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?

\[ \text{risers} :: \text{Ord} \ a \Rightarrow [a] \to [[[a]]] \]
\[ \text{risers} \ [] = [] \]
\[ \text{risers} \ [x] = [[[x]]] \]
\[ \text{risers} \ (x:y:etc) = \text{if } x \leq y \]
\[ \quad \text{then } (x:s):ss \]
\[ \quad \text{else } [x]:(s:ss) \]
\[ \quad \text{where } (s:ss) = \text{risers} \ (y:etc) \]

> \text{risers} \ [1,2,3,1,2]

[[[1,2,3],[1,2]]]
Answer: Yes

Reasoning:

\[(s:ss) = risers(y:etc)\]
\[\therefore risers(_:_)(=_:_)=(_:_)\]

By case analysis:

\[\text{risers}[x] = [[x]]\]
\[\text{risers}(x:y:etc) = \]
\[(x:s):ss \quad \text{or} \quad [x](s:ss)\]
Is this safe?

\[
\text{transpose :: } [[a]] \rightarrow [[a]] \\
\text{transpose } x \mathbin{@} (_:_):_ = \\
\quad \text{map head } x : \\
\quad \text{transpose (map tail } x) \\
\text{transpose } x = []
\]

> transpose ["123", "456", "789"]
[["147", "258", "369"]


Answer: No

Try:

\[
\text{transpose ["123", "45"]}
\]

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 Cons₁ Cons₂ | Nil

head @1 = case @1 of Cons -> @1.Cons₁

map @1 @2 = case @2 of
    Nil -> Nil
    Cons -> Cons (@1 @2.Cons₁) (map @1 @2.Cons₂)

reverse @1 = rev @1 Nil

rev @1 @2 = case @1 of
    Nil -> @2
    Cons -> rev @1.Cons₂ (Cons @1.Cons₁ @2)
An overview

1. Haskell Program
2. Reduced Haskell
3. Find non-exhaustive patterns
4. Find callers
5. Perform backward analysis
6. Perform fixed pointing
7. Report result
Constraints, intro by example

\[ \text{head} \ (x:xs) = x \]
\[ \text{head@1} \ \&\{ \text{::} \} \]

\[ \text{fromJust} \ (\text{Just} \ x) = x \]
\[ \text{fromJust@1} \ \&\{ \text{Just} \} \]

\[ \text{foldr1} \ f \ [x] = x \]
\[ \text{foldr1} \ f \ (x:xs) = f \ x \ (\text{foldr1} \ f \ xs) \]
\[ \text{foldr1@2} \ \&\{ \text{::} \} \]
Constraints with paths

\[
\text{mapHead} \; [\;] = \; [\;]
\]

\[
\text{mapHead} \; (x : xs) =
\]

\[
\text{head} \; x : \; \text{mapHead} \; xs
\]

\[
\text{mapHead@1} \; \text{.*tail} \; \text{.head}[:,:]
\]

\[
\text{mapHead@1} \; \text{.head}[:,:} \; ^
\]

\[
\text{mapHead@1} \; \text{.tail} \; \text{.head}[:,:} \; ^
\]

\[
\text{mapHead@1} \; \text{.tail} \; \text{.tail} \; \text{.head}[:,:} \; ^\ldots
\]
Finding a fixed point

- In `mapHead`
  - `@1 ← @1.tail`

- Condition, ignoring recursive call
  - `mapHead@1.head{: }

- Rule
  - `@n ← @n.path ⇒ @n_∞ = @n.*path`
    - `mapHead@1.*tail.head{: }`
Infinite constraints

\( \text{revHead } x = \text{mapHead (reverse x)} \)

\( \text{revHead@1.*tail{:}} \lor \text{revHead@1.*tail.head{:}} \)

\( \text{revHead@1{:}} \uparrow \)

\( \text{revHead@1.tail{:}} \uparrow \)

\( \text{revHead@1.tail.tail{:}} \uparrow \ldots \)

\( \text{revHead@1 is infinite} \)
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` ...

\[
\text{test } n \ x = \text{map} (f \ n) \ x
\]

\[
\text{mapf} \ n \ [ ] = [ ]
\]
\[
\text{mapf} \ n \ (x:xs) = f \ n \ x : \text{mapf} \ n \ xs
\]
Laziness

- A function may be safe lazily, but not strictly

\[
\text{safeTail } X = \text{cond} \ (\text{null } x) \ [] \ (\text{tail } x)
\]
\[
\text{cond } c \ t \ f = \text{if } c \ \text{then } t \ \text{else } f
\]

- Can inline

\[
\text{safeTail } x = \text{if null } x \ \text{then } [] \ \text{else tail } x
\]
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

\[ \varphi(\text{arg } n, r, c) \rightarrow \langle \text{qual}(n), r, c \rangle \]

\[ \varphi(E, r, c) \rightarrow \langle E', r', c' \rangle \]

\[ \varphi(\text{sel } E \ C \ m, r, c) \rightarrow \langle E', C_m, r', c' \rangle \]

\[ \varphi(E_1, \frac{\partial r}{\partial C_1}, c) \rightarrow E'_1, \cdots, \varphi(E_n, \frac{\partial r}{\partial C_n}, c) \rightarrow E'_n \]

\[ \varphi(\text{make } C \ E_1 \cdots E_n) \rightarrow (\lambda \in \text{I}(r) \Rightarrow C \in c) \land E'_1 \land \cdots \land E'_n \]

\[ \varphi(\text{do}(E_0), r, c) \rightarrow P \]

\[ P[\langle \text{arg } 1, r_1, c_1 \rangle/\varphi(E_1, r_1, c_1), \cdots, \langle \text{arg } n, r_n, c_n \rangle/\varphi(E_n, r_n, c_n)] \rightarrow P' \]

\[ \varphi(\text{apply } E_0 \ E_1 \cdots E_n, r, c) \rightarrow P' \]

\[ C = \{ x | \text{type}(x) = \text{type}(C_1) \} \]

\[ P = (\varphi(E, \lambda, C \setminus C_1) \lor \varphi(E_1, r, c)) \land \cdots \land (\varphi(E, \lambda, C \setminus C_n) \lor \varphi(E_n, r, c)) \]

\[ \varphi(\text{case } E \text{ of } \{ C_1 \rightarrow E_1; \cdots; C_n \rightarrow E_n \}, r, c) \rightarrow P \]