Unfailing Haskell:

Stopping Pattern Match Errors

Neil Mitchell and Colin Runciman

University of York, UK

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

Is this safe?

```
risers :: Ord a => [a] -> [[a]]
risers [] = []
risers [x] = [[x]]
risers (x:y:etc) = if x <= y
                   then (x:s):ss
                   else [x]:(s:ss)
    where (s:ss) = risers (y:etc)

> risers [1,2,3,1,2]
[[1,2,3],[1,2]]
```

Answer: Yes

Reasoning:

`(s:ss) = risers (y:etc)`

∴ `risers (_:_) = (_:_)`

By case analysis:

`risers [x] = [[x]]`

`risers (x:y:etc) =`

`(x:s):ss` *or* `[x](s:ss)`

Is this safe?

```
transpose :: [[a]] -> [[a]]
transpose x@((_:_):_) =
  map head x :
  transpose (map tail x)
transpose x = []

> transpose [“123”,”456”,”789”]
[“147”,”258”,”369”]
```

Answer: No

Try:

```
transpose [“123”,”45”]
```

```
Program error:
pattern match failure:
head []
```

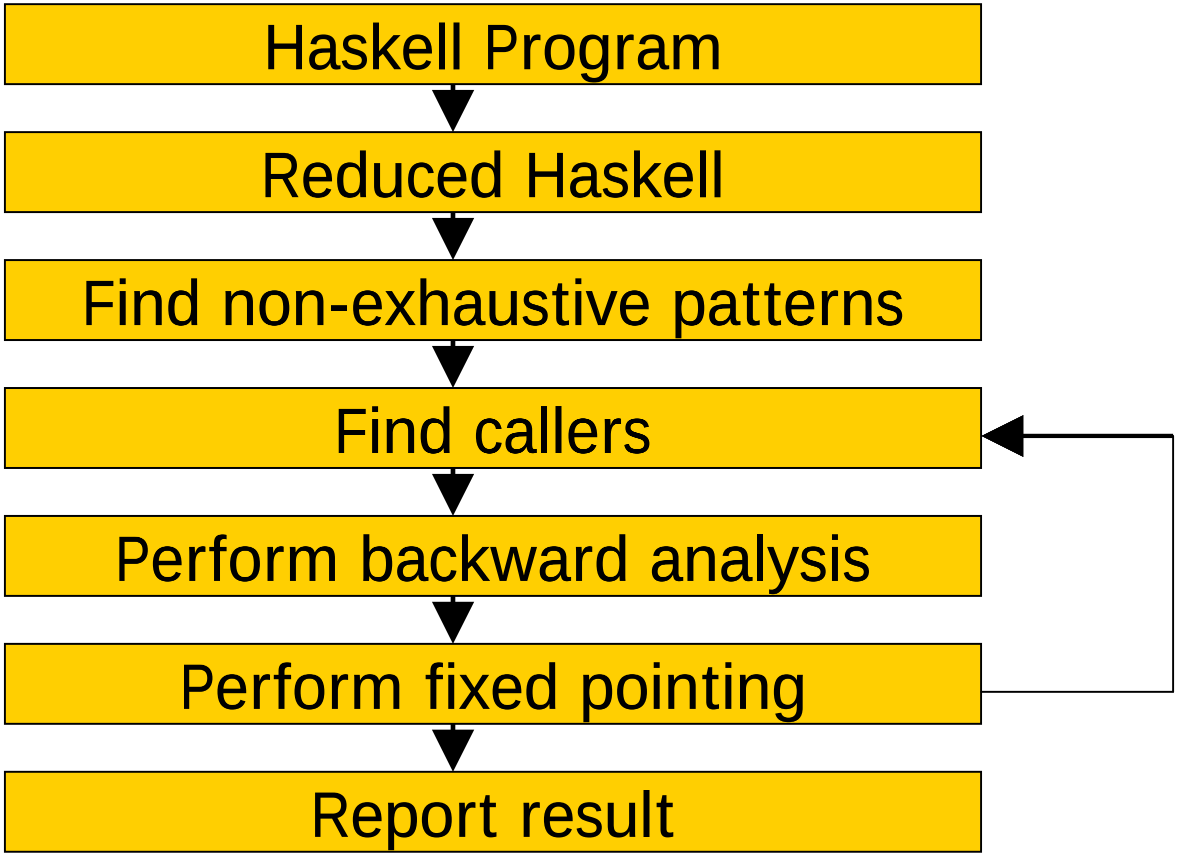
The checker

- Takes reduced Haskell
- Generates a proof that a program will not crash with a case error
- Uses static analysis
- It is conservative

Reduced Haskell

```
data List = Cons Cons_1 Cons_2 | Nil

head @1 = case @1 of Cons -> @1.Cons_1

map @1 @2 = case @2 of
    Nil -> Nil
    Cons -> Cons (@1 @2.Cons_1) (map @1 @2.Cons_2)

reverse @1 = rev @1 Nil

rev @1 @2 = case @1 of
    Nil -> @2
    Cons -> rev @1.Cons_2 (Cons @1.Cons_1 @2)
```

λ An overview



Haskell Program

Reduced Haskell

Find non-exhaustive patterns

Find callers

Perform backward analysis

Perform fixed pointing

Report result

Constraints, intro by example

```
head (x:xs) = x
head@1{:}

fromJust (Just x) = x
fromJust@1{Just}

foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
foldr1@2{:}
```

Constraints with paths

```
mapHead [] = []
mapHead (x:xs) =
    head x : mapHead xs
mapHead@1.*tail.head{:}

mapHead@1.head{:} ^
mapHead@1.tail.head{:} ^
mapHead@1.tail.tail.head{:} ^ …
```

Finding a fixed point

- In `mapHead`
  - `@1 ← @1.tail`
- Condition, ignoring recursive call
  - `mapHead@1.head{:}`
- Rule
  - `@n ← @n.path` ⇒ `@n`$_{\infty}$ `= @n.*path`
  - `mapHead@1.*tail.head{:}`

Infinite constraints

```
revHead x = mapHead (reverse x)
revHead@1.*tail{:} ∨
revHead@1.*tail.head{:}

revHead@1{:} ∧
revHead@1.tail{:} ∧
revHead@1.tail.tail{:} ∧ …
```

`revHead@1` is infinite

Backward Analysis

- `head@1{:}`, applies to head
- `f @1 = head (init @1)`
- `(init f@1){:}`, applies to...?

- Backward analysis

`Constraint Expr -> Constraint Arg`

- `f@1.tail{:}`

Higher Order Functions

- They complicate analysis
- Can be removed in some cases
  - `map, foldr, foldl, filter ...`

```
test n x = map (f n) x

mapf n []     = []
mapf n (x:xs) = f n x : mapf n xs
```

Laziness

- A function may be safe lazily, but not strictly

```
safeTail X = cond (null x) [] (tail x)
cond c t f = if c then t else f
```

- Can inline

```
safeTail x = if null x then [] else tail x
```

Real Programs

- Has been tested on real programs
  - Clausify – propositional simplifier
  - Adjoxo – adjudicate XOX games
  - Soda – word search solver
- Minor modifications were needed for success
- Apart from Clausify

Conclusions

- Manages to prove a function safe wrt pattern match errors, even if incomplete patterns
- Algorithm identified and implemented
- Good initial results
- Future Work
  - Improve results
  - Better support for full Haskell

The Rules

$$\varphi(\texttt{arg}\ \ n, r, c) \rightarrow \langle \mathrm{qual}(n), r, c \rangle$$

$$\frac{\varphi(E, r, c) \rightarrow \langle E', r', c' \rangle}{\varphi(\texttt{sel}\ \ E\ \ C\ \ m, r, c) \rightarrow \langle E', C_m.r', c' \rangle}$$

$$\frac{\varphi(E_1, \frac{\partial r}{\partial C_1}, c) \rightarrow E'_1, \cdots, \varphi(E_n, \frac{\partial r}{\partial C_n}, c) \rightarrow E'_n}{\varphi(\texttt{make}\ \ C\ \ E_1 \cdots E_n) \rightarrow (\lambda \in L(r) \Rightarrow C \in c) \wedge E'_1 \wedge \cdots \wedge E'_n}$$

$$\frac{\begin{array}{c}\varphi(\mathcal{D}(E_0), r, c) \rightarrow P \\ P[\langle \texttt{arg}\ \ 1, r_1, c_1 \rangle / \varphi(E_1, r_1, c_1), \cdots, \langle \texttt{arg}\ \ n, r_n, c_n \rangle / \varphi(E_n, r_n, c_n)] \rightarrow P' \end{array}}{\varphi(\texttt{apply}\ \ E_0\ \ E_1 \cdots E_n, r, c) \rightarrow P'}$$

$$\frac{\begin{array}{c} C = \{x | \mathrm{type}(x) = \mathrm{type}(C_1)\} \\ P = (\varphi(E, \lambda, C \backslash C_1) \vee \varphi(E_1, r, c)) \wedge \cdots \wedge (\varphi(E, \lambda, C \backslash C_n) \vee \varphi(E_n, r, c)) \end{array}}{\varphi(\texttt{case}\ \ E\ \ \texttt{of}\ \ \{C_1\texttt{->}E_1\texttt{;}\ \cdots \texttt{;}\, C_n\texttt{->}E_n\}, r, c) \rightarrow P}$$