Why Haskell?

• Functional programming language
  ▪ Short, beautiful programs

• Referential transparency
  ▪ Easier to reason about and manipulate

• Lazy
  ▪ Beta-reduction holds
  ▪ Can inline easily
• Transform
  ▪ Make transformations concise
• Optimise
  ▪ Make programs execute faster
• Analyse
  ▪ Generate proofs of safety
  ▪ Pinpoint unsafe aspects
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

\[
f :: \text{Expr} \rightarrow \text{[String]}
\]
\[
f \ (\text{Let} \ x \ y) = \text{concatMap} \ (f.\text{snd}) \ x \ +\ + \ f \ y
\]
\[
f \ (\text{App} \ x \ y) = f \ x \ +\ + \ \text{concatMap} \ f \ y
\]
\[
f \ (\text{Case} \ x \ y) = f \ x \ +\ +
\]
\[
\quad \text{concatMap} \ f \ [[a,b] \mid (a,b) <- y]
\]
\[
f \ (\text{Fun} \ x) = [x]
\]
\[
-- \text{lots more cases}
\]
Removing Boilerplate

uniplate \(x\) = \([x \mid \text{Fun } x \leftarrow \text{universe } x]\)

\[
\text{syb } x = \text{everything } (++) ([[] `mkQ` getFun) \\
\quad \text{where getFun (Fun } x) = [x] \\
\quad \text{getFun } _\_ = []
\]

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)
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

\[
\text{removeLet } (\text{Let } \text{bind } x) = \text{Just } \$
\text{substitute } \text{bind } x
\text{removeLet } _ = \text{Nothing}
\]

• The transformation

\[
\text{removeAllLet } = \text{rewrite } \text{removeLet}
\]
Concise and Fast

Graphs showing conciseness and performance for different options:
- Conciseness: Compos, Uniplate, SYB
- Performance: Compos, Uniplate, SYB

Graphs indicate SYB has the highest conciseness and performance.
Uniplate in the World

- My uses
  - Optimiser, Analyser
  - Hoogo (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 saturation 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

\[ f(x, y, z), \text{ where } \text{arity}(f) < 3 \]

main = \texttt{odd} 12

\(<\text{odd } _> \ x = (.) \text{ not even } x\)

main = \texttt{<odd } _> 12
Undersaturation

\[ f \ x \ (g \ y) \ z, \ \text{where} \ \text{arity}(g) > 1 \]

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

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

**let** $z = f \, x \, y$, **where** $\text{arity}(f) > 2$

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

$d = \text{Ctor} \,(f \, x) \, y$, **where** $\text{arity}(f) > 1$

- (ctor-under) rule
- inline $d$
- The “dictionary” rule
let \( x = (\text{let } y=z \text{ in } q) \text{ in } \ldots \) let/let

\text{case} (\text{let } x=y \text{ in } z) \text{ of } \ldots \text{ case/let}

\text{case} (\text{case } x \text{ of } \ldots) \text{ of } \ldots \text{ case/case}

\text{(case } x \text{ of } \ldots) y z \text{ app/case}

\text{case } C x \text{ of } \ldots \text{ case/ctor}
**data** List a =

    Nil
  | Cons a (List a)

len x = \texttt{case} x \texttt{of}

    Nil → 0
  Cons y ys →
    1 + len ys

\[\text{nil} = \lambda n \ c \rightarrow n\]
\[\text{cons} \ x \ y = \lambda n \ c \rightarrow c \ x \ y\]

len x = x

\[\lambda y \ ys \rightarrow (1 + \text{len} \ ys)\]
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 $\perp$
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

\[ \text{data} \quad \text{List} \ \alpha = \text{Nil} \mid \text{Cons} \ \alpha \ (\text{List} \ \alpha) \]

• MP is:  non-recursive ♦ recursive
  ▪ Non-recursive represents top-level values
  ▪ Recursive represents \textit{all} other values

\[ (\text{Cons} \ _ \ *) \ ♦ \ (\text{Cons} \ _ \ * \mid \text{Nil}) \]
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

\[(\text{True } \lor \_) \lor (\text{False } \lor \_) = (\_ \lor \_)\]
- 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

Lines of Code vs. Secs

Graph showing the relationship between lines of code and 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