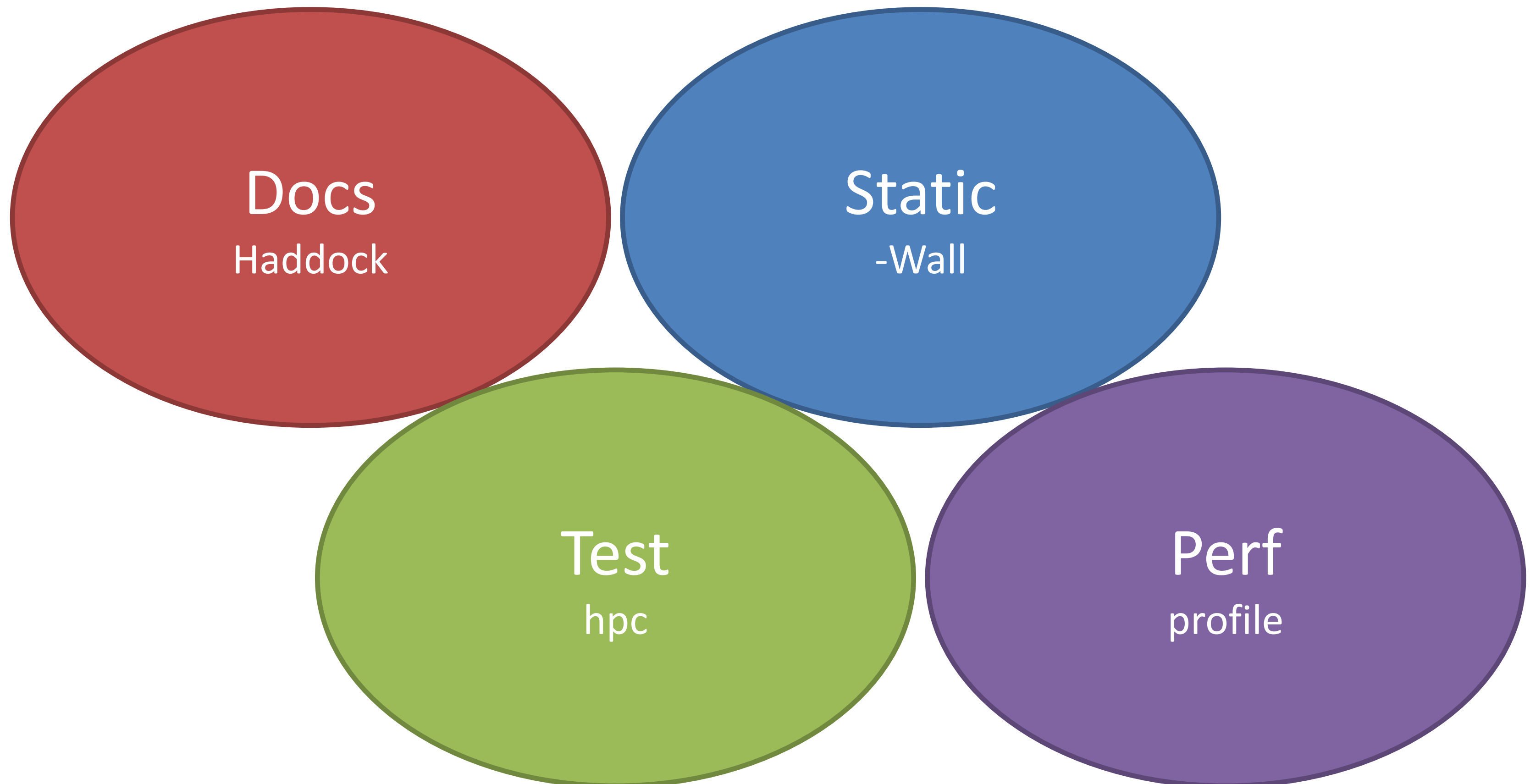
Drive-by Haskell Contributions

Neil Mitchell

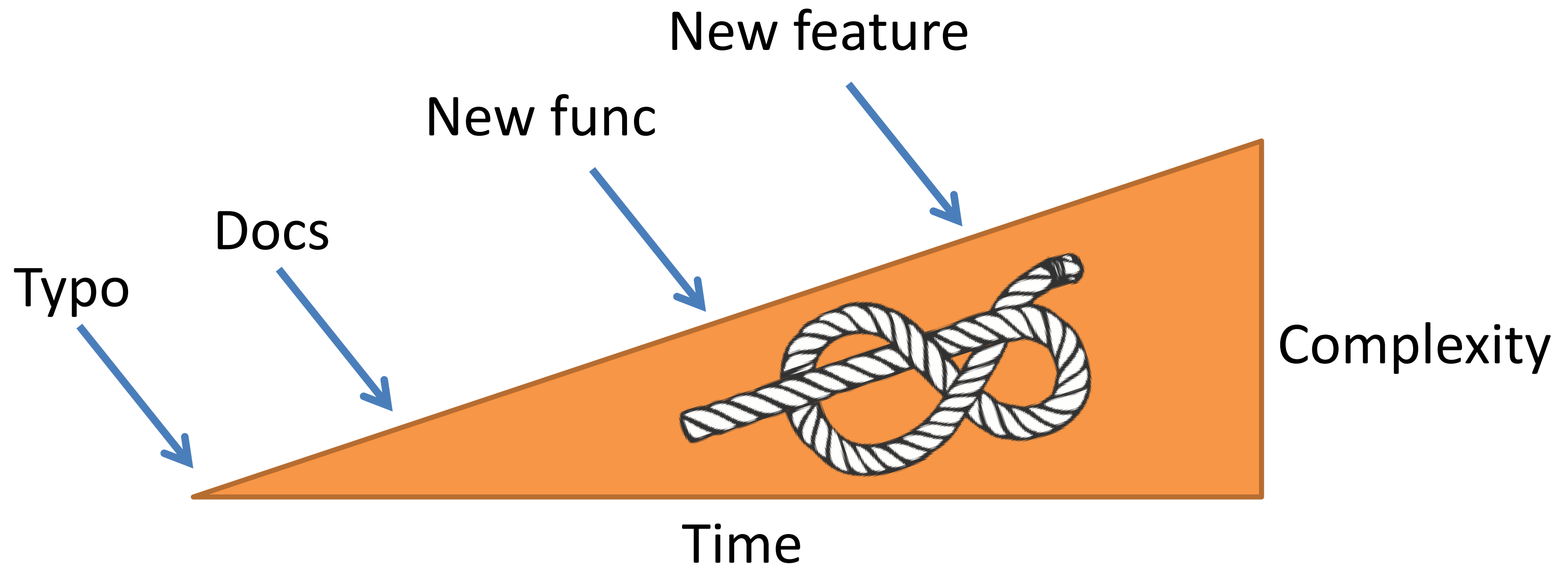
http://ndmitchell.com

Getting started contributing

- *Or*: ideas to improve your existing project



Goal: Start doing cooler stuff



Do: Check your change is welcome

- Is the project on GitHub?
- Look at the open PR’s – do they languish?
- When was the last commit?
- Does it compile with the latest deps?
- On Stackage?
- Improve things you use/believe-in
- Is there a contrib policy? Is it friendly?
- Ask before investing too much (github issue)

Perhaps: Infer tone from docs

I welcome and appreciate contributions. If you've contributed to my code, and we meet in real life, I'll buy you a beer.

If you want to amend a pull request, rewrite your branch and leave a comment. Do not add commits to the branch or open new pull requests for that.

Don't: Rearrange the deck chairs

e.g. Reindent, add -Wall, add new dependencies

Docs

Build and use it

- Was that easy?
- If not, improve the README
  - What it does
  - Why you should use it
  - How you use it (example)

- Maintainers have too much knowledge to do this well

Docs

https://www.haskell.org/haddock/

Look at the Haddock

- Are the functions clear?
  - More examples required?
  - Are corner cases clear?
  - Add docs liberally, don’t worry about being wrong
- Haddock coverage stats are useless
  - I use “haddock --hoogle” then munge the output

Do the docs work

- Rule: Code that is not compiled rots
  - Includes Haddock comments and examples in the manual
- Manually check a few instances
  - Report any buggy examples
  - Perhaps a bigger project of automatic checks?

Docs

Is the bug tracker clean?

- Are all the things on the bug tracker still relevant?
- Are there things on the bug tracker that are related but not cross-linked?

- Beware: Don’t want to add to maintainer woes

Static

Apply static checkers

- Do: apply static checkers, report good finds
- Don’t: make maintainers use them

- Maintainers may *choose* to use the static checker if the payoff is high, but that’s up to them

Static

-Wall

- cabal build --ghc-options=-Wall
  - Get a list of the issues, which make sense?
- Example: Shake has 895 warnings
  - Most in the test suite, plenty unused arguments

```
Shake\Classes.hs:5:15: warning: [-Wdodgy-exports]
    The export item `Typeable(..)' suggests that
    `Typeable' has methods, but it has none
```

A thread to pull on, not an answer

Static

http://hackage.haskell.org/package/hlint

HLint

- cabal install hlint && hlint . --report
  - See report.html, which make sense?

All hints
- **Warning: Use and (1)**
- Warning: Use elem (1)

All files
- Sample.hs (2)

Report generated by HLint v0.0 - a tool to suggest improvements to your Haskell code.

Sample.hs:5:7: Warning: Use and
Found

```
foldr1 (&&)
```

Why not

```
and
```

Note: removes error on `[]`

HLint best hints

- HLint reports a lot – find the "good stuff"

- From Shake:

```
{-# LANGUAGE GeneralizedNewtypeDeriving,
DeriveDataTypeable, ScopedTypeVariables,
ConstraintKinds #-}
{-# LANGUAGE UndecidableInstances, TypeFamilies,
ConstraintKinds #-}
```

Static

HLint good stuff

- Redundant language extensions
- Use of mapM instead of mapM_
- Simple sugar functions (concatMap)
  - Look for refactor introduced noise

- Don’t rearrange the deck chairs:
  - If vs case
  - Redundant lambda

Static

How HLint works

- Parse the source (using haskell-src-exts)
- Traverse the syntax tree (using uniplate)
- Some hints are hardcoded (e.g. extensions)
- Most hints are expression templates
  - {lhs: map (uncurry f) (zip x y), rhs: zipWith f x y}
  - {lhs: not (elem x y), rhs: notElem x y}
  - {lhs: any id, rhs: or}

Static

How HLint works

```
findIdeas
    :: [HintRule] -> Scope ->
    -> Decl_ -> [Idea]
findIdeas matches s decl =
   [ (idea (hintRuleSeverity m) (hintRuleName m) x y
[r]){ideaNote=notes}
   | (parent,x) <- universeParentExp decl, not $ isParen x
   , m <- matches, Just (y,notes, subst, rule) <- [matchIdea s
decl m parent x]
   , let r = R.Replace R.Expr (toSS x) subst (prettyPrint rule)]
```

Static

http://hackage.haskell.org/package/weeder

Weeder

- Finds the “weeds” in a program
  - weeder .

```
= Package ghcid

== Section exe:ghcid test:ghcid_test
Module reused between components
* Ghcid
Weeds exported
* Wait
  - withWaiterPoll
```

Module used in two cabal projects

Function exported but not used elsewhere

Static

Weeder best hints

- Code is exported and not used outside
  - Delete the export
- GHC warnings detect definition is unused
  - Delete the code entirely
- Package dependency is not used
  - Remove a dependency (see also packdeps)

Static

How Weeder works

- Stack compiles with dump .hi files
  - Each module has a large blob of text
- Parse these .hi files, extract relevant data
  - What packages you make use of
  - What imported identifiers you use
- Analyse
  - If ‘foo’ is exported, but not used, it’s a weed

Static

How Weeder works

```
data Hi = Hi
  {hiModuleName :: ModuleName
    -- ^ Module name
  ,hiImportPackage :: Set.HashSet PackageName
    -- ^ Packages imported by this module
  ,hiExportIdent :: Set.HashSet Ident
    -- ^ Identifiers exported by this module
  ,hiImportIdent :: Set.HashSet Ident
    -- ^ Identifiers used by this module
  ,hiImportModule :: Set.HashSet ModuleName
    -- ^ Modules imported and used by this module
```

Static

HLint and Weeder

- Both have binary releases on github

```
curl -sL https://.../hlint/travis.sh | sh -s .
```

- Both have ignore files

```
weeder . --yaml > .weeder.yaml
hlint . --default > .hlint.yaml
```

Tests are great

- Writing good tests takes time – often missed
- Find tests that are missing
  - Will often lead to bugs (also fun to fix)

- Beware, tests are not always good:
  - Verbosity (don’t check a dumb 1 liner)
  - Performance (1M iterations of QuickCheck)
  - Maintenance (do they need updating often)

Test

Read bug reports

- Take a bug report
  - Is there a reproducible case? If not, write it
  - Is the test case machine checked? If not, make it
  - Is it ready to go in the test suite? If not, make it
- Now you have your test
  - Is it fixed? Great, submit a pull request with it
  - Is it still broken? Share the test anyway

Test

Use HPC

- Run the test suite through HPC

```
ghc -fhpc Main.hs && ./main
hpc report main.tix && hpc markup main.tix
```

```
readRule :: HintRule -> [HintRule]
readRule (m@HintRule{hintRuleLHS=(fmapAn -> hintRuleLHS), hintRu
    (:) m{hintRuleLHS=hintRuleLHS,hintRuleSide=hintRuleSide,hint
        (l,v1) <- dotVersion hintRuleLHS
        (r,v2) <- dotVersion hintRuleRHS
        guard $ v1 == v2 && l /= [] && (length l > 1 || length r
        if r /= [] then
            [m{hintRuleLHS=dotApps l, hintRuleRHS=dotApps r, hin
            ,m{hintRuleLHS=dotApps (l++[toNamed v1]), hintRuleRH
         else if length l > 1 then
            [m{hintRuleLHS=dotApps l, hintRuleRHS=toNamed "id",
            ,m{hintRuleLHS=dotApps (l++[toNamed v1]), hintRuleRH
         else []
```

Test

HPC – complex and untested

- Do: Look for the sweet spot
  - Code that is not obviously correct
  - Code that is untested
  - Add a test based on its docs (are they sufficient?)
- Don’t: Aim for 100% coverage
  - You want to *reach* that, not *aim* for it
  - Incentives matter

Test

Run on Travis/Appveyor

- A good CI is import for a project
  - Travis = Linux/Mac, Appveyor = Windows
- Very time consuming to set up
- There is a lot of variety
  - Hvr provides a PPA archive of GHC binaries
  - Stack can grab GHC binaries
  - I use bootstrap scripts

Test

Bootstrap scripts

- Each repo…

```
curl -sL https://.../travis.sh | sh
```

- …calls a centralised shell script…

```
apt-get install ghc-$GHCVER
cabal install neil
./neil
```

- …which calls Haskell

```
system_ "cabal check"
```

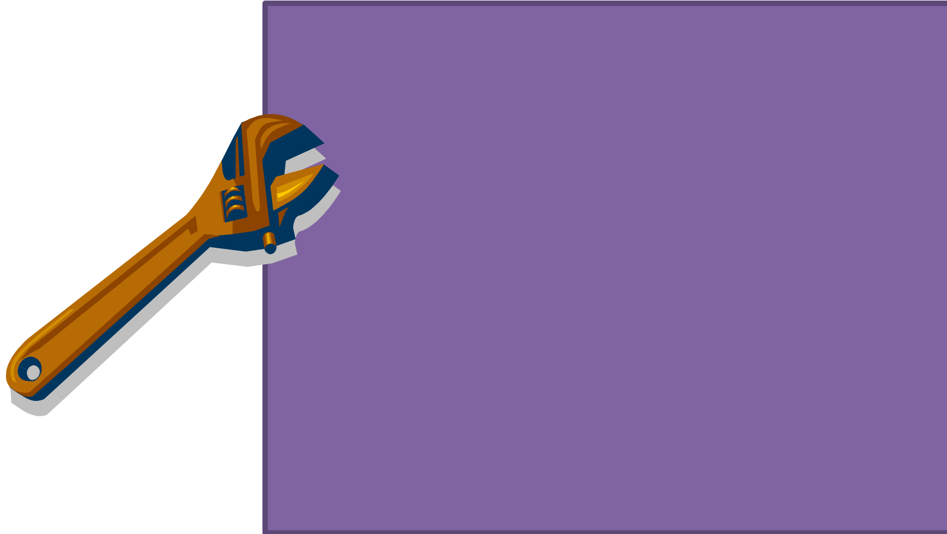
No \r
Installs cleanly
Full documentation
Lowercase cabal keys

Performance

- It’s nice for most code to be faster, smaller
  - But make sure the tests are reasonable first
- Do: Check performance matters
  - Saving 20% on a 1ms operation is often useless
  - Saving 50% on something running yearly is useless
  - All these apply to memory as well

The simple view

- Measure, Whack, repeat
  - Something to measure
  - Somehow to direct your whack



Time profiling

```
ghc Main.hs -prof -auto-all && ./Main +RTS -p
```

- HLint generates 6590 lines, top is a table

COST CENTRE	MODULE	% TIME	% ALLOC
unifyExp	Hint.Match	23.0	2.0
findIdeas	Hint.Match	10.5	0.2
uniplateData	Data.Generics.Uniplate.Internal.Data	7.5	19.8
set_unions	Data.Generics.Uniplate.Internal.Data	6.2	3.8
matchIdea	Hint.Match	6.1	12.8
follower	Data.Generics.Uniplate.Internal.Data	4.1	1.0
pushContextL	Language.Haskell.Exts.ParseMonad	4.0	5.3

Perf

Time profiling tree

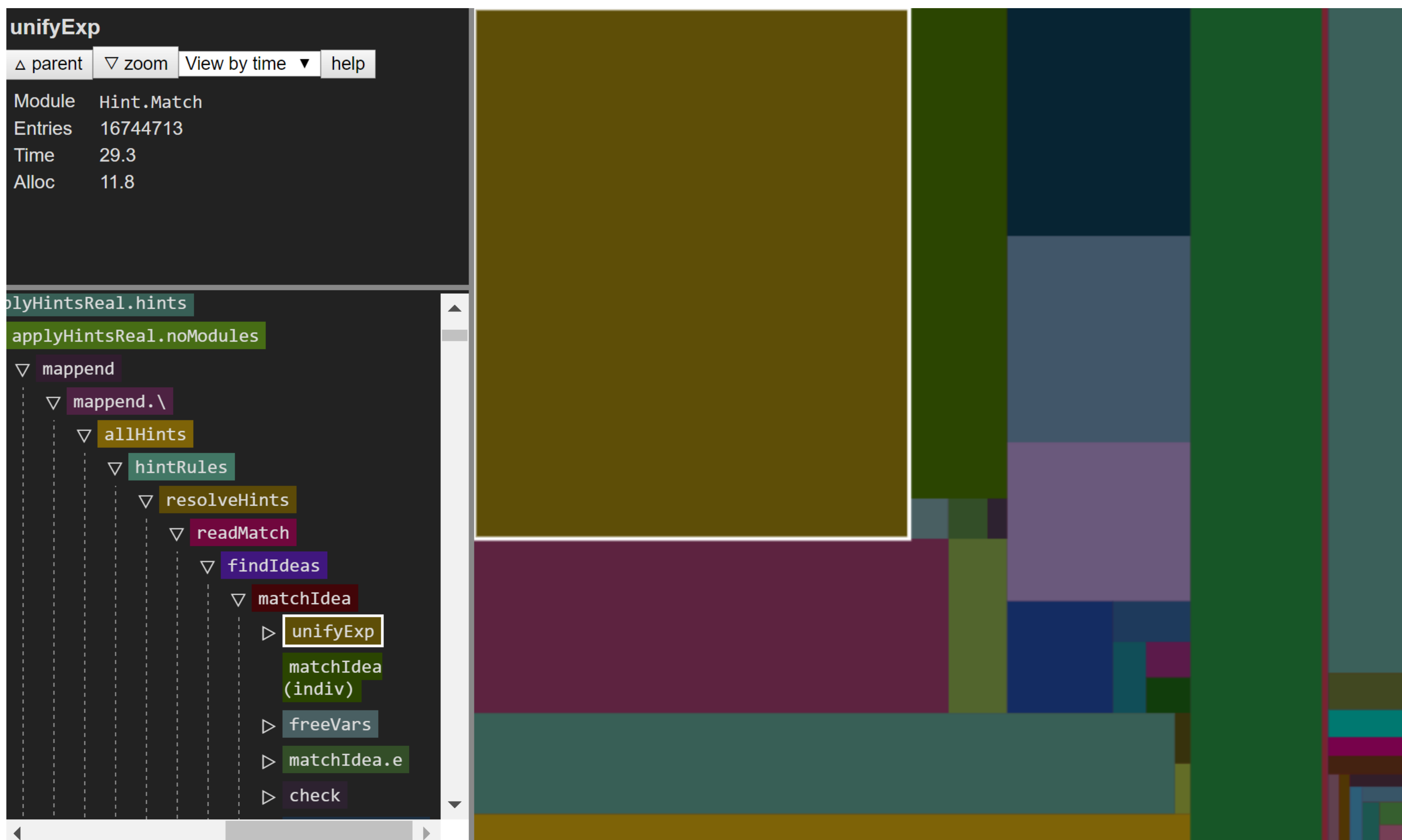
COST CENTRE	MODULE	ENTRIES	%TIME	%ALLOC	%TIME	%ALLOC
unifyExp	Hint.Match	16744713	23	2	29.3	11.8
isDot	HSE.Util	3	0	0	0	0
rebracket	Hint.Match	41	0	0	0	0
opExp	HSE.Util	152744	0	0.1	0	0.1
nmOp	Hint.Match	678224	0	0	1.7	3.4
isDol	HSE.Util	706356	0	0	0	0
matchIdea.nm	Hint.Match	831226	0	0	1.8	2.5
fromParen	HSE.Util	433761	0.2	0	0.2	0
fromNamed	HSE.Match	1728163	0.2	0	0.2	0
isUnifyVar	Config.Type	1728163	0	0	0	0

In reality, way harder to view…

Perf

http://hackage.haskell.org/package/profiteur

Time profiling - Profiteur



Perf

http://hackage.haskell.org/package/profiterole

Time profiling - Profiterole

- Profiterole generates 442 lines, CSE and roots

TOT	INH	IND	
51.0	47.4	-	Hint.Match readMatch (53)
12.0	12.0	-	Data.Generics.Uniplate.Internal.Data readCacheFollower (3)
10.3	10.2	.6	Language.Haskell.Exts parseFileContentsWithComments (53)
8.7	7.5	7.5	Data.Generics.Uniplate.Internal.Data uniplateData (1377837)
99.9	5.2	-	MAIN MAIN (0)
2.9	2.8	2.0	Data.Generics.Uniplate.Internal.Data descendBiData (109203)
2.4	2.4	-	HSE.All runCpp (53)

Perf

Profiterole tower

TOT	INH	IND	
12.0	12.0	-	Data.Generics.Uniplate.Internal.Data readCacheFollower (3)
7.2	7.2	-	Data.Generics.Uniplate.Internal.Data insertHitMap (2)
7.2	7.2	-	Data.Generics.Uniplate.Internal.Data fixEq (7)
7.2	7.2	6.2	Data.Generics.Uniplate.Internal.Data set_unions (0)
1.0	1.0	1.0	Data.HashMap.Array new_ (558259)
4.8	4.8	4.1	Data.Generics.Uniplate.Internal.Data follower (2)
.5	.5	.5	Data.HashMap.Base sparseIndex (635260)

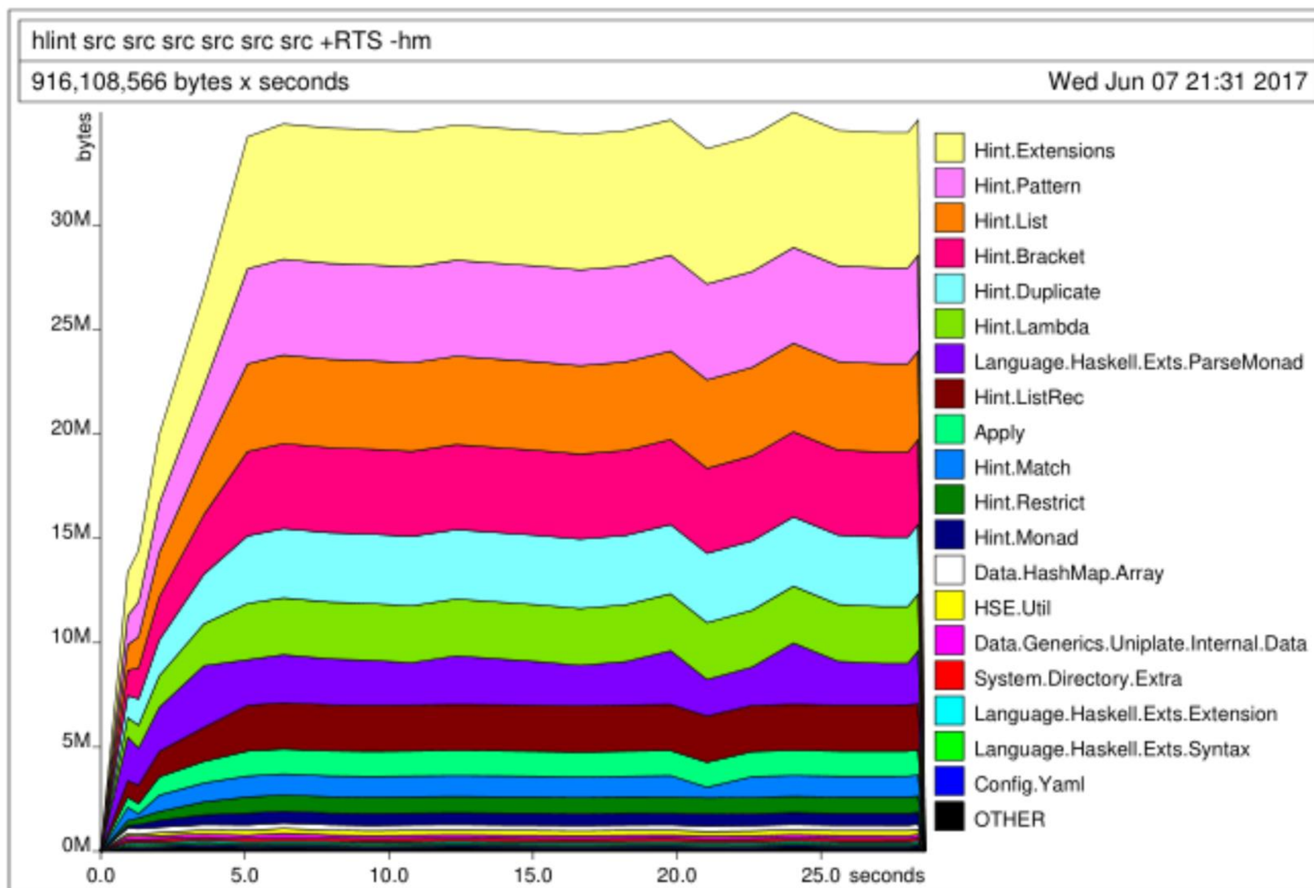
Previously readCacheFollower was in 155 distinct places

How Profiterole works

- Read GHC .prof with ghc-prof library
- Build a Tree Val, Val = {Name, TOT, INH, IND}
- Find roots
  - Called by more than 2 places, or in a config file
- Lift roots to the top-level
- Merge equally named roots
- Write back out
- Can take 200K lines to 5K

Memory profiling

```
ghc Main.hs -prof -auto-all && ./Main +RTS -hm
hp2ps -c Main
```



Perf

https://github.com/ndmitchell/spaceleak

Stack profiling

```
ghc --make Main.hs -rtsopts -prof -auto-all
```

- Compile with profiling

```
./Main +RTS -K${N}K
```

- Find lowest `${N}` where program works

```
./Main +RTS -xc -K${N-1}K
```

- Get a stack trace, examine it

- Fix. Repeat until `-K1K` works

Find performance bugs in vector, base, QuickCheck, happy, pretty…

Let the drive-by contributions begin!