CATCH: **Case** and Termination Checker for Haskell

Neil Mitchell
(Supervised by Colin Runciman)

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

- Take a Haskell program
- Analyse it
- Prove statically that there are no "unsafe pattern matches"
- No additional user work

- Termination – not in 18 minutes!
Is this safe?

\[
\begin{align*}
\text{risers} & \; [\;] \; = \; [] \\
\text{risers} & \; [\; x \;] \; = \; [[[x]]] \\
\text{risers} & \; (x:y:etc) \; = \\
& \quad \text{if} \; x \; \leq \; y \\
& \quad \text{then} \; (x:s):ss \\
& \quad \text{else} \; [x]:(s:ss) \\
& \quad \text{where} \; (s:ss) \; = \; \text{risers} \; (y:etc)
\end{align*}
\]
\[ \text{risers } [] = [] \]
\[ \text{risers } [x] = [[[x]]] \quad \text{-- } ((x:[]):[]) \]
\[ \text{risers } (x:y:etc) = \]
\[ \text{if } x \leq y \]
\[ \text{then } (x:s):ss \]
\[ \text{else } [x]:(s:ss) \]
\[ \text{where } (s:ss) = \text{risers } (y:etc) \]
How does Catch work?

- Transform to reduced Haskell
- Apply transformations on reduced Haskell
- Generate a condition for case safety
- Propagate this condition
- Figure out if the precondition is True
Pattern Matches

\(\text{let } (a,b) = y \text{ in } (b,a)\)

\(\text{f } x \mid \text{null } x = []\)
\(\text{otherwise } = \text{tail } x\)

\(\text{f } x = \text{ys}\)
\(\text{where } (y:ys)\)

\(\text{case } x \text{ of}\)
\(\text{[ ] } \rightarrow \text{True}\)
\((a:b) \rightarrow a\)

\(\text{f } x = \text{ys}\)
\(\text{where } (y:ys)\)

\(\text{do } (x:xs) \leftarrow f y\)
\(\text{return } xs\)

\(\text{f } [x] = x\)

\(\text{\(\Lambda\)}\)
Reduced Haskell

- Only simple case, functions, applications, constructors

```haskell
data [] = [] | (:) hd tl

map f xs =
  case xs of
    [] -> []
    (_:_) -> f xs.hd : map f xs.tl
```
Generating Reduced Haskell

- Fully automatic
- Uses Yhc’s Core language
  - Yhc is a fork of nhc98
  - Specify –core or –corep to see it
- Some additional transformations
  - Remove a few let’s
- By the end, reduced Haskell
Transformations

- About 8 are applied
- Reachability
  - Eliminate dead code
- Arity raising
  - Take out points free code
  - odd = not . even
- Defunctionalisation [Reynolds 72]
  - Remove all higher order functions
The Checker itself

- Operates on a simple first order language
- Uses constraints of the form: 
  - `<expression, path, constructors>`
- From the expression, if I follow any valid path, I get to one of the constructors
Constraints, intro by example

\[
\text{head}\ (x:xs) = x \\
\text{fromJust}\ (\text{Just}\ x) = x
\]

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

mapHead x = case x of
    [] -> []
    (:) -> head x. hd : mapHead x. tl

<mapHead@1, tl*.hd, {::}>

<mapHead@1, hd, {::}>
<mapHead@1, tl.hd, {::}>
<mapHead@1, tl.tl.hd, {::}>

...
Dealing with recursion

- Just keep expanding it
  - $x \ ^ x.a \ ^ x.aa \ ^ x.aaa \ ^ x.aaaa$

- At a certain depth, give up
  - $x.aaaa \rightarrow x.aaa^*$

- Simplify after
  - $x \ ^ x.a \ ^ x.aa \ ^ x.aaa \ ^ x.aaa^* = x.a^*$
Going back to Risers

\lambda \langle \text{risers} (y: \text{etc}), \lambda, \{ : \} \rangle \\
\langle (y: \text{etc}), \lambda, \{ : \} \rangle \\
\text{True}

Risers is safe 😊
Other programs

- **Soda (Word search)**
  - One minor tweak required
  - Was safe already

- **Adjoxo (XOX checker)**
  - One fix required
  - Was NOT safe before
  - Improves code readability
State of play

- Have a working prototype
  - Full Haskell 98
  - A number of Haskell 98 libraries
  - Works on 1-2 page programs

- Still lots to do
  - A bit slow in some cases
  - Some programs don’t work yet
Conclusion

- CATCH is a practical tool for detecting pattern match errors
- Uses a constraint language to prove safety
- [http://www.cs.york.ac.uk/~ndm/](http://www.cs.york.ac.uk/~ndm/)
- A release is coming soon (2 months)
Transformation rules

\[ \varphi\langle e \cdot s, r, c \rangle \to \varphi\langle e, s \cdot r, c \rangle \]  
(se1)

\[ \Lambda_{i=1}^{\#e'} \varphi\langle e_i, \frac{dr}{\partial S(c, i)}, c \rangle \to P \]
\[ \varphi\langle C \; e', r, c \rangle \to (\lambda \in L(r) \Rightarrow C \in c) \land P \]  
(con)

\[ \varphi\langle f \; e', r, c \rangle \to \varphi\langle D(f, e'), r, c \rangle \]  
(app)

\[ \Lambda_{i=1}^{\#e'} (\varphi\langle e_0, \lambda, C(C_i) \rangle \lor \varphi\langle e_i, r, c \rangle) \to P \]
\[ \varphi\langle \text{case } e_0 \text{ of } \{C_1 \; \vec{v} \rightarrow e_1 ; \cdots ; C_n \; \vec{v} \rightarrow e_n \}, r, c \rangle \to P \]  
(cas)
Yhc vs GHC Core

- GHC Core is:
  - More complex (letrec’s, lambda’s)
  - Lacks source position information
  - Piles and piles of type information
  - Slower to generate
  - Harder to change GHC
  - Less like the original code