

Haskell Program Coverage Toolkit

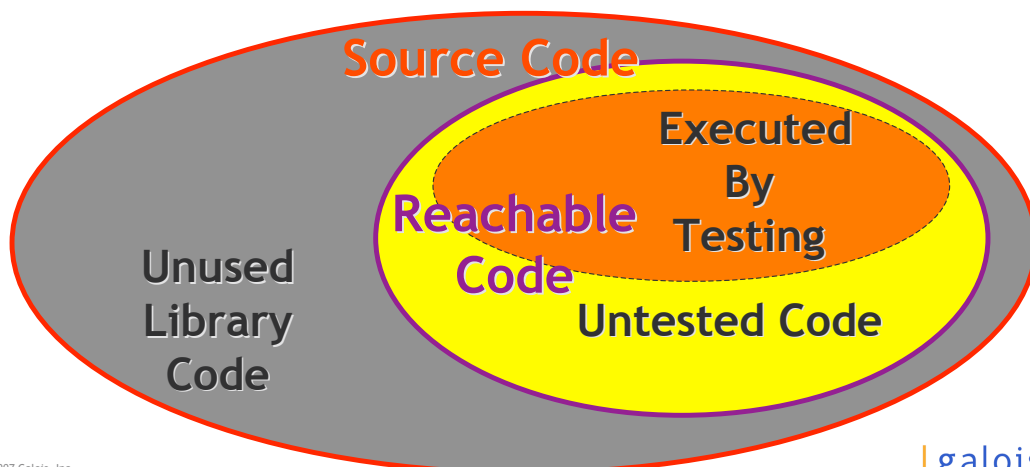
Andy Gill
Colin Runciman



© 2007 Galois, Inc.

Why Study Code Coverage?

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



© 2007 Galois, Inc.



Implementing Applications in Haskell

How do you debug and test Haskell programs?

- Run system tests
 - Try typical inputs, compare results
 - Remember to test the automatic help...
 - Remember to test the boundary cases...
- Run QuickCheck and QuickCheck style 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...

© 2007 Galois, Inc.

| galois |

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

© 2007 Galois, Inc.

| galois |

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

© 2007 Galois, Inc.

| 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

© 2007 Galois, Inc.

| galois |

Hpc Demo

- ...

© 2007 Galois, Inc.

| galois |

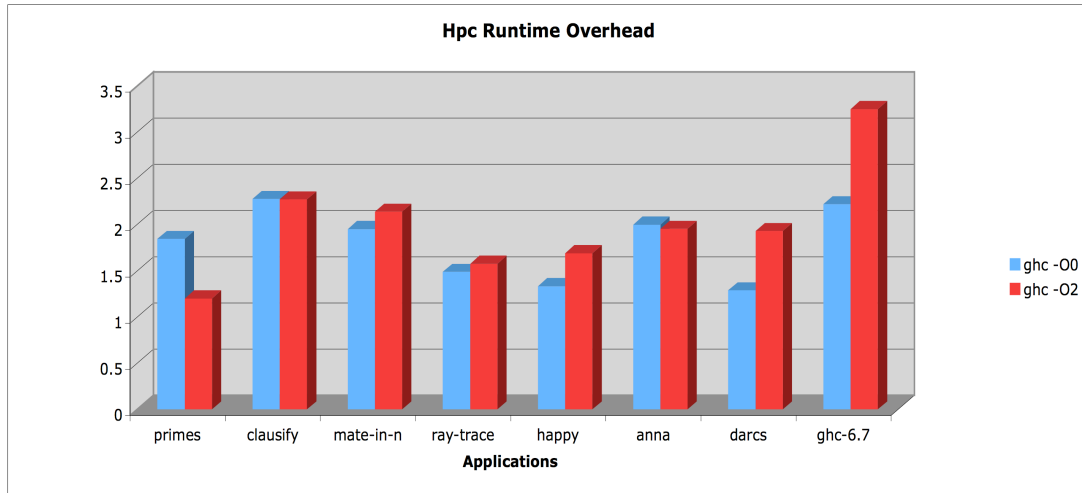
The Haskell Program Coverage Toolkit

- Hpc consists of
 - A compiler option inside the Glasgow Haskell compiler (OR a Haskell to Haskell translator)
 - Command line tools for processing coverage data
- Hpc can give the programmer
 - Marked-up code and summary tables for viewing in any browser
 - ASCII and/or XML code coverage totals
- Hpc can be extended
 - File based 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 seamlessly

© 2007 Galois, Inc.

| galois |

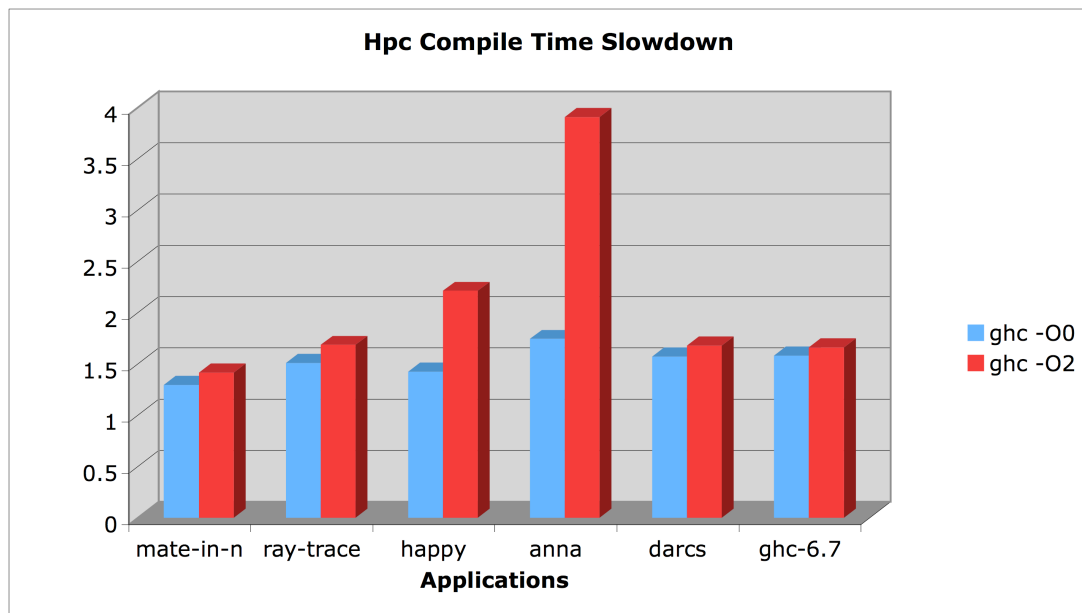
What is the runtime overhead?



© 2007 Galois, Inc.

| galois |

What is the compile time overhead?



© 2007 Galois, Inc.

| galois |

How does Hpc work?

- 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-expressions
 - Work by benign side-effect
- Inside GHC, ‘tick’ is an extension to our AST
 - This tick has span information from parsing
 - Record mapping from tick <n> to location line:col-line:col for each tick
- These ticks live as special case statements in GHC Core

```
case tick 1 of
  _ -> f (case tick 2 of { ..}) (case tick 3 of { ... })
```
- At the very back end, we generate 64-bit increment for each tick entered

© 2007 Galois, Inc.

| galois |

Practical experiences from using Hpc

- Be mindful of the two basic ways of invoking programs
 - As applications
 - From inside unit testsWhere do these two intersect, where do they overlap?
- Hpc can spot when QuickCheck properties are not actual deep enough
 - Can not cheat with inadequate QuickCheck properties
- Panoramic view of your program
 - I see dead code...
- Encouraging refactoring
 - If I refactor this function to use this other function, I only need to write tests for one function!
 - This table has some “holes”, perhaps I should use a different type?
- Makes you think about the failure/failing cases
 - QuickCheck lets you think about Successes
 - Hpc reminds you of your Failures!
- It is just hard to get to 100% code coverage

© 2007 Galois, Inc.

| galois |

Why is some 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

© 2007 Galois, Inc.

| galois |

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*.

© 2007 Galois, Inc.

| galois |

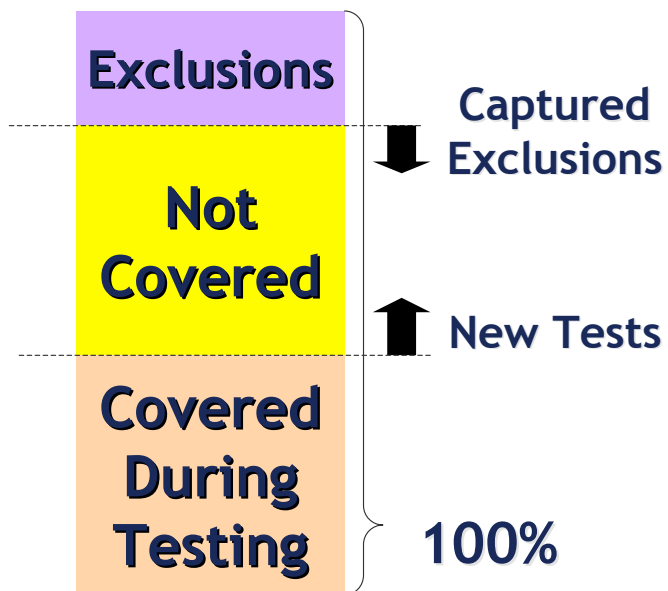
Methodology for Reaching 100% Coverage

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



© 2007 Galois, Inc.

| galois |

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 small bugs in existing code (typically missing preconditions)
 - Xmonad will shortly require 100% coverage before commits
- Technology reusable for other Haskell projects
 - Hpc addressed 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
 - Source level entry counts available to other tools and optimizers

© 2007 Galois, Inc.

| galois |