Haskell Program Coverage

Andy Gill
Galois, Inc.
andy@galois.com

Colin Runciman
University of York
colin@cs.york.ac.uk

Abstract

We describe the design, implementation and use of HPC, a toolkit to record and display Haskell Program Coverage. HPC includes tools that instrument Haskell programs to record program coverage, run instrumented programs, and display information derived from coverage data in various ways.

Categories and Subject Descriptors D.2.5 [*SOFTWARE ENGINEERING*]: Testing and Debugging—Testing tools (e.g., data generators, coverage testing)

General Terms Measurement

Keywords Haskell, Software Engineering, Code Coverage

1. Introduction

Computer programs need thorough testing. To be thorough may mean many things, but surely this is one of them: testing should cover every reachable part of the program. To meet this coverage requirement, we must first establish exactly what should be regarded as program parts. Then as we test our programs we need convenient ways to record and review which parts have been covered and which have not.

In conventional procedural languages, the parts of a program for the purposes of test coverage are often taken to be of two kinds [17, 15]. First there are the atomic commands or statements. Second there are the branches or paths by which control passes between these statements. In large programs, by abstraction, the procedures exported by each module or package might be treated as the atomic commands, and calls from one module to another as the control-passing paths.

In a non-strict purely functional language like Haskell, conventional commands such as destructive re-assignment are not present, and flow of control is rarely explicit. Functional programs are composed from expressions, not commands, and the natural units of abstraction are functions, not procedures. The flow of control between expressions is determined by the rules of lazy evaluation. A function may be called many times yet have subexpressions in its body that are never evaluated.

This paper describes the design, implementation and deployment of HPC, a code coverage tool for Haskell that embraces Haskell's lazy evaluation, and offers fine grain, expression level coverage analysis. Section 2 sets out our design goals and requirements for HPC. Section 3 explains how we present the coverage information we have collected. Section 4 explains how HPC instruments code to collect coverage. Section 5 explains how HPC implements instrumentation. Section 6 describes our experience with applications for HPC. Section 7 evaluates HPC from various perspectives. Section 8 briefly discusses other code coverage tools. Section 9 presents conclusions and ideas for further work.

Haskell'07, September 30, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-674-5/07/0009...$5.00

2. Design Goals and Requirements for Hpc

Our design decisions for HPC were informed by our previous experiences of implementing tools such as Hood [7] and Hat [14, 11] which also record and display traces of computational events. We also were taking into account Galois' Haskell development and quality assurance needs.

Our main design decisions for HPC, and some of our reasons for them, were as follows:

Scalability HPC works on large Haskell applications, including Haskell programs with complex build systems. At Galois, we have used HPC on several large Haskell applications, and HPC has been used to examine coverage for GHC [6], itself a large Haskell program.

Ease of Use HPC coverage instrumentation can be introduced using a single flag in GHC, or in a portable way using a HPC script that uses hmake.

Granularity HPC records coverage in relation to both small-scale and large-scale components of programs. Traditional line-based coverage is inadequate in lazy functional languages where every expression is evaluated by need. Also, it is necessary to record at the level of individual data literals, so that we can establish coverage for table-driven applications. Unlike any other coverage tool that we know of, HPC detects any expression, no matter how small, that is never used in a recorded run of a program. But if HPC is to be used for large applications, coverage information attached to larger units seems essential. So HPC also records and summarizes coverage information at the level of declarations and modules. Finally, HPC also records coverage of some control paths that cannot be identified with use of an expression.

Portability HPC is portable between Haskell implementations. Portable coverage instrumentation can be introduced by a source-to-source transformation and the run-time library is simple. This approach also has the advantage of making the meaning of coverage records open to inspection.

Language Extension Support We have also found it useful to develop a specialized variant of HPC integrated with the GHC compiler and runtime system, allowing every GHC language-extension and other features of the compiler to be supported by HPC.

Accumulation Not all coverage questions can be answered in relation to a single run of a single executable copy of a program.

HPC coverage records are cumulative, representing zero or more runs of a program. Records from different installations or users of the same program can also be combined. So, for example, program testing can be distributed without losing an overall view of the coverage obtained.

Openness HPC uses a simple, open and documented set of file formats for storing coverage information. The two major data structures are an indexed list of integers, and a mapping from index positions to source locations and associated information. Although the design of these formats has been driven by the needs of the HPC toolkit, the information is not linked in any obscure way to HPC internals. Other developers could write new tools using these formats, optionally supported by small library modules from HPC.

Costs The costs of using a tool such as HPC can be divided into those incurred at compile-time (including source-to-source instrumentation) at run-time and when post-processing recorded information for presentation to a user. In HPC run-time costs are minimized by restricting the run-time representation of the coverage record to a numerically-indexed array of entry counters which we call *tick boxes*. Our aim is to make the use of these tick boxes both cheap and conceptually simple. The only overheads are for operations to increment a tick-box at a given position and to update the tick-box file associated with the program. The costs of assigning and interpreting meanings of tick boxes are paid at instrumentation time and by the tools that present coverage information, not at run time.

Results Although sources are the most direct medium of thought for programmers, statistical summaries are more concise for large programs, and more amenable to comparison and processing. HPC therefore presents coverage information in two forms: highlighting of sources and summary statistics.

Non-annotation Any tool whose use requires programs first to be modified or annotated imposes upon its users additional work. This cost would be unacceptable for larger applications. It is best if such requirements are minimized. HPC does not require the programmer to annotate, or otherwise alter, the source code in *any* way.

Selectivity It may often be the case that coverage information is only needed or appropriate for particular program components. Other components may, for example, be standard and already well-tested. Or perhaps they are unavailable in source form. Or else some components may use a non-standard extension for which no instrumentation rules are available in HPC. Or else again, for larger and more demanding applications, the overheads of generating coverage information may simply not be affordable for the whole program. For all these reasons we have been careful in the design of HPC to ensure that it can be applied to selected modules in a program, with *zero overhead* for modules not selected.

Entry Counts Though the HPC coverage reporting tools consider the entry count information in a boolean fashion, we record entry *counts* internally rather than just the boolean entered or not entered for each sub-expression. The extra overhead of counting rather than ticking is small (see Section 7); accurate entry counting gives is a poor-mans profiler for free; and we anticipated using such information in the future for profile base optimizations. We use the nomenclature 'ticking' in this paper to refer to the incrementing of a tick count box.

Design and Implementation Plan

These criteria lead us to the following simple design and implementation plan.

- We initially use a Haskell to Haskell rewriter that instruments code to record what parts of every expression and sub-expression are entered as a benign, type-preserving side-effect.
- We record to a file the source code location associated with every tick-box introduced — we call this a *.mix* file (Module IndeX file).
- We also record for each module the total number of tick boxes introduced so that local box numbers can be mapped to global ones — this information goes in a *.pix* file (Program IndeX file).
- Each side-effecting tick increments a 64-bit natural — large enough that it will not overflow under any reasonable use— and we have distinct 64-bit naturals for each expression (and sub-expression).
- The recorded ticks are stored at the end of each program run in a file called the *.tix* file (TIcks file, sounds like ticks).
- We use post-processing command line tools to present the content of .tix files in human readable formats – both summary tables and marked up source code.
- After developing these basics, we add an hpc option to GHC, implementing our side-effects inside the GHC code generator for compile-time and run-time efficiency.

3. Observing Program Coverage

HPC provides coverage information of two kinds: source coverage and boolean-control coverage. Source coverage is the extent to which every part of the program was used, measured at three different levels: declarations (both top-level and local), alternatives (among several equations or case branches) and expressions (at every level). Boolean coverage is the extent to which each of the values True and False is obtained in every syntactic boolean context (i.e. guard, condition, qualifier).

Hpc displays both kinds of information in two different ways: textual reports with summary statistics (hpc-report) and sources with color mark-up (hpc-markup). For boolean coverage, there are four possible outcomes for each guard, condition or qualifier: both True and False values occur; only True; only False; never evaluated. In hpc-markup output, highlighting with a yellow background indicates a part of the program that was never evaluated; a green background indicates an always-True expression and a red background indicates an always-False one.

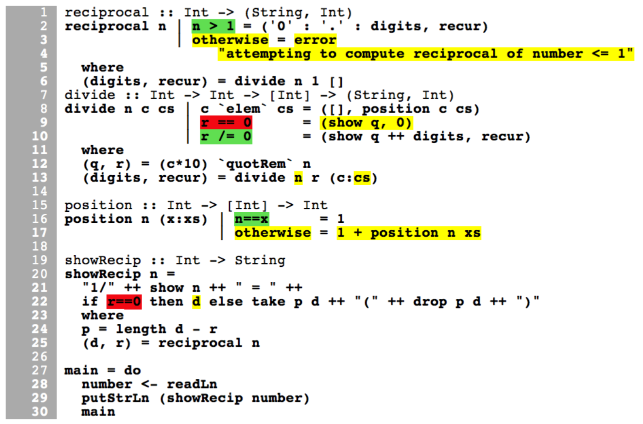
The programs hpc-report and hpc-markup both take as input the .mix and .tix files for an HPC-instrumented program. They then zip together each module-block of tick-boxes with the associated series of source positions and box-labels from the appropriate .mix file.

Hpc-report accumulates counts of ticked and unticked boxes in the various categories to provide summary statistics for each module. It also accumulates and reports for each module a list of names declared but never used.

Hpc-markup selects for each module the unticked boxes only. These are sorted according to their associated source-position — descending end positions within ascending start positions. Traversing the sorted list in tandem with the corresponding source, an HTML version of the source is generated; color mark-up is introduced to highlight unused fragments of the program and any control expressions for which boolean coverage is incomplete.

3.1 A Small Example: Reciprocation

For an example we have a program which computes exact decimal representations of reciprocals, with recurring parts indicated in brackets. We first build an instrumented version using the hpc-build script. In the following example, the file `Recip.hs` contains



```
 1 reciprocal :: Int -> (String, Int)
 2 reciprocal n | n > 1 = ('0' : '.' : digits, recur)
 3              | otherwise = error
 4                  "attempting to compute reciprocal of number <= 1"
 5   where
 6   (digits, recur) = divide n 1 []
 7 divide :: Int -> Int -> [Int] -> (String, Int)
 8 divide n c cs | c `elem` cs = ([], position c cs)
 9               | r == 0      = (show q, 0)
10               | r /= 0      = (show q ++ digits, recur)
11   where
12   (q, r) = (c*10) `quotRem` n
13   (digits, recur) = divide n r (c:cs)
14
15 position :: Int -> [Int] -> Int
16 position n (x:xs) | n==x      = 1
17                   | otherwise = 1 + position n xs
18
19 showRecip :: Int -> String
20 showRecip n =
21   "1/" ++ show n ++ " = " ++
22   if r==0 then d else take p d ++ "(" ++ drop p d ++ ")"
23   where
24   p = length d - r
25   (d, r) = reciprocal n
26
27 main = do
28   number <- readLn
29   putStrLn (showRecip number)
30   main
```

Figure 1. Example output from hpc-markup. Yellow-highlighted expressions were never evaluated, green-highlighted expressions were evaluated but always True, red-highlighted expressions were evaluated but always False.

a Haskell program to re-express reciprocals as possibly recurring decimals:

```
$ hpc-build Recip
transforming Recip.hs into ./.hpc/Recip.hs
ghc ...<lots of flags>... Recip.hs
```

Instrumented sources and HPC module index files are place in the subdirectory **.hpc**.

```
$ ./Recip
1/3
= 0.(3)
```

The execution of the instrumented binary deposits a **.tix** file into the current working directory.

To obtain a textual summary of coverage we run:

```
$ hpc-report Recip
 80% expressions used (81/101)
 12% boolean coverage (1/8)
      14% guards (1/7), 3 always True,
                        1 always False,
                        2 unevaluated
       0% 'if' conditions (0/1), 1 always False
     100% qualifiers (0/0)
 55% alternatives used (5/9)
100% local declarations used (9/9)
100% top-level declarations used (5/5)
```

Finally, we generate a marked-up version of the source.

```
$ hpc-markup Recip
writing Recip.hs.html
```

If we use an HTML browser to view Recip.hs.html we see something like Figure 1.

hpc-markup also generates a summary dashboard for each Haskell application that it provided markup for. Figure 2 shows the summary dashboard for a small chess problem solving program – with 5 modules – a more interesting example than our single module Recip program.

3.2 Combining Multiple Coverage Records

Each run of an HPC-instrumented program will result in the generation of a .tix file in the current working directory. Sometimes it is useful to common up these distinct runs into a single, summary .tix file. So we provide a combining utility, hpc-combine, which reads multiple .tix files, writing a new .tix file.

The default use of hpc-combine is to sum the entry counters in the input .tix files, but there are other possibilities. The hpc-combine tool is our swiss-army knife on .tix files — we use it to process and plumb between our testing framework running instrumented code and our coverage reporting tools. Inside hpc-combine, we also provide

- The ability to take the difference between two .tix files; we shall see examples of where this is useful in Section 6.
- The ability to select specific modules, which is useful when we want to see coverage for a specific component, or set of modules, in isolation.

In every case, hpc-combine writes a new file, it does not alter the original .tix output created by instrumentation.

3.3 A DSL for Coverage Exclusions

We consider .tix files to be first class. We can generate them by running instrumented code, merge them using hpc-combine, and render them into summaries and marked up code using hpc-report and hpc-markup.

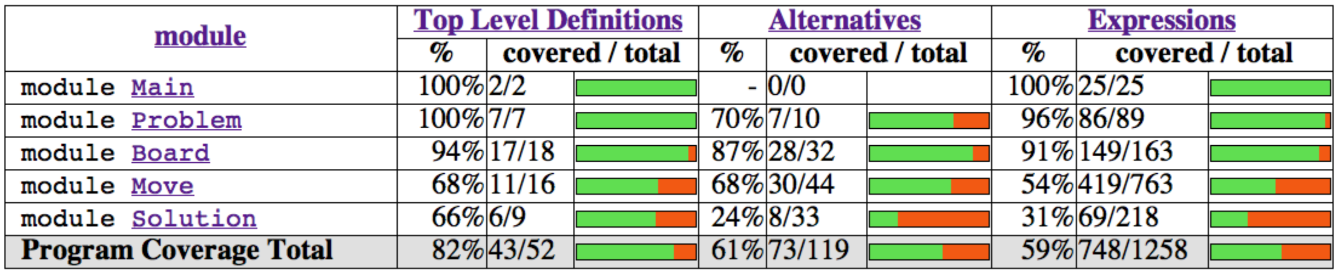
It is also useful to generate a .tix file from a human-readable specification, rather than an instrumented program. The idea is to give programmers a way to record *exclusions* when assessing the coverage obtain by running a program. The programmer can say: "I know these components may never be reached in the program, but that is for a good reason, so consider them covered".

These readable specifications of ticks to be recorded are written in a small domain-specific language, and put in files with an .mtix extension (for manual ticks, or meta-ticks). We provide two tools for working with .mtix files. The first, hpc-makemtix, generates an .mtix file that specifies ticks exactly where they are *not* recorded in a given .tix file. Here is an example, based once again on the Recip program:

```
$ hpc-makemtix Recip.tix
module "Recip" {
  function "reciprocal" {
    tick expression "otherwise" on line 3;
    tick expression "error ...
  }
  function "divide" {
    tick expression "(show q, 0)" on line 9;
    tick expression "n" on line 13;
    tick expression "cs" on line 13;
  }
  function "position" {
    tick expression "otherwise" on line 17;
    tick expression "1 + position n xs" on line 17;
  }
  function "showRecip" {
    tick expression "d" on line 22;
  }
}
```

The programmer can read this specification as a list of coverage gaps, and an invitation to decide in each case whether it should be excluded from consideration. They can edit such a specification deleting only the items representing a genuine gap in expected program coverage. What remains is a specification of items to be treated *as if* they were covered.

Another tool, hpc-maketix, works in the opposite direction. Given a .mtix specification it generates a .tix file with a tick recorded for every specified item *and all its subexpressions*.



module	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
module Main	100%	2/2	🟩	-	0/0		100%	25/25	🟩
module Problem	100%	7/7	🟩	70%	7/10	🟩🟥	96%	86/89	🟩🟥
module Board	94%	17/18	🟩🟥	87%	28/32	🟩🟥	91%	149/163	🟩🟥
module Move	68%	11/16	🟩🟥	68%	30/44	🟩🟥	54%	419/763	🟩🟥
module Solution	66%	6/9	🟩🟥	24%	8/33	🟩🟥	31%	69/218	🟩🟥
Program Coverage Total	82%	43/52	🟩🟥	61%	73/119	🟩🟥	59%	748/1258	🟩🟥

Figure 2. Example Summary Dashboard from hpc-markup for small application

So hpc-makemtix and hpc-maketix are not exact inverses of each other; but they are complementary in the following sense. Suppose prog.tix is the coverage record from one or more runs of a program and we perform the following commands.

```
$ hpc-makemtix prog.tix | hpc-maketix > non/prog.tix
$ hpc-combine prog.tix non/prog.tix > sum/prog.tix
```

Now sum/prog.tix records 100% coverage.

The DSL for tick specifications is a little more powerful than the example output from hpc-makemtix illustrates. For example, it is not necessary to give exact line numbers, and the language allows for regular expression matching as well as string matching.

This completes our tour of the HPC post-processing tools. In the next section we discuss how we instrument Haskell code to produce coverage information in the first place.

4. Instrumentation: How HPC Works

In the transformed source each significant expression of the original program is associated with a uniquely numbered *tick-box*. On *entry* to the expression a tick is added to the relevant box by applying a function `tick :: Int -> a -> a`. The `Int` argument is the appropriate tick-box number, and the polymorphic argument is the expression being entered. The result of `tick n e` is just `e`, but with a tick recorded in box `n` as a benign side-effect.

As a small example, consider the expression

```
  f 99 (g n)
```

This might be translated as

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

An astute reader might have observed that we have not put a tick round *every* expression. We will return to the important rationale behind the missing ticks shortly.

Attaching tick-boxes to all value declarations is achieved by a few tricks adding extra equations. For example, a pattern-declaration

```
(d,r) = reciprocal n
```

is transformed along the following lines

```
v = tick 1 reciprocal (tick 2 n))
(d,_) = tick 3 v
(_,r) = tick 4 v
```

where `v` is fresh variable, so as to record use of `d` and `r` separately. We have been careful to ensure that all such pattern-related transformations preserve the strictness or laziness of the original patterns.

The transformation of a function declaration

```
null []    = True
null (_:_) = False
```

looks something like this

```
null _ | tick 1 False = undefined
null [] = tick 2 True
null (_:_) = tick 3 False
```

where tick-box number 1 records whether the null function is ever used.

4.1 Eliminating Redundant Tick Boxes

It is not necessary to attach a tick box to *every* expression in a program. When evaluation of an expression always strictly entails evaluation of a particular subexpression, a tick-box attached to the subexpression is redundant. For example, consider let binding:

```
 let v = <rhs> in <body>
```

Evaluation of the whole let-expression strictly entails evaluation of its body. (This would not be true in a language with strict bindings: evaluation of the right-hand-side of a binding might fail to terminate or throw an exception in which case the body is never evaluated.)

Eliminating tick-boxes for strict subexpressions is a critical optimization. The costs saved are far greater than might be expected. Consider the following expression:

```
 addInt x y
```

Suppose we introduce a tick-box for every sub-expression. Because addInt is curried, the instrumented expression is:

```
   tick 0 (tick 1 ((tick 2 addInt) (tick 3 x))
                  (tick 4 y)
          )
```

What happens when this instrumented expression is evaluated?

```
   tick 0 (tick 1 ((tick 2 addInt) (tick 3 x))
                  (tick 4 y)
          )
      ==>
   tick 1 ((tick 2 addInt) (tick 3 x)) (tick 4 y)
      ==>
   (tick 2 addInt (tick 3 x)) (tick 4 y)
```

The call to `addInt`, something we might expect to be a highly efficient primitive operation, has become a higher order call from inside our implementation of tick. Such arbitrary calls are already expensive, but the overhead gets worse. Let us continue evaluation:

```
     ==>
   (addInt (tick 3 x)) (tick 4 y)
```

The evaluation involves returning a partially applied `addInt`, for which space must be allocated on the heap! All this for what was a primitive addition operation.

Not placing tick-boxes on strict subexpressions rescues us from drastic performance loss in such cases. In the example above, evaluation of the `tick 0` application always strictly entails evaluation of `tick 1` and `tick 2`. In general, applications are strict in the function to be applied. So an expression such as

```
f (g x)
```

can be instrumented like this:

```
tick 1 (f (tick 2 (g (tick 3 x))))
```

This optimization enables HPC to give just as much coverage information, but with less box-ticking and less impact on the efficiency of compiled applications.

4.2 Syntactical Sugar

We add ticks round (almost) every expression, even if they are syntactic sugar. For example

```
[1..n]
```

gets translated into

```
tick 1 [(tick 2 1)..(tick 3 n)]
```

We also put ticks round monadic expressions, which allows us to capture the equivalent of line coverage when programming in the IO monad.

4.3 Literal Data Structures

Data structures are expressed by applications of constructors. These are transformed just like any other applicative structure, with a tick-box for every component at every level. This kind of coverage is very important for interpretive applications that do a lot of table-driven processing. All entries in the tables should be tested, and lists include a tick box for each element.

Although the literal character list `['H','e','l',...,'!']` is equivalent to the literal string `"Hello world!"` HPC records coverage differently – we attach only a single tick-box to a string literal. When we came to define the transformation of string literals, recording character-by-character coverage seemed excessive, and the atomicity of string coverage has not been a problem in practice. However, problems *could* be buried in the untested tails of strings, so we may revisit this design decision in the future, and provide more fidelity as a flag if requested.

We could take the requirement for fine-grained coverage to extremes. Following the shadow register bit-level approach of tools like Valgrind[13] would even reveal the usage of individual bits in an Int! But we choose to associate tick-boxes only with source-level expressions.

4.4 Boolean Control Coverage

Boolean expressions play a distinctive role in Haskell programs where they occur in one of three specific syntactic contexts. The `Bool` type is unique in this respect. It is useful, and quite straightforward, to exploit the basic tick-box machinery to check that both `True` and `False` outcomes occur for each of these controlling expressions. We do this by adding two ticks round the *result* of the boolean in question in such a way as to detect if the boolean is `True` or `False`.

```
if <boolean-expression>
then tick 1 True
else tick 2 False
```

We only add these extra ticks in places where boolean expressions appear in syntactically significant contexts: if-conditions, guards and qualifiers.

4.5 The Awkward Squad and Friends

HPC handles all the major extensions of Haskell beyond a traditional functional language without a problem.

Concurrent threads: The tick function is thread safe, so HPC can handle concurrent programs.

Exceptions: Box-ticking is robust in the presence of exceptional behavior. We first record *entry* into an expression, and only then evaluate it. Exceptions propagate just as usual.

Foreign function calls: Such calls work in instrumented programs without complication. Indeed, we have applied HPC to several large applications that make use of the FFI. We leave unchanged any FFI specifications when instrumenting programs, though we could record the use of a specific FFI call in the same way as we record other top level functions. Of course the functions that call the FFI functions are observed by HPC.

"Unsafe" IO: Applications of unsafePerformIO are completely compatible with HPC. (Some might regard this as a bug, not a feature!)

4.6 Module-index (.mix) Files

In addition to the instrumented source-code, the transformation stage also generates a *module-index file* recording details of the tick-box associated with each number. Module index entries have the following type.

```
data Mix
  = Mix
    FilePath   -- location of original file
    Integer    -- time (in seconds) of original file
    Hash       -- hash of mix entry + timestamp
    Int        -- tab stop value.
    [MixEntry] -- entries
  deriving (Show,Read)

type MixEntry = (SourcePosition, BoxLabel)

data BoxLabel = ExpBox  Bool -- isAlt
              | TopLevelBox [String]
              | LocalBox [String]
              | BinBox BoolCxt Bool
  deriving (Read, Show, Eq, Ord)

data BoolCxt = GuardBinBox
             | CondBinBox
             | QualBinBox
  deriving (Read, Show, Eq, Ord)
```

The `BoxLabel` values provide the information needed for the statistical summaries of `hpc-report`. An `ExpBox True` is attached to each alternative in a multi-equation definition or case expression. The `[String]` components of each `TopLevelBox` or `LocalBox` are compound names. For example, the tick-box associated with the declaration of a local variable `v` in the body of a function `f` declared as a method in a class instance `C Int` would have the `BoxLabel`:

```
LocalBox ["C Int", "f", "v"]
```

`BoolCxt` is used to notate the context for binary tick locations: guards, conditionals and list comprehension qualifiers – the places in Haskell syntax where booleans are used to inform control flow.

The file format is simply the `show` of the `Mix` data structure, allowing for easy reading and writing from inside Haskell programs.

4.7 Program Ticks (.tix) Files

Any run of HPC instrumented code results in a .tix file being written, which contains a list of modules that have been instrumented, along with a simple Integer list for each module representing what tick boxes have been ticked.

```
data Tix = Tix [TixModule]
  deriving (Read, Show)

data TixModule
  = TixModule
       String    -- module name
       Hash      -- hash number
       Int       -- length of tix list
       [Integer] -- actual ticks
  deriving (Read, Show, Eq)
```

The hash number is used to perform sanity checking when merging tix data from separate binaries that share the same instrumented module. The number of ticks is stored simply to make processing of .tix files easier for other (non-Haskell) tools. In this file format, we use Integer rather than the 64-bit word we use in our implementation to keep our API Haskell 98 compliant. In practice no Integer larger than 2^{64} is ever generated, and when there might be an overflow risk we will modify our implementation to use a larger word size – the file format will remain the same.

Similar to .mix files, .tix files are simply a show of the `Tix` data structure.

4.8 Program-index (.pix) Files

In addition to the module-index file, the transformation stage updates a program-index file that records for each module the number of tick-boxes allocated. This index enables the use of a single array of tick-boxes at run-time. For each module an offset is computed for the translation of module-level box numbers into program-level box numbers. The Haskell datatype we use is

```
data Pix = Pix [PixEntry]
  deriving (Read, Show)

data PixEntry
  = PixEntry
       String  -- module name
       Hash    -- Mix's hash number
       Int     -- Number of Tix's
  deriving (Read, Show)
```

Again the file format is simple; .pix files are a show of the `Pix` data structure.

5. Implementation Details

We have two implementations of instrumentation, one based on source-to-source transformation, and one wired deep inside GHC. We shall describe both.

5.1 Source-to-Source Translation

Our source-to-source transformer re-uses the parser and pretty-printer from the NHC98 compiler [9] and the HAT tracing system [8]. Between the parser and pretty printer we insert a rewriting pass that acts on every declaration, applying the instrumentation techniques outlined in the previous section; adding ticks and recording details of locations. The rewriter also inserts an import referring to a small HPC run-time library and a few auxiliary definitions.

We have tried to keep the HPC run-time library very simple. Just three functions are exported for use in our transformed modules:

```
offset :: String -> Int
run    :: Pix -> String -> IO a -> IO a
tick   :: Int -> a -> a
```

The function `offset` takes a module name, and returns an offset into a global tick-box array. It is applied once in each instrumented module, in the declaration of a module-specific value `off`, to compute the correct offset for all tick-box indices in that module. Like this:

```
off :: Int
off = offset "Foo.Bar"

foo n | tick (off + 1) False = undefined
foo n = tick (off + 2)
        (print (tick (off + 3) "Hello"))
```

This run-time computation of offsets is the price we pay for independent instrumentation of modules. Our use in Section 4 of numeric literals as tick-box arguments was just to keep the explanation simple.

The wrapper-function `run` handles initial and final processing of the coverage record. It is defined like this:

```
run pix progName main = do
  count <- tickBoxCount pix progName
  initialiseTickBoxes count progName
  result <- try main
  finaliseTickBoxes count progName
  ( case result of
    Left exception -> throwIO exception
    Right normal   -> return normal )
```

The `tickBoxCount` function determines its result from the .pix file. The `initialiseTickBoxes` function reads the .tix file for previously recorded runs of the program, or if there is no such file it sets all tick counts to zero. The instrumented main is evaluated under `try`: if it fails with a run-time error the exception is caught so that the .tix file is still updated by `finaliseTickBoxes`.

The HPC transformation of programs alters the main computation to become an application of `run` to the tick-instrumented version of the original main function. For example, the declaration

```
main = putStrLn "Hello world!"
```

is transformed to become something like this (again we omit offsets for simplicity and also the details of auxiliaries to read information from the .pix file):

```
main =
  run <pix> "HelloWorld" main
  where
  main | tick 1 False = undefined
  main = tick 2 (putStrLn (tick 3 "Hello world!"))
```

An efficient internal representation of the tick-box mapping is important to avoid excessive run-time overheads. At first we tried a purely functional data structure. But the current version of HPC uses a C array and Haskell's *foreign-function interface* (FFI). If we could freely write a C expression as the body of a Haskell function, `tick` might be defined as follows:

```
tick n x = tickBox[n]++, x
```

Boolean Control Coverage

For if-conditions, guards and qualifiers, *boolean coverage* is recorded by applying a function `boolTick`, defined as follows.

```
boolTick :: Int -> Int -> Bool -> Bool
boolTick nt _ True  = tick nt True
boolTick _ nf False = tick nf False
```

Each `Int` argument is a distinct tick-box number: one box records occurrences of `True` values and the other records occurrences of `False` values. The `Bool` argument is the expression being evaluated.

Take this small example

```
if x > y then a else b
```

We would translate this into

```
if tick 1 (boolTick 3 4 ((tick 5 x) > (tick 6 y)))
  then (tick 7 a)
  else (tick 8 b)
```

Although one might expect `boolTick` to be defined in the HPC library, in fact a local `boolTick` is introduced in each module so that its applications can more easily be inlined. The definition of `boolTick` we have given here could fall foul of common-subexpression elimination in some compilers. A call to `boolTick` with no free variables might be shared between calls. Even with this optimization, we still get valid coverage, but the relevant tick boxes are ticked at most once, rather than the number of times each value occurred in the boolean context.

5.2 GHC Specific Implementation

To allow HPC to support code that uses GHC language extensions (or libraries that use these extensions), we have pushed the source-to-source translator into GHC as an early compiler pass, and made various other changes to allow efficient instrumented code to be generated.

Rewriting Haskell AST

We walk over the Haskell abstract syntax tree after typechecking and renaming, but just before desugaring. At this point, the AST is annotated with source spans locating the beginning and end of each syntactic unit in the source. So it is straightforward to generate `MixEntry` values for tick boxes.

We have added two new constructors to the AST datatype for expressions.

```
data HsExpr id
  = ...
 | HsTick
     Int -- module-local tick-box number
     (HsExpr id) -- sub-expression
  | HsBinTick
     Int -- module-local tick-box number for True
     Int -- module-local tick-box number for False
     (HsExpr id) -- sub-expression
```

`HsTick` and `HsBinTick` correspond exactly with `tick` and `boolTick` used by our source-to-source translator. The tick operations are represented by distinct constructions, rather than by applications of wired-in identifiers, because the constructor representation (1) is more compact, giving shorter compile-times, and (2) makes it easier to ensure that tick-box numbers remain as constants throughout compilation.

The pass that adds in `HsTick` and `HsBinTick` works exactly like the source-to-source translator. Every time it adds a tick or binTick it records the location, and what style of expression was ticked. This information is written out during the compilation of every module, into the .mix file.

Desugaring HsTick and HsBinTick to GHC Core

The desugar pass has two new simple rules:

```
dsExpr (HsTick ix e) =
   [[ case tick# <ix,moduleName> of
        DEFAULT -> e
   ]]
dsExpr (HsBinTick ixT ixF e) =
   [[ case e of
        True -> case tick# <ixT,moduleName> of
                  DEFAULT -> True
        False -> case tick# <ixF,moduleName> of
                  DEFAULT -> False
   ]]
```

In GHC Core, `case` is a strict operator that always evaluates the scrutinee, even if the result is never used. tick# is a magic Id, specifically marked as side effecting, sharing this property with various other primitive functions. The GHC rewrite engine knows not to move side-effecting code, so the optimizer will cause the case (with the tick) to get evaluated if and only if a normal order evaluation would evaluate it.

The magic Id tick# has many instances, each of which contain the module name and tick number. The tick# can be freely passed around between modules via the cross-module inliner, because the tick# has the module of origin baked into it.

From Core To C--

After optimization, Core is translated into Stg, which uses the explicit construct StgTick. The operational interpretation of this constructor is

- Tick the tick box, then
- Enter the given expression.

```
data GenStgExpr bndr occ =
    ...
  | StgTick
    Module                  -- src module
    Int                     -- tick number
    (GenStgExpr bndr occ)   -- sub expression
```

With Stg code that is annotated using StgTick, it is easy to generate native code that reflect the semantics of StgTick.

```
cgExpr (StgTick m n expr) = do cgTickBox m n
                               cgExpr expr
```

`cgTickBox` generates a single 64 bit increment to the preallocated tick box array of the correct module; in C-- (GHC's internal C-like language) such an increment might be written as

```
 I64[_hpc_tickboxes_Main_hpc+40]
    = I64[_hpc_tickboxes_Main_hpc+40] + 1 :: I64;
```

Here `_hpc_tixboxes_Main_hpc` is a static array that the linker can resolve.

6. Use: Applications of HPC

6.1 Surprises in Larger Examples: Visualization Programs

One of the first substantial application programs that we tried instrumenting with HPC was a ray-tracing program. It was written by a Galois team for the ICFP 2000 programming contest. In the HPC coverage report, quite a few auxiliary definitions in the final

```
# usage: hpc-fix progname progargs
PROG=${1:-Main} ; shift ; PROGARGS=$*
echo > old.hpc ; touch new.hpc
until cmp -s new.hpc old.hpc
  do
    mv new.hpc old.hpc
    hpc-run ${PROG} ${PROGARGS}
    hpc-report ${PROG} > new.hpc
  done
rm old.hpc new.hpc
```

Figure 3. Testing until coverage reaches a fixpoint.

program are marked as unused in the final program — only to be expected as the contest forced rapid exploration of alternative ideas. But there is a surprise in the table-driven core of the program: the entire final row in one of the key matrices is redundant!

More recently, we have applied HPC to another table-driven visual application, this time a lazily-streamed version of the marching cubes method for isosurface extraction [4]. The program computes the coordinates of a triangular mesh for rendering by OpenGL. In order to compare the speed of different solutions, one approach is to compute just the number of triangles generated. HPC nicely highlights the shortcoming that this computation does not entail full evaluation of all the coordinates, so the comparison is inaccurate.

We mention these examples for two reasons. First, they confirm the occurrence in practice of examples where fine-grained coverage information tells us something important that a coarser-grained approach would hide. Second, tools are only useful if they are informative, and one measure of information is surprise.

6.2 Using HPC with QuickCheck

Perhaps the most widely used tool for testing Haskell programs is QuickCheck [2], a combinator library for type-driven random testing. As the QuickCheck authors themselves put it:

> *the major limitation of QuickCheck is that there is no measurement of test coverage* [2]

So here is a ready-made application for HPC: it can be used to measure source-coverage in a program when various required properties of its components are tested using QuickCheck.

There are other uses of HPC in connection with QuickCheck. Not only is the application program under test written in Haskell; the *generators* for test values and the property-based *specification* to be tested are also written in Haskell. So HPC can be used to check for coverage of test generators and specifications.

Another issue for QuickCheck testing is when to stop. How many random test cases are enough? Once we have a coverage tool a good pragmatic answer is that testing should continue until further testing no longer improves the accumulated measure of coverage. That is to say, the coverage measure reaches a fixpoint. Figure 3 shows a simple shell script, hpc-fix, which implements this idea.

We have tried hpc-fix on various examples. Among the more interesting of these is a previously published QuickCheck exercise testing the equivalence of an interpreter and a compiler for a simple imperative language [3]. The equivalence is tested both for randomly generated closed expressions and randomly generated programs. Because there is an element of randomness, the coverage figures obtained using hpc-fix vary between runs. But a typical result is that after five or six iterations 98% expression coverage is achieved – an optimal result as some expressions could only be evaluated if there were errors such as the generation of bad code.

6.3 Using HPC to Find Programming Errors

Hpc can be a surprisingly effective tool for locating programming mistakes. It can provide programmers with two kinds of information:

Where not to look: Only code actually executed can contribute to any incorrect result produced by a program. Applying HPC to a single failing run identifies a much smaller debugging target than the whole program. If the programmer can confidently attribute a fault to one of a small number of modules, HPC can be applied to just these modules. Of course a bug may be *not* calling a specific piece of code, but this itself is a bug, and always inside the covered code.

Where to start looking: Another application of HPC is to identify parts of a program most likely to be responsible for a fault that occurs in some but not all runs. Suppose we have a list, `goodTix`, of coverage records for faultless runs of the program, and another such list, `badTix`, for faulty runs. In we consider our list of tick boxes as a bit-vector representation of a set by using '0' for no ticks and '1' for a non-zero number of ticks it is straightforward to compute the union (`\/`), intersection (`/\`) or difference (`\\`) of two the coverage sets. We can use these operations to compute the coverage set

```
fold (/\) badTix \\ fold (\/) goodTix
```

representing the parts of the program that are used in every faulty run but not used in any faultless run. Assuming that the fault is a mistake in the program, it must occur in one of these parts. A variant of hpc-markup can highlight them for the programmer in some distinctive way not used in ordinary hpc-markup output.

6.4 Reaching for 100% Haskell Program Coverage

Our experience of using HPC is one of pragmatic realization that programs simply can not have 100% coverage. There are too many places where it is reasonable that code would not be used during execution. Examples include:

(): it is always acceptable to omit unit from consideration from coverage testing. It can only have one value (or bottom), and evaluating (or not evaluating) bottom can never leave parts of a program untested. A common example of partially covered code is the expression `return ()`. If we know the result of executing some monadic code will return (), we do not pattern match on (), rather we use syntactical sugar that never evaluates the ().

Assert error messages and impossible cases: We often put asserts into our code, as well as calls to error for cases that should never happen. This becomes a judgement call; unit tests for `head` should test that `head []` calls `error`, but functions that have preconditions on being called might have the invocations to `error` masked from coverage counts. This is where the published list of exclusions comes in useful — one can see what standard the tester has applied. As more functional-level tests are performed (eg. by QuickCheck), fewer exclusions are required.

Combinator Library Error Information: Sometimes results from library combinators, for tasks such as parsing, have sum types representing either success with a value, or failure with some information about the "error". Programs may apply such combinators in a way that often results in failures, but the error information is never needed as the response to such failures is always to try a different application. So expressions for error information may never be evaluated.

Dead Code: Arguably dead code should simply removed, but sometimes dead code in one way of building an application is actually an alternate API or entry point for another way of building the same application.

When using HPC to guide a quality improvement initiative, it is helpful to somehow exclude cases, and focus on real coverage holes. We use the mtix DSL to capture such things; for example to ignore every (), we write

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

If we have a debugging entry point for GHCI, we write

```
  module "Misc" {
    tick function "debug"     [debugging];
  }
```

and so on. The effect is that we capture our exclusion in a case by case, application by application basis – documentation of what our programs are doing; surely a good thing.

6.5 Cheap Value Coverage using HPC

With small code modification, HPC can be used to emulate value coverage. Consider the following program fragment.

```
... = filter f . filter g
```

If the Haskell programmer wants to know if the intermediate list between the filters is invoked with both empty and not-empty values, we can using HPC, and write

```
... = filter f . coverage . filter g
  where
     coverage xs = case xs of
                     (_ : _) -> xs
                     [] -> xs
```

Our coverage tool will now highlight the first branch of the case if the list was never non-empty, and the second branch of the case if the list was never empty. This idea can be generalized to handle predicates, and test for boundary conditions. For example, if we wanted to check that at least one intermediate list contains at least one negative number, we could write

```
... = filter f . coverage . filter g
  where
     coverage xs
       | length (filter (< 0) xs) > 0 -> xs
       | otherwise -> xs
```

Our course using this idiom breaks our design principles of non-annotation. The coverage function provided by the user can be arbitrarily complex, and even providing a runtime QuickCheck style test for completeness of value coverage, so we mention it as an extra use-case for HPC, even if outside our original aims.

7. Evaluation

7.1 Performance

For any programming tool that adds extensive instrumentation to a program the extra cost of that machinery is an important issue. In our very first tests with a precursor to HPC, instrumented programs slowed by a factor of over a hundred — which is clearly an unacceptable overhead.

We have used HPC on many large Haskell programs, and found the performance overhead (both compile time and run time) to be both significant and acceptable in practice. In order to quantify our experience, we have run the benchmarks listed in Table 1 with and without instrumentation, and with and without optimization. We use `-O0` for explicitly no optimizations, and `-O2` for strong optimizations. We focus on a short list of benchmarks, including some examples from nofib [10], and a few open source applications written in Haskell. We measure wall-clock time.

Using the current HPC (version 0.4) and compiling with GHC -O2, the worst slow-down factor we have seen is around 3 for GHC -fhpc instrumented code, and around 5 for source-to-source instrumented code. Some programs, such as those with inner loops in non-instrumented `Prelude` or library functions, slow by much smaller factors. Tables 2 give some illustrative run-times with and without HPC-instrumentation[1].

The runtime cost is primarily one of missed optimization opportunity; the actual cost of ticks is very low in our implementation (a 64-bit increment), but as the ticks force the output program to keep some of the structure of the original Haskell, optimization opportunities are lost. We verified this using a special build of GHC that did coverage instrumentation, then at C-- code generation time did not actually emit the tick box increment instructions. We observed that the cost of the ticks themselves account for approximately 30% of the overhead of coverage gathering, supporting our thesis.

Compilation costs also increase when using HPC. As shown in Table 3, in the worst-case, for unoptimised compilation and source-to-source instrumentation, total compile-time increases by a factor of around 6. But using the instrumentation pass integrated in GHC the factor is typically less than 2. We consider these costs more than acceptable, given the new information Haskell programmers can obtain about their code.

7.2 Portability

The original version of HPC was developed using versions 6.2 and 6.4 of GHC [6] under Linux.

Although GHC implements a variety of extensions to the standard Haskell 98 language we have tried to minimize the use of these extensions in HPC. The source-to-source translator uses two such extensions:

- The *Foreign Function Interface*, or FFI, is needed for a small part of HPC's run-time library implementing tick boxes as an array in C. The FFI is defined in an approved addendum to the Haskell report[5], and supported by other implementations.
- The *exception-handling library* in GHC is used to handle any occurrence of a run-time error in an HPC-instrumented program. We must ensure that the coverage record for a failing run is properly recorded before the program exits. When porting HPC to other compilers, some means must be found to achieve the same effect.

Avoiding *dependence* on non-standard extensions is attractive, but what about *support* of non-standard extensions that may be used in application programs? The instrumenting transformation in HPC is syntax-driven, so it is syntactic extensions that pose a problem. The only syntactic extensions recognized by the current version of hpc-build are multi-parameter type classes and functional dependencies. However, the version of HPC integrated with GHC applies the instrumenting transformation to the compiler's internal representation of programs and therefore supports the full range of GHC's language extensions.

Pre-processors of various kinds effectively extend the source language of Haskell programmers still further. Our rule here is that any pre-processing must precede the instrumenting transformation; we do not wish to over-burden HPC's definition of coverage to accommodate a plethora of meta-languages. The current hpc-build script supports pre-processing by `cpphs`. Often it is not appropriate to instrument automatically generated modules for coverage as they are not intended for human readers.

[1] A bug in GHC-6.6 invalidated the optimized build of clausify in both the instrumented and non-instrumented code.

Operating-system dependencies in HPC are reflected mainly in hpc-build and other shell scripts, and in the path-naming conventions for HPC-related files.

7.3 Limitations

HPC transforms modules independently, by purely syntactic rules, without relying on type information and without changing the names or types of declarations. The big advantage of this approach is that modules can be instrumented selectively, according to quite simple rules, at low cost.

The disadvantages do not seem great in comparison. As the hpc-build based transformation is not informed by operator priorities, operands in chains of infix applications are treated as immediate subexpressions of the whole chain. (This imprecision is not present in the GHC internal version, which acts after operator priority resolution). This imprecision really doesn't matter too much since whenever any sub-chain is unevaluated all its operand are unevaluated and can be marked accordingly. The lack of operator priorities does prevent any reliable extension of boolean coverage to the operands of specified logical operators. It will be interesting to see whether that extension is something users ask for.

The other main feature of our design that might seem limiting is the tick-box-array representation of coverage. Our original intention was to record not just tick or no-tick, but a tick-count. The current HPC still has numbers in the boxes, but we only distinguish between zero and non-zero in our presentations. If we finally decide to relinquish numbers, keeping only bits, there could be a performance gain. Or if we keep the numbers perhaps we should make fuller use of them. Koen Claessen suggested to us that we look into using the coverage counts for the purpose of performing statistical analysis of the correlation between bugs and source code locations [16].

HPC cannot currently be used with GHCI, the GHC interpreter. The reason is that HPC is based on the concept of running code and ticking boxes corresponding to entry points, with a pre-determined list of tick-boxes. GHCI permits the user to start and stop execution, and to reload modules. It is not immediately clear how our simple .mix/.tix model could be used to present such dynamically changing information. The tick-box implementation we describe in this paper has already been used in the new GHCI debugger [12] – not for coverage gathering, but to push breakpoints from Haskell source to byte-code.

8. Related Work

So far as we know, the only previous coverage tools for Haskell are those based on HAT traces [1, 11]. These tools highlight parts of the program to which there are references in a trace as they *have* been used. HAT traces contain much more information than is needed for a coverage tool; the traces are expensive to construct and continue to grow as the computation proceeds. So the scale of application for these HAT-based coverage tools is inevitably limited.

There have been a small number of coverage tools developed for other functional languages. OCaml has a simple coverage tool that records entry counts at selected code locations, and can output source code with entry counts in comments. SML/NJ added a simple form of coverage in version 110.50.

Various coverage tools have been developed for mainstream imperative languages such as Java or the C family. For a recent concise survey see [15]. The slow-down costs of these tools are lower than for HPC — less than a factor of two. However, they typically rely on low-level primitives added to a specific implementation of the language. More importantly, the expression coverage of HPC is finer-grained than statement and branch coverage of these tools. Finally, there is still some redundancy in our current scheme for allocating tick-boxes so we may be able to reduce costs further.

9. Conclusions and Further Work

9.1 Conclusions

We have described the rationale behind the design of a Haskell coverage tool-kit, explained how it works and given some results from early experience using it.

The architecture of the tool-kit, particularly the distribution of work between build-time, run-time and report-time, seems to work well. Our main practical difficulty has been the integration of HPC with program-building conventions and processes that are already adapted in quite complex ways to the needs of other tools. As the intended contribution of HPC is essentially practical, and our conclusions are preliminary. The real test of a tool like HPC is how often application developers and testers find it gives fresh and useful information. With `-fhpc` being provided as a standard option with ghc-6.8 we hope for a flood of new HPC users, and thereby better tested Haskell applications and libraries.

We find HPC useful, and hope others will read this paper, use HPC and find it useful too.

9.2 Further Work

Beyond the current implementation there are various ideas we have discussed, rejected for now, but might come back to. An instrumenting transformation informed by precedence and type information could generalize the present boolean coverage for guards, conditions and qualifiers, to include:

- boolean arguments in applications of `Prelude` functions such as the logical operators `(&&)` and `(||)`;
- boolean arguments in applications of specified user-defined functions;
- similar value-coverage for other datatypes.

Another view of boolean coverage is that it extends the scope of coverage to include execution paths not directly associated with the evaluation of a source expression. Other possible extensions with the same characteristic include:

- success and failure coverage of argument patterns;
- path coverage in the multi-graph of dependencies between declarations.

The default by design is that the coverage information for a program that is run many times accumulates in a single .tix file. So it is hardly surprising that our current tools for processing HPC results are applied to a single .tix coverage file — excepting the simple test for equality of .tix files by hpc-fix. However, as already noted in Section 6, separate coverage records for multiple runs could provide inputs to further tools for program slicing and the location of errors.

HPC could be used to show the sequence in which source expressions are evaluated. Calls to a `tickBox` variant could inform an interactive source highlighter. A variation of this idea would be to keep details of the N most recent tick operations, instrumenting calls to `error` to trigger interactive playback.

Haskell is a fine imperative programming language, and a reasonable question might be to ask how HPC compares with traditional tools when Haskell is used as an imperative language. Would line coverage be enough? Are path coverage extensions needed?

9.3 Obtaining HPC

HPC is open-source and obtainable under a BSD-style license from

```
http://projects.unsafePerformIO.com/hpc
```

At the time of writing the current version is HPC-0.4.

9.4 GHC version of HPC

HPC instrumentation will be included with GHC-6.8. This paper uses the commands provided with HPC-0.4, and there are some cosmetic differences in the version provided with GHC. The principal difference is the HPC post-processing tools have been given a darcs and svn style interface, in which a single binary (hpc) brokers the functionality provided by hpc-report, hpc-markup, and the other HPC-0.4 tools.

Like darcs and svn, the hpc tool is self documenting.

```
$ hpc help
Usage: hpc COMMAND ...

Commands:
  help        Display help for hpc or a single command
Reporting Coverage:
  report      Output textual report about program coverage
  markup      Markup Haskell source with program coverage
Processing Coverage files:
  combine     Combine multiple .tix files in a single .tix file
Coverage Overlays:
  overlay     Generate a .tix file from an overlay file
  draft       Generate draft overlay that provides 100% coverage
Others:
  show        Show .tix file in readable, verbose format
  version     Display version for hpc
```

There is no hpc build provided with GHC – we use -fhpc instead.

Acknowledgments

We developed the design ideas and first implementation of HPC at Galois in the early Summer of 2006. Colin Runciman thanks Galois and the University of York for hospitality and support during his visit.

The source-to-source transformer in HPC re-uses the parser and pretty-printer from the NHC98 compiler and the HAT tracing system. All parts of HPC were developed and tested using the GHC compiler. We are pleased to acknowledge the authors of these systems for their direct or indirect contributions to our work.

We would also like the thank Malcolm Wallace for being our patient and diligent beta tester, Thorkil Naur for his detailed comments, and the anonymous reviewers for their helpful feedback and suggestions.

References

[1] Olaf Chitil. Hat-explore: Source-based trace exploration. In Colin Runciman, editor, *Hat Day 2005: work in progress on the Hat tracing system for Haskell*, pages 1–5. Tech. Report YCS-2005-395, Dept. of Computer Science, University of York, UK, October 2005.

[2] K. Claessen and R. J. M. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th ACM Conf. on Functional Programming (ICFP'00)*, pages 268–279. ACM Press, 2000.

[3] K. Claessen, C. Runciman, O. Chitil, R. J. M. Hughes, and M. Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *Advanced Functional Programming (AFP'02)*, pages 59–99. Springer LNCS 2638, 2002.

[4] D. Duke, M. Wallace, R. Borgo, and C. Runciman. Fine-grained visualization pipelines and lazy functional languages. *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[5] The haskell 98 foreign-function interface 1.0: an addendum to the haskell 98 report. http://www.cs.unsw.edu.au/c̃hak/haskell/ffi.

[6] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/.

[7] A Gill. Debugging Haskell by observing intermediate data structures. In *Haskell Workshop*, 2000.

[8] Hat – the Haskell Tracer. http://www.cs.york.ac.uk/fp/hat.

[9] The nhc98 Haskell compiler. http://www.cs.york.ac.uk/fp/nhc98.

[10] W. Partain. The NoFib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[11] C. Runciman. Deriving program coverage from Hat traces. In *Hat Day 2005: work in progress on the Hat tracing system for Haskell*, pages 27–32. YCS 395, Dept. of Computer Science, University of York, 2005.

[12] B. Pope S. Marlow, J. Iborra and A. Gill. A lightweight interactive debugger for haskell. In *Haskell Workshop*. ACM Press, September 2007.

[13] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, 2005.

[14] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *Proc. 5th Haskell Workshop*, 2001.

[15] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *Proc. Intl. Workshop on Automation of Software Testing (AST'06)*, pages 99–103. ACM Press, 2006.

[16] A. Zheng, M. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs, 2003.

[17] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29:366–427, 1997.

Table 1. Benchmarks for HPC

Program name	modules	lines of code	Description	Problem
`primes`	1	18	the classic primes example from nofib	finding the 10,000th prime
`clausify`	1	184	from nofib	same input as nofib
`mate-in-n`	5	411	is a chess end-game solver	with the *kohtz* mate-in-5 problem as input
`ray-trace`	16	2312	from the ICFP 2000 programming contest	with the *dice* example as input
`happy`	17	6348	a parser generator	generating GHC's parser
`anna`	32	9561	from nofib	same input as nofib
`darcs`	124	31,553	distributed version control	creating a local copy of the hpc repo
`ghc-6.7`	270	185,631	our development copy of ghc, version 6.7	compiling the the ray-trace example

Table 2. Runtimes for our Benchmarks

	hpc build (ghc-6.6)				-fhpc flag (ghc-6.7)			
	Without Optimization		With Optimization		Without Optimization		With Optimization	
	Without	With	Without	With	Without	With	Without	With
Program name	HPC	hpc-build	HPC	hpc-build	HPC	-fhpc	HPC	-fhpc
`primes`	6.7s	52.7s (x7.8)	3.5s	7.9s (x2.3)	7.7s	14.2s (x1.9)	3.5s	4.2s (x1.2)
`clausify`	1.9s	32.6s (x16.8)	–	–	1.8s	4.1s (x2.3)	1.1s	2.5s (x2.4)
`mate-in-n`	19.3s	203.3s (x10.5)	6.8s	28.8s (x4.2)	19.0s	37.1s (x2.0)	6.5s	13.9s (x2.1)
`ray-trace`	10.6s	91.3s (x8.6)	3.1s	14.7s (x4.7)	13.5s	20.1s (x1.5)	3.3s	5.2s (x1.6)
`happy`	–	–	–	–	9.9s	13.2s (x1.3)	6.8s	11.5s (x1.7)
`anna`	2.4s	43.0s (x18.1)	2.0s	7.0s (x3.5)	2.6s	5.2s (x2.0)	2.2s	4.3s (x2.0)
`darcs`	–	–	–	–	7.3s	9.4s (x1.3)	2.9s	5.6s (x1.9)
`ghc-6.7`	–	–	–	–	19.0s	42.0s (x2.2)	9.5s	26.7s (x2.8)

Table 3. Compile times for our Benchmarks

	hpc build (ghc-6.6)				-fhpc flag (ghc-6.7)			
	Without Optimization		With Optimization		Without Optimization		With Optimization	
	Without	With	Without	With	Without	With	Without	With
Program name	HPC	hpc-build	HPC	hpc-build	HPC	-fhpc	HPC	-fhpc
`primes`	0.8s	1.6s (x2.0)	1.1s	2.1s (x1.9)	0.9s	0.9s (x1.0)	1.0s	1.0s (x1.0)
`clausify`	1.0s	6.0s (x5.9)	–	–	1.2s	1.2s (x1.0)	1.4s	2.6s (x1.9)
`mate-in-n`	1.6s	6.0s (x3.8)	7.3s	17.4s (x2.4)	1.7s	2.2s (x1.3)	2.9s	4.1s (x1.4)
`ray-trace`	4.9s	17.2s (x3.5)	33.5s	61.4s (x1.8)	5.3s	8.0s (x1.5)	9.2s	15.5s (x1.7)
`happy`	–	–	–	–	8.7s	12.4s (x1.4)	17.4s	38.5s (x2.2)
`anna`	7.7s	47.6s (x6.2)	57.4s	155.2s (x2.7)	7.8s	13.6s (x1.7)	17.4s	68.0s (x3.9)
`darcs`	–	–	–	–	273.6s	429.6s (x1.6)	439.4s	738.0s (x1.7)
`ghc-6.7`	–	–	–	–	195.5s	303.3s (x1.6)	421.6s	797.4s (x1.9)