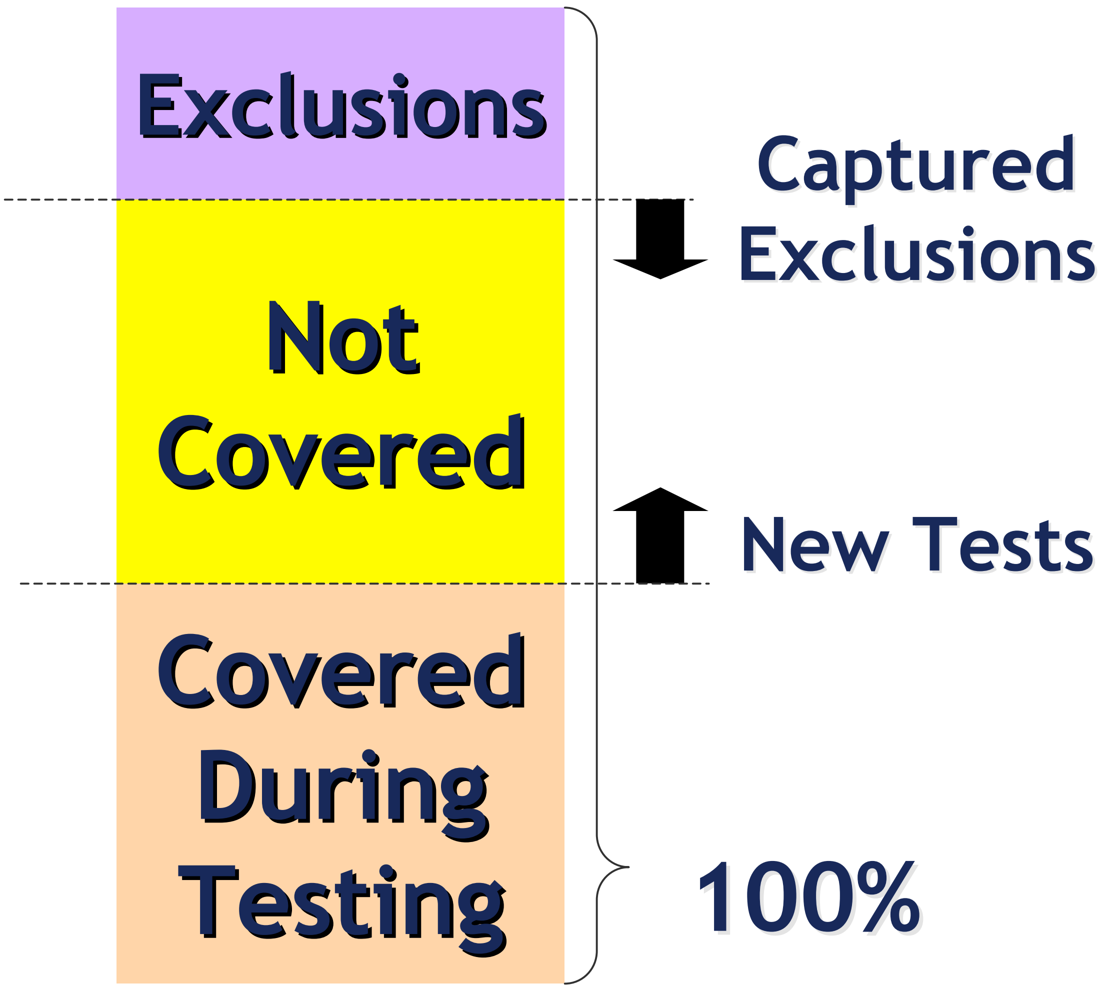
We call these a coverage overlays

- They overlay coverage *found via execution* with information about reasonable gaps in coverage *found by inspection*.

Methodology for Reaching 100% Coverage

Two directions of coverage improvement

- Add new tests
- Add exclusions to overlay



The captured exclusions specify what a human reviewer has considered reasonable never to reach

Hpc also includes a tool which automatically generates a first draft of this list of exclusions

Dynamic Code Coverage

Simple Idea - Give the programmer the ability to read and write to the current state of the code coverage counters - at **runtime**.

- When testing (using QuickCheck) we can separate coverage into successful and unsuccessful coverage buckets, in a single run.
  - Prototyped inside QuickCheck2
  - Taking the difference between successful and unsuccessful coverage automatically finds the code uniquely executed by unsuccessful QuickCheck properties.
  - Automated “heads-up” for finding bugs.
- We can observe coverage in a running program over a single external event or transaction
  - For example what code do we use to handle serving a web page?
- … and many others …

Summary

- Haskell has high-fidelity coverage information tools
  - Human overhead is nominal (add a single compiler flag)
- Toolkit gives state-of-the-art coverage information
  - Covered code is marked up in HTML with dashboard to help navigation
  - Coverage can combine multiple binaries that share code
  - Includes scripting language for specifying exceptions
- Hpc is useful to Galois and the wider Haskell community
  - Possible to demonstrate coverage on real code
  - Found bugs in existing code (typically missing preconditions)
  - There are grass root plans to use Hpc on core Haskell libraries before next GHC release cycle
- Technology reusable for other Haskell projects
  - Hpc adressed the problem of mapping source locations to locations inside an executable binary - the new Haskell debugger uses the Hpc solution to allow expression-level debugging
  - There is a dynamic tracer based on coverage ordering, also with accurate source locations available
  - Plans for profile based optimizations and profile based deforestation