Static Analysis of Haskell

Neil Mitchell

http://ndmitchell.com

Static Analysis is…

…getting insights at compile time

- Full branch coverage
- Terminates
- Doesn’t rely on a test suite

Types are static analysis. Let’s talk about more fun ones.

Examples

- Practical
  - GHC exhaustiveness checker
  - HLint style checker
  - Weeder dead export detector
  - LiquidHaskell refinement type analysis
  - AProVE termination checking
  - Catch error free checker
- Academic

Static Analysis is not perfect

data GoodBad = Good | Bad

truth_p :: Program -> GoodBad

analysis_p :: Program -> Maybe GoodBad

	Good	**Bad**
Just Good	☺	False negative
Nothing		
Just Bad	False positive	☺

Static Analysis thoughts

- Given a warning, what does it mean?
- Can you ignore false positives?
- Is heat-death of the universe a concern?

- Does the analysis check something useful?
  - Property you actually want (don’t crash)
  - Property the analysis aims for (complete patterns)
  - Property the analysis reaches (some patterns)

GOAL:

Maintainable program that does the right thing

Type System

Goal: No errors caused by values from the wrong set. Provide documentation.

Method: Hindley-Milner type inference, unification, System-F.

Caveats: unsafeCoerce, unsafePerformIO, newtype deriving, imprecise sets

So good it is built into the language!

GHC Pattern Match Checker

Is this function fully defined? Over-defined?

```
zip :: [a] -> [b] -> [(a,b)]
zip []     []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

“GADTs Meet Their Match”

Zip pattern results

```
zip :: [a] -> [b] -> [(a,b)]
zip []     []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

```
PMatch.hs:5:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for `zip':
        Patterns not matched:
            [] (_:_)
            (_:_) []
```

Another pattern example

Is this function over defined? Any redundant lines?

```
g :: Bool -> Bool -> Int
g _    False = 1
g True False = 2
g _    _     = 3
```

Pattern match checking

Goal: Detect any missing patterns. Aware of laziness, GADTs, view patterns, guards etc.

Method: For each clause

- C: what is covered – {[] []} {_:_ _:_}
- D: what diverges – {⊥ _, [] ⊥}
- U: what is uncovered – {_:_ [], [] _:_}

Pattern match problems

Caveats:

- If you use ‘head’ you get no warning – says about pattern matches, not runtime errors
- Problem is NP at worst, so has fuel limit
  - `f A = (); f B = (); f C = (); …`
  - Does (#ctors-1)! steps, e.g. 26 = 1.5e26
- Uses an imprecise oracle for guards etc
- Doesn’t understand pattern synonyms (v8.0)

Catch

```
risers :: Ord a => [a] -> [[a]]
risers [] = []
risers [x] = [[x]]
risers (x:y:etc)
    | x <= y = (x:s):ss
    | otherwise = [x]:(s:ss)
    where (s:ss) = risers (y:etc)
```

“Not all patterns but enough”

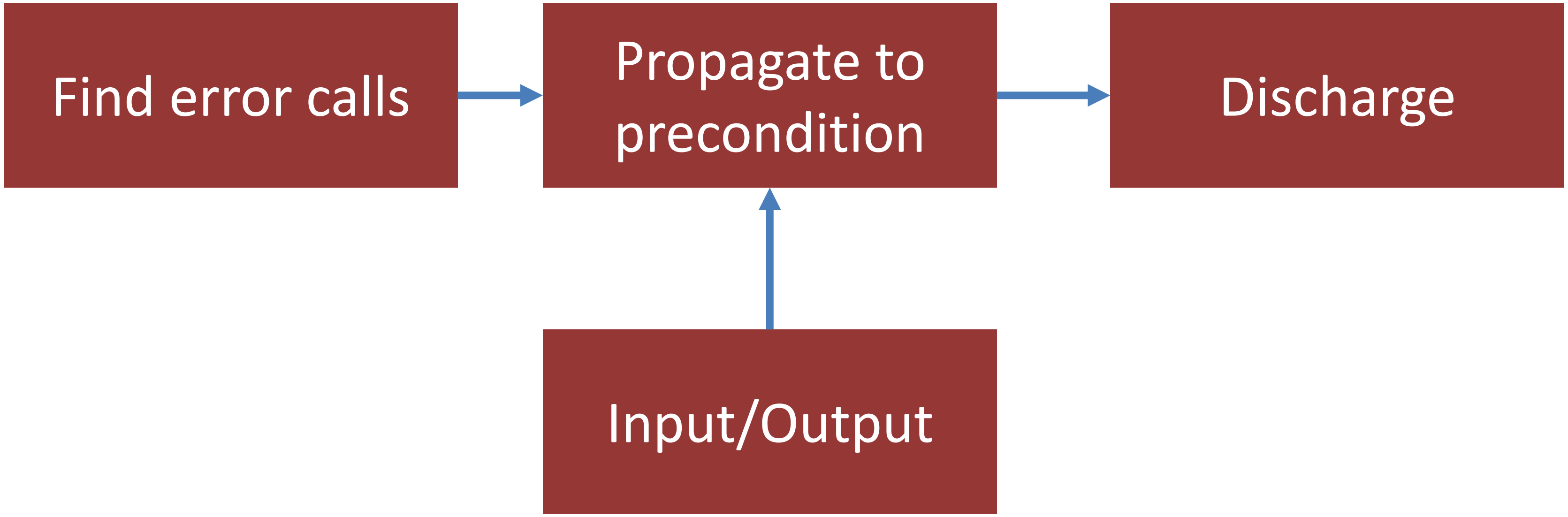
Catch explanation

- *Not* fully defined – GHC raises a warning
- Yet *will not* raise an error at runtime
- Catch infers relationships:
    - risers x = {_:_} ⇒ x = {_:_}
    - otherwise = {True} ⇒ True

Goal: Prove the program will not raise an error

Catch details

Method: For each call to `error`, prove it is unreachable



Catch relations

- precond :: FuncName -> Prop (Arg, Pat)
  - What properties do the arguments need to satisfy
  - To avoid an error
- postcond :: FuncName -> Pat -> Prop (Arg, Pat)
  - To obtain the returning pattern

- Functions are *recursive*, so take fixed point
- Pat has to be limited (paper has two forms)

Catch overview

```
head x = case x of x:xs -> x; [] -> error
main = head (risers [1])
```

precond head = {_:_}

postcond risers {_:_} = {_:_}

precond risers = {*}

precond main = {*}

Catch Weaknesses

Caveats:

- Research tool that used to work with Yhc only
- Patterns are necessarily finite, so approximate
- Code must be first-order
    - Used in conjunction with Firstify, whole program

On the plus side, found 4 real bugs with HsColour and proved the rest correct

Liquid Haskell

- Tool for giving more expressive types
  - But these types are a bit weird, so still fun ☺
- Checking integer predicates using SMT
  - SMT = huge hammer, but available pre-built

```
{-@ type NonEmpty a = {v:[a] | 0 < len v} @-}
{-@ head :: NonEmpty a -> a @-}
head (x:_) = x
```

"Refinement Types For Haskell"

Int’s instead of structure

- Patterns are Int, not structural
  - Very different to GHC warnings/Catch
  - But can do termination and error detection
- Very suitable for Vector/ByteString indexing
  - Found a bug in text mapAccumL
- Type checking plus SMT

```
risers :: l:_ -> {v:_ | NonEmp l => NonEmp v}
```

Liquid Haskell summary

Goal: Catch errors with a bit of Int.

Method: Type system with SMT to solve Int bit.

Caveats: Weird! Very different to dependent types – is this the direction we should go in? LiquidHaskell has lots of things in it, a bit of a mixed bag? I failed to install when I tried a while back.

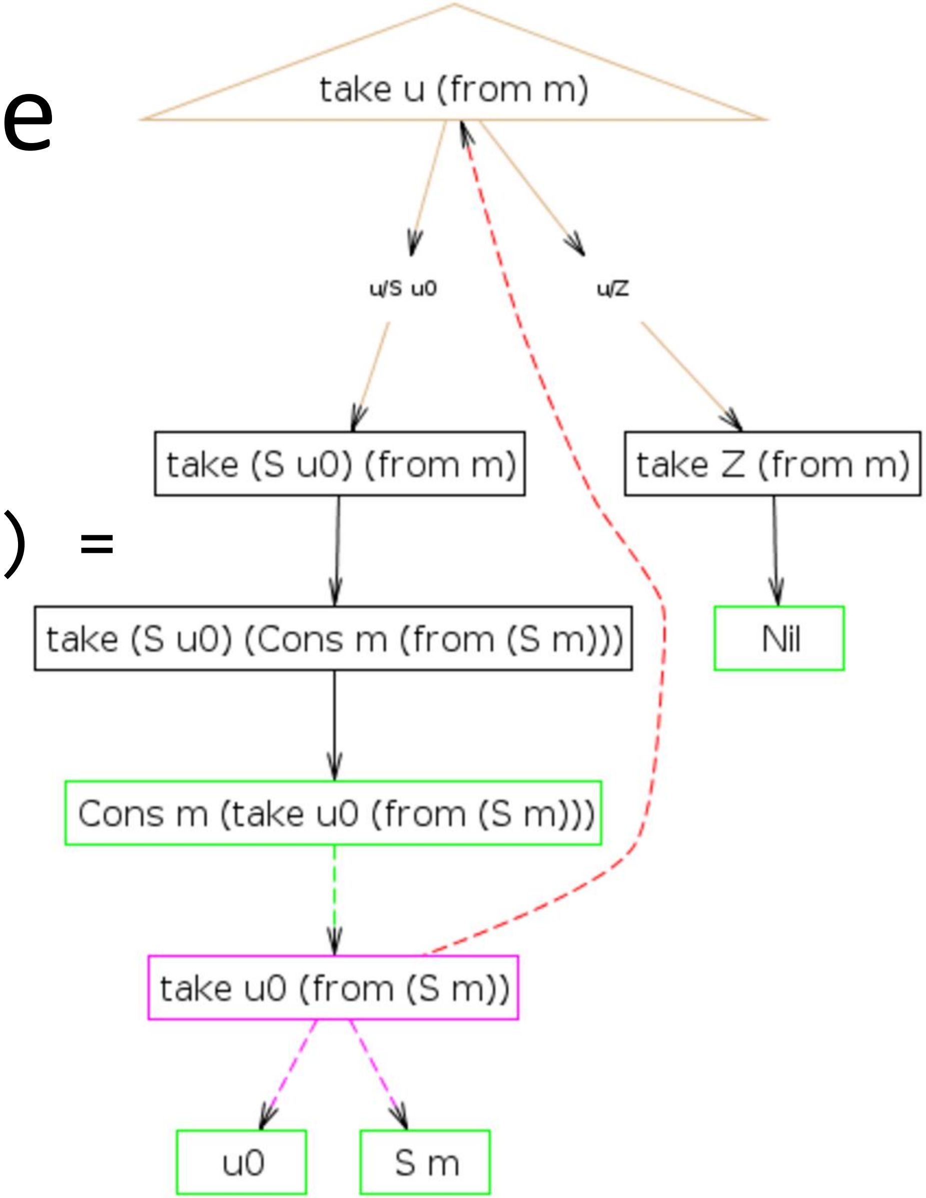
AProVE

- Termination checker – prove the program terminates
  - Take an amazing term-rewriting system (TRS) termination checker
  - Smash Haskell into a TRS

"Automated Termination Analysis for Haskell"

Example

```
take Z xs = Nil
take n Nil = Nil
take (S n) (Cons x xs) =
  Cons x (take n xs)
```



```
new_take(S(u0), m) →
  new_take(u0, S(m))
```

AProVE summary

Goal: Detect non-termination.

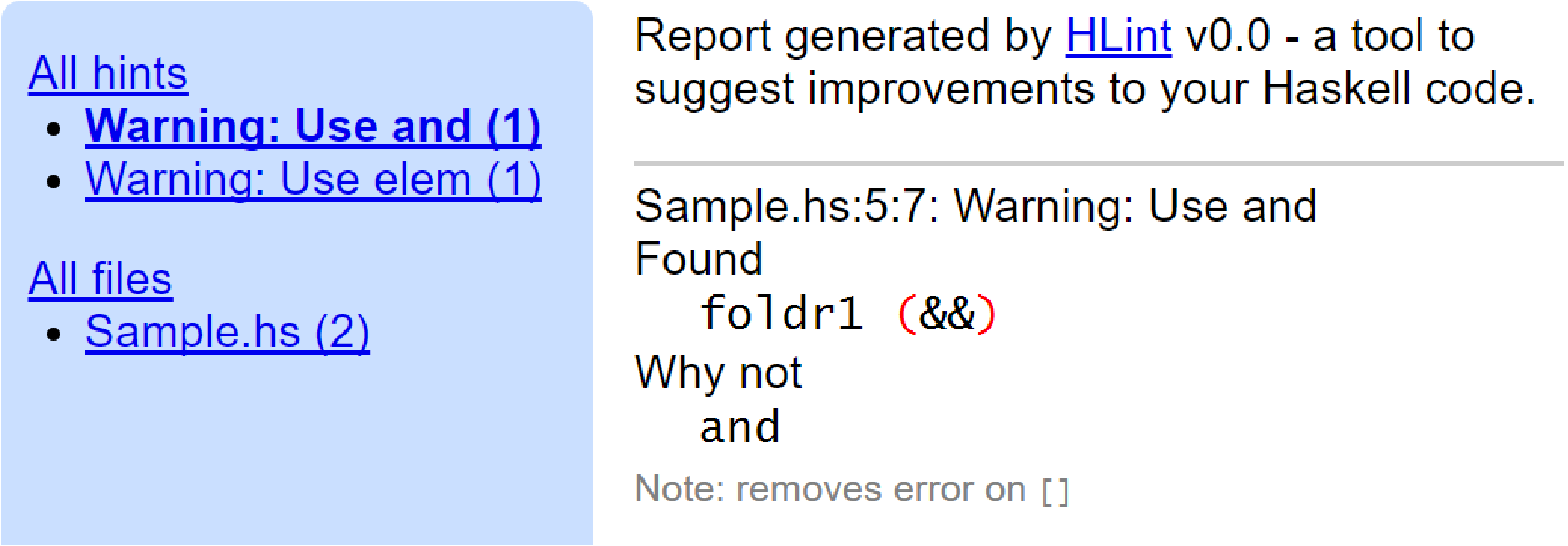
Method: Convert Haskell98 to TRS. Apply cutting-edge TRS approach.

Caveats: Not in terms a Haskeller understands. Haskell98 only. No community adoption.

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

HLint

- A tool for suggesting stylistic improvements



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 `[]`

Example hints

- Redundant language extensions

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

- Use of mapM instead of mapM_
- Simple sugar functions (concatMap)

Overall workings

- 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}

Detailed workings

```
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)]
```

Where does it go wrong?

- Monomorphism restriction
  - `foo x = bar x`
- RankN polymorphism
  - `foo (g x y z)`
- Operator precedence/overriding
  - `g x + g x ^^^ f y`
- Seq strictness breaks lots of laws
  - `\x -> f x`

HLint summary

Goal: Make the code prettier. Mopping up after refactorings.

Method: File-at-a-time, some hardcoded suggestions, some driven by a rule config.

Caveats: Can’t deal with CPP. Pretty is subjective. No types. No scope info. Lots of “close but not quite” rules. But see comparable tools in other languages…

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

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)

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

Hi file data type

```
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
```

Caveats

- **Data.Coerce** If you use Data.Coerce.coerce the constructors for the data type must be in scope, but if they aren't used anywhere other than automatically by coerce then Weeder will report unused imports.

- **Declaration QuasiQuotes** If you use a declaration-level quasi-quote then weeder won't see the use of the quoting function, potentially leading to an unused import warning, and marking the quoting function as a weed.

Weeder summary

Goal: Find code/imports that are not required.

Method: Pull apart the .hi files and reuse that information with some analysis predicates.

Caveats: Can’t deal with CPP. Sometimes limited by the .hi files.

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
```

Call to arms!

- Static analysis is cool, we should do more of it
  - Generally, whole program is easiest to prototype
  - GHC doesn’t make that very easy…
  - Someone want to make it easy?

- Static analysis can give lots of great insights
  - In C/C++/Java there’s a cottage industry
  - Are we spoilt by types?

How many do you use?

- Type safety
- GHC warnings
- HLint style checker
- Weeder dead export detector
- LiquidHaskell refinement type analysis
- AProVE termination checking
- Catch error free checker

… others …?