Transformation and Analysis of Haskell Source Code

Neil Mitchell

www.cs.york.ac.uk/~ndm

Why Haskell?

- Functional programming language
  - Short, beautiful programs
- Referential transparency
  - Easier to reason about and manipulate
- Lazy
  - Beta-reduction holds
  - Can inline easily

Goals

- Transform
  - Make transformations concise
- Optimise
  - Make programs execute faster
- Analyse
  - Generate proofs of safety
  - Pinpoint unsafe aspects

Haskell Source

```
data Core = Core [Data] [Func]
data Func = Func Name [Args] Expr
data Expr = Let [(Name,Expr)] Expr
          | App Expr [Expr]
          | Case Expr [(Expr,Expr)]
          | Var Name
          | Fun Name
          | Con Name
          | -- lots more
```

Find all functions

```
f :: Expr → [String]
f (Let x y) = concatMap (f.snd) x ++ f y
f (App x y) = f x ++ concatMap f y
f (Case x y) = f x ++
    concatMap f [[a,b] | (a,b) <- y]
f (Fun x) = [x]
-- lots more cases
```

Removing Boilerplate

```
uniplate x = [x | Fun x <- universe x]

syb x = everything (++) ([] `mkQ` getFun)
   where getFun (Fun x) = [x]
         getFun _       = []

compos :: Tree c -> [Name]
compos (Fun x) = [x]
compos x = composOpFold [] (++) compos x
```

Generic Traversals

- Reduce the quantity of code
- Make programs more readable
- Make code more robust

My extra goal:

- Use Haskell 98 (no scary types)

Fewer Extensions

- Uniplate (GHC, Yhc, nhc, Hugs – H98)
  - Advanced features require Hugs/GHC – H'
- SYB (GHC 6.4+ only)
  - Requires rank-2 types
  - Data instances in the compiler
- Compos (GHC 6.6+ only)
  - Rank-2 types
  - GADT's (very unportable)

Central Idea

```
class Uniplate a where
  uniplate :: a → ([a], [a] → a)
  uniplate x = (get,set)
```

- Children
  - maximal contained items of the same type
  - Get the children
  - Set a new set of children

Traversals

- Queries
  - Extract information out
  - Already seen an example
- Transformations
  - Create a modified value
  - Some change

Removing Let's

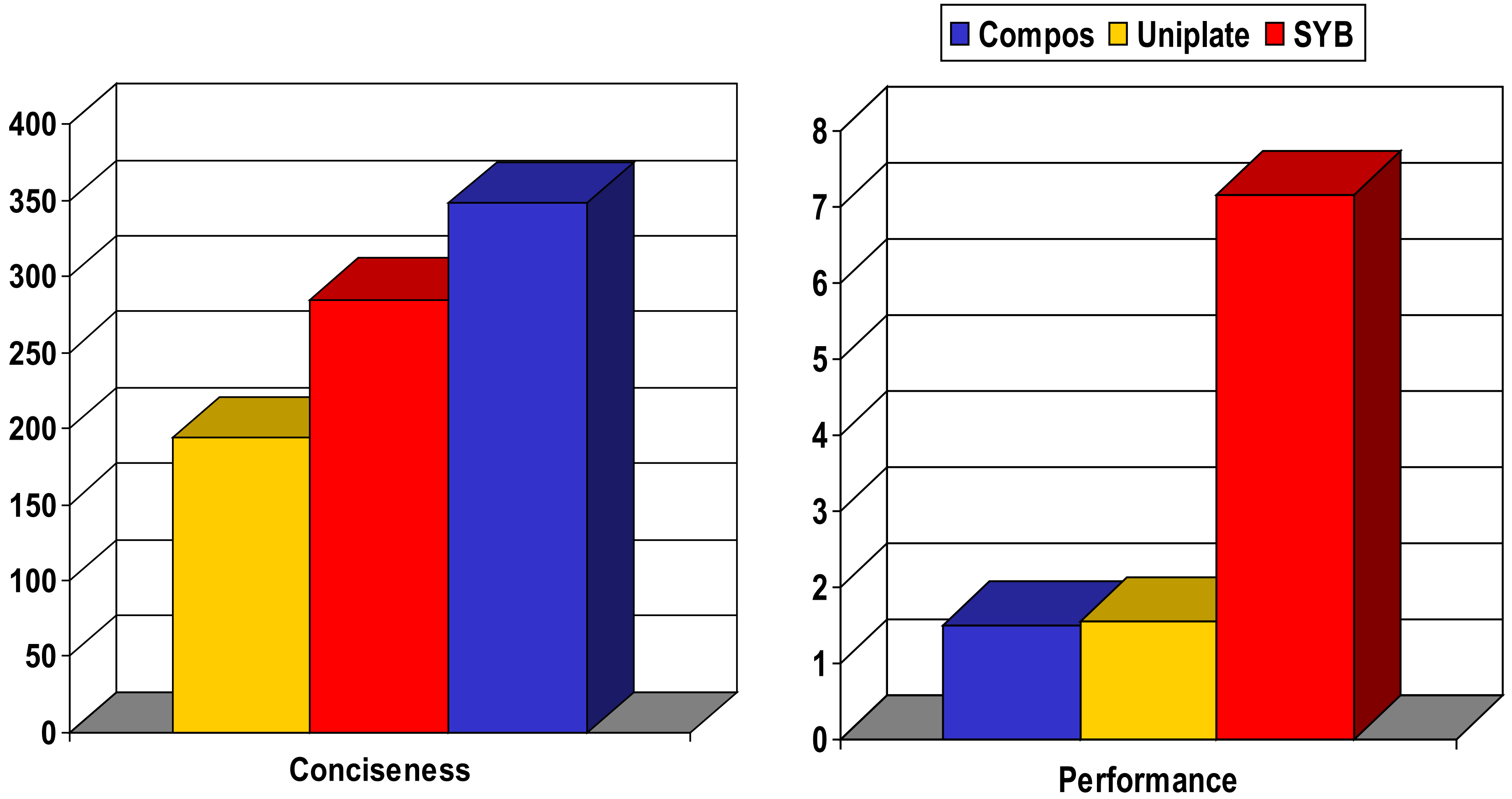
- The operation

```
removeLet (Let bind x) = Just $
            substitute bind x
removeLet _ = Nothing
```

- The transformation

```
removeAllLet = rewrite removeLet
```

Concise and Fast



Uniplate in the World

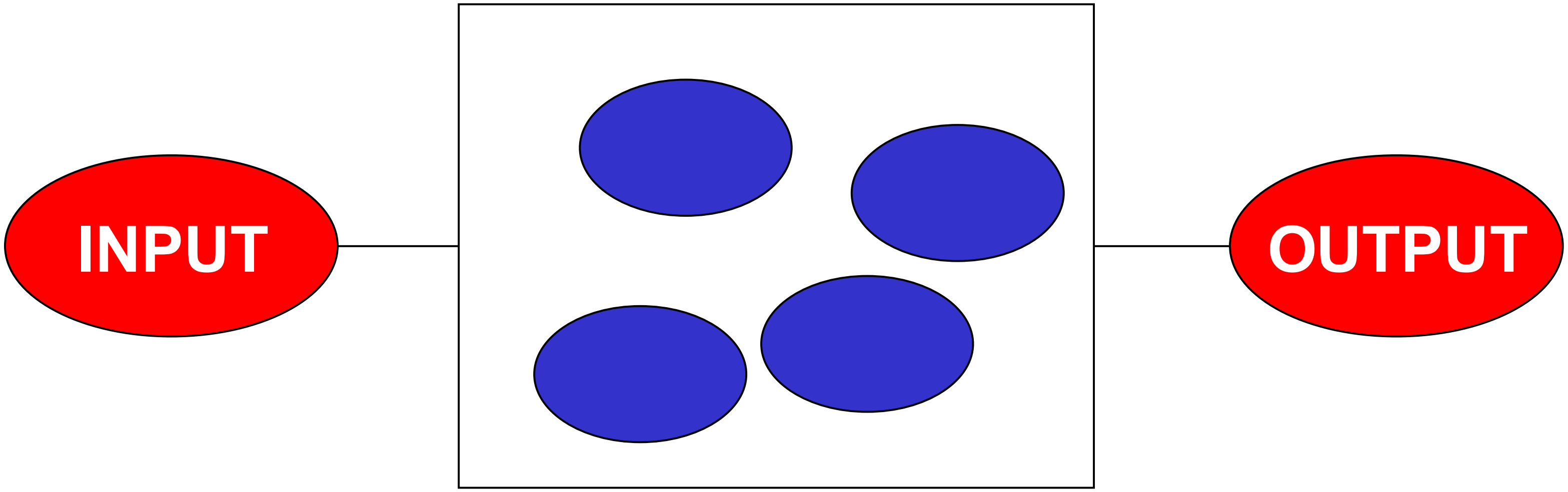
- My uses
  - Optimiser, Analyser
  - Hoogle (Haskell search engine)
  - Dr Haskell (Haskell tutorial tool)
- Matt Naylor's uses (see next)
  - Reach, Reduceron
- Several other projects
  - Configurations, QHC, Javascript generator…

Optimisation

- Goal
  - Haskell code should be as fast a C
  - Code should remain high-level
- Central idea
  - Remove *overhead*
  - Remove intermediate steps

Intermediate Steps

- Eliminate values (data/functions)
  - `length [1..n]`
  - `not (not x)`



The Method

- Remove higher order functions
  1. Either: using specialise/inline rule
  2. Or: using over/under staturation rules
- Convert data to functions
  - Church encoding
- Remove higher order functions

- Leaves little data or functions

First Order Haskell

- Remove lambda abstractions (lambda lift)
- Leaving only partial application/currying

```
odd = (.) not even
(.) f g x = f (g x)
```

- Generate templates (specialised bits)

Oversaturation

```
f x y z, where arity(f) < 3
```

```
main = odd 12
```

```
<odd _> x = (.) not even x
main = <odd _> 12
```

Undersaturation

```
f x (g y) z, where arity(g) > 1

<odd _> x = (.) not even x

<(.) not even _> x = not (even x)
<odd _> x = <(.) not even _> x
```

Special Rules

`let z = f x y, where arity(f) > 2`

- (let-under) rule
- inline `z`, after sharing `x` and `y`

`d = Ctor (f x) y, where arity(f) > 1`

- (ctor-under) rule
- inline d
- The “dictionary” rule

Standard Rules

let x = (**let** y=z **in** q) **in** … let/let

case (**let** x=y **in** z) **of** … case/let

case (**case** x **of** …) **of** … case/case

(**case** x **of** …) y z app/case

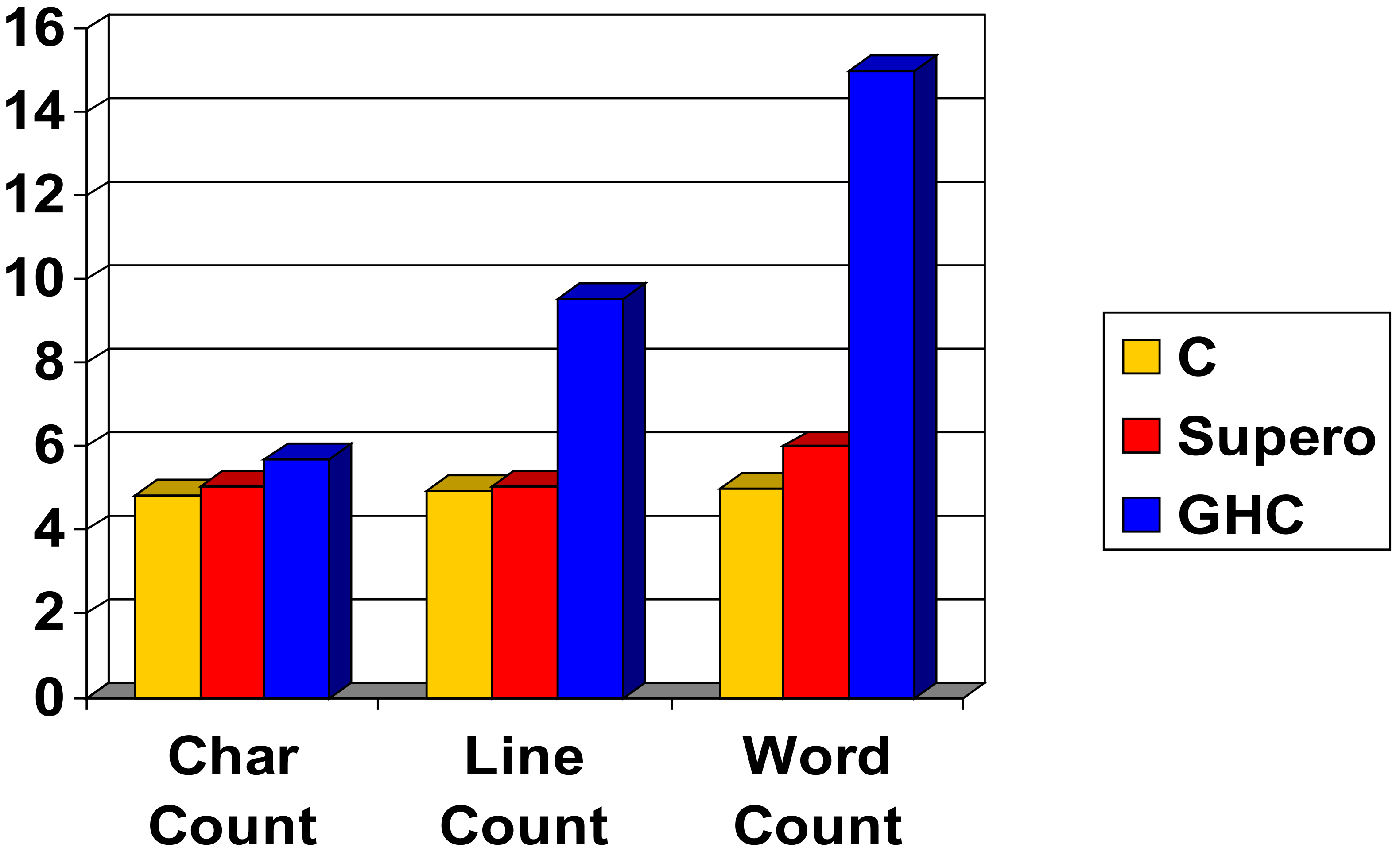
case C x **of** … case/ctor

Church Encoding

```
data List a =
    Nil                    nil        = \n c → n
  | Cons a (List a)        cons x y = \n c → c x y

len x = case x of          len x = x
   Nil → 0                     0
   Cons y ys →                 (\y ys →
      1 + len ys                 1 + len ys)
```

The Preliminary Results



Future Work

- Refactoring
  - Requires extensible transformations
  - Needs to integrate with GHC's IO Monad
- More Benchmarks
- Proofs
  - Correctness
  - Laziness/strictness preserving
  - Termination

Analysis: Pattern matching

- Haskell programs may crash at runtime
  - Pattern-match errors are quite common

```
head “neil” = ‘n’
head []     = ⊥
```

- Can get very complex

The Goal

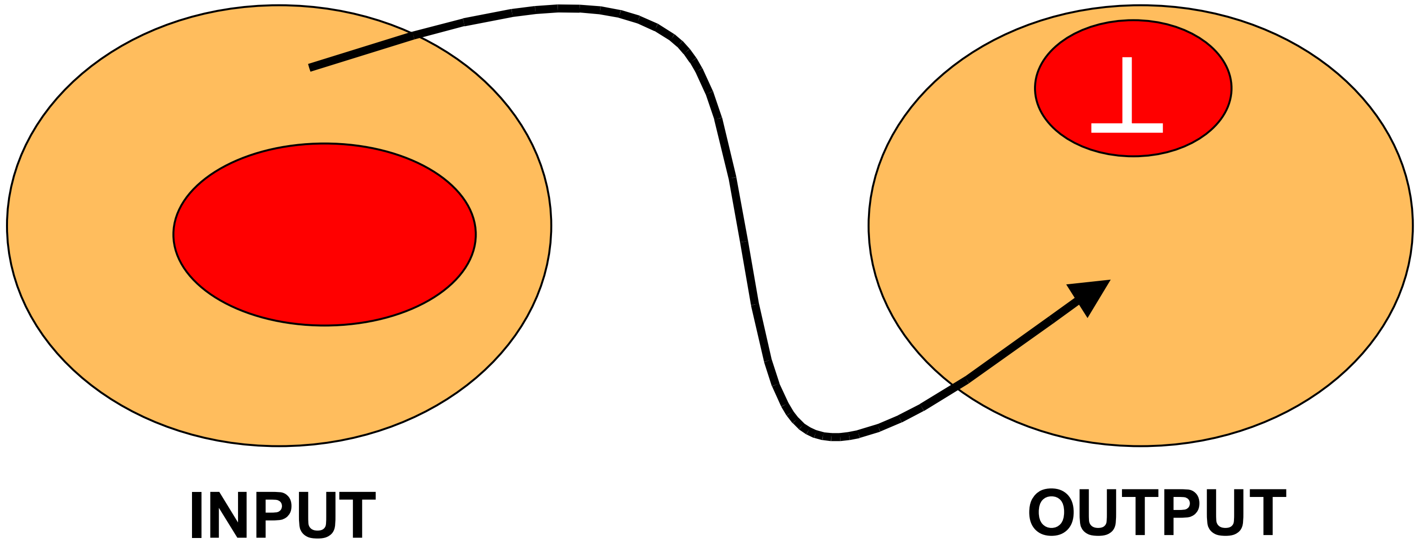
- Statically prove the absence of pattern-match errors
  - Be conservative
  - Generate a "proof" of safety
- Entirely automatic
  - No annotations
- Practical
  - Catch tool has been released

A Pattern-Match Error

- In Haskell you match a value with a set of patterns
  - Patterns *do not* have to be exhaustive
- A "default" pattern is inserted, calling error
- Analysis:
  - Can the error case be reached?
  - What are the preconditions on functions?

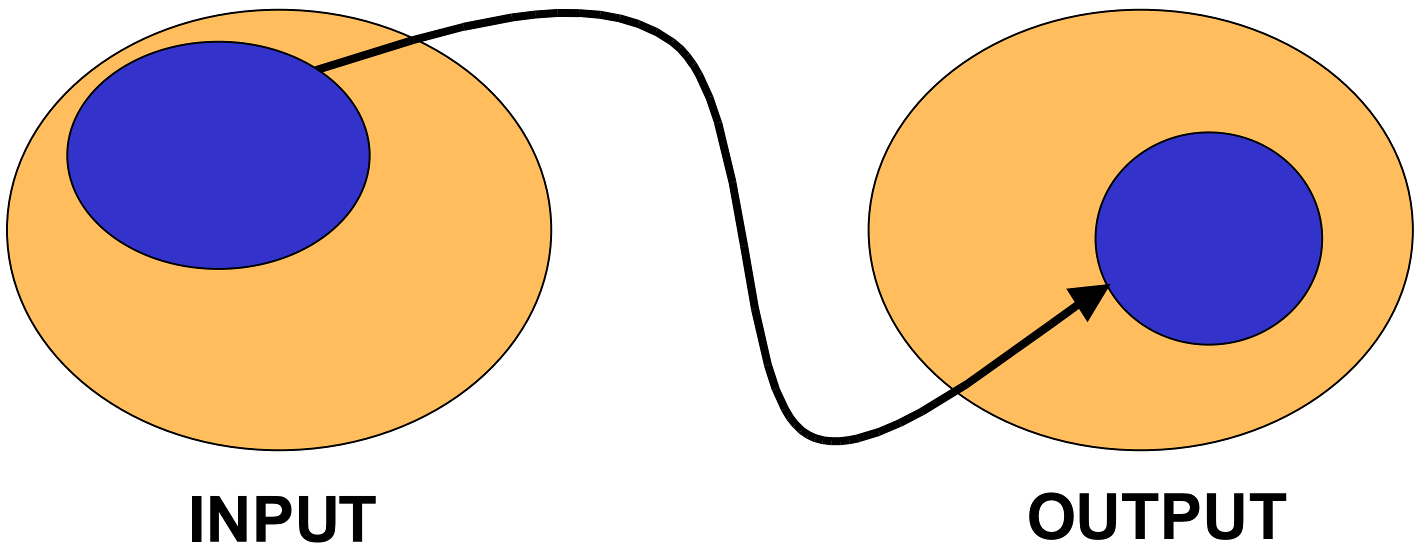
Preconditions

- Calculate a precondition on the input
  - Sufficient to ensure the output is never ⊥



Properties

- Calculate a precondition on the input
  - Sufficient to ensure a particular output



Automatic inference

- Can automatically infer the properties and preconditions
  - Precondition of error is False
  - Precondition of an expression can be expressed as preconditions of its parts
  - Properties are used for calculating preconditions on function results

Constraints

- All based on the partitioning of a function
  - Constraints on values are used
- BP constraints – list of patterns
- RE constraints – use regular expressions
- MP constraints – clever list of patterns
  - Used in Catch

MP Constraints

- Haskell has recursive data structures

```
data List α = Nil | Cons α (List α)
```

- MP is: non-recursive ♦ recursive
  - Non-recursive represents top-level values
  - Recursive represents *all* other values

```
(Cons _ *) ♦ (Cons _ * | Nil)
```

MP Examples

`(Cons _ *) ♦ (Cons _ * | Nil)`

- Non-empty list

`(Cons True *) ♦ (Cons True *)`

- Infinite list of True

`True ♦ _`

- The value True

`(Zero | One | Pos) ♦ _`

- A natural number

Key MP Property

- Any proposition on MP constraints of one variable is equivalent to one MP constraint

```
(True ♦ _) ∨ (False ♦ _) = (_ ♦ _)
```

  - Works in all cases

- Results in simplification, and fast analysis

A real-world program

- XMonad: An window manager for X
  - Lots of low-level details
  - A single pure core module “StackSet”
  - No special annotations
- Running Catch:

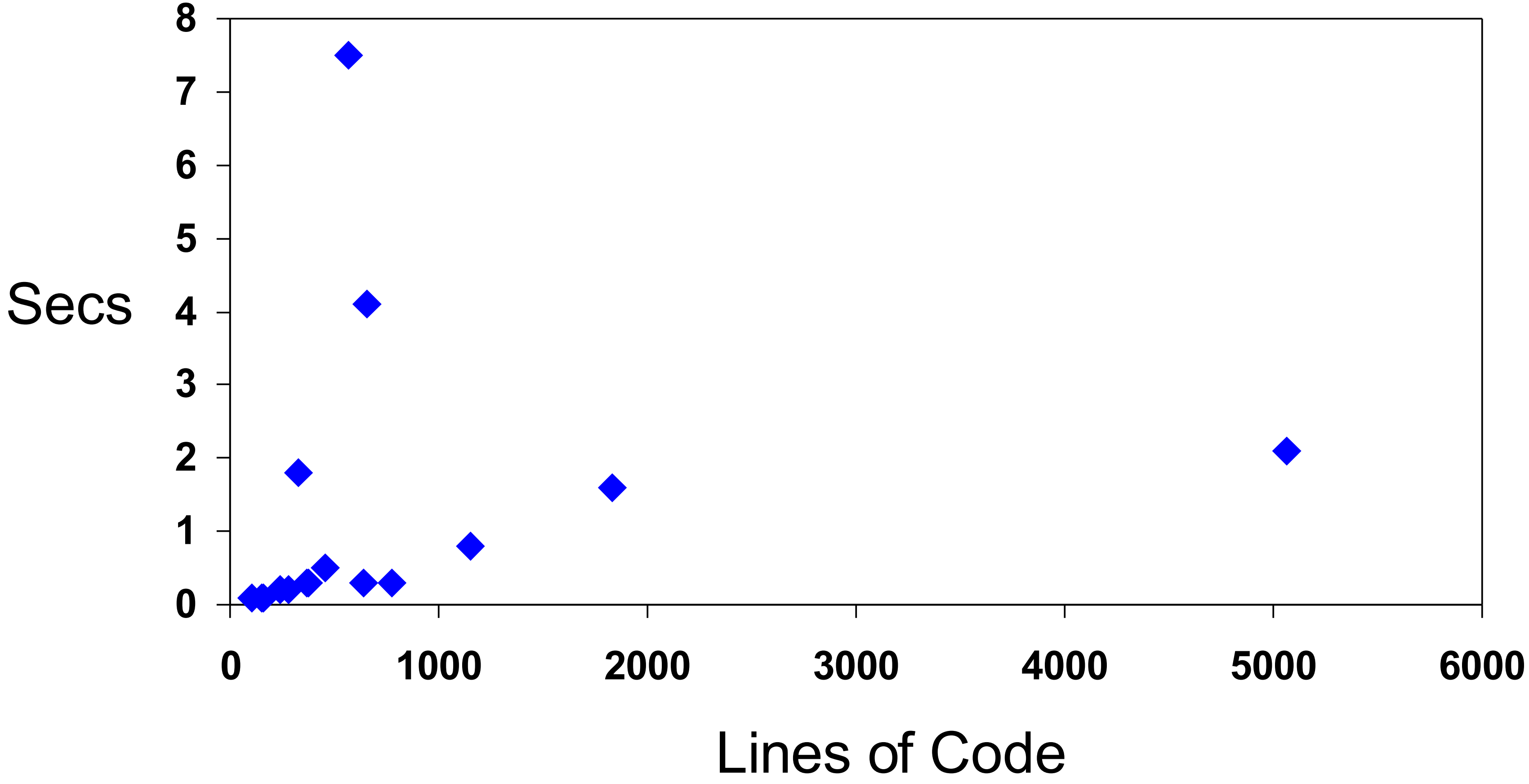
```
$ catch StackSet.hs --quiet
Checking StackSet
14 error calls found
All proven safe
```

One XMonad sample

```
views n
    | n < 1 = …
    | otherwise = h : g t
   where (h:t) = [f i | i ← [1..n]]
```

- This is safe for Int, Integer
- Not safe for all numeric types

Analysis Times



Catch in the Real World

- XMonad was proven safe
  - Developers have started using it as standard
- FilePath library checked
- FiniteMap library checked
- HsColour program checked
  - Found 3 previously unknown, genuine bugs

Conclusions

- Transform: Uniplate
  - Concise and fast code
  - Without scary types (beginner friendly)
- Optimise: Supero
  - Fast code, with reasonable compile times
- Analyse: Catch
  - Can automatically check real world programs
  - Can find genuine bugs