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?

```
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)
```

Yes

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

```
let (a,b) = y
in (b,a)
```

```
if x then f else g
```

```
case x of
   [] -> True
   (a:b) -> a
```

```
f x | null x = []
    | otherwise = tail x
```

```
[x | Just x <- xs]
```

```
f (x:xs) = x
```

```
f x = ys
   where (y:ys)
```

```
do (x:xs) <- f y
   return xs
```

```
f [x] = x
```

```
\(a,b) -> a ++ b
```

Reduced Haskell

- Only simple case, functions, applications, constructors

```
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

λ 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

λ Constraints, intro by example

```
head (x:xs) = x
<head@1, λ, {:}>

fromJust (Just x) = x
<fromJust@1, λ, {Just}>

foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
<foldr1@2, λ, {:}>
```

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 -> x.aaa*
- Simplify after
  - x ^ x.a ^ x.aa ^ x.aaa ^ x.aaa* = x.a*

Going back to Risers

```
<risers (y:etc), λ, {:}>
<(y:etc), λ, {:}>
True
```

Risers is safe ☺

Other programs

- Soda (Word search)
  - One minor tweak required
  - Was safe already
- Adjoxo (XOX checker)
  - One fix requried
  - 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/
- A release is coming soon (2 months)

Transformation rules

$$\varphi\langle e \cdot s, r, c\rangle \to \varphi\langle e, s \cdot r, c\rangle \qquad \text{(sel)}$$

$$\frac{\bigwedge_{i=1}^{\#\overrightarrow{e}} \varphi\langle e_i, \frac{\partial r}{\partial \mathcal{S}(C,i)}, c\rangle \to P}{\varphi\langle C\ \overrightarrow{e}, r, c\rangle \to (\lambda \in L(r) \Rightarrow C \in c) \wedge P} \qquad \text{(con)}$$

$$\varphi\langle f\ \overrightarrow{e}, r, c\rangle \to \varphi\langle \mathcal{D}(f, \overrightarrow{e}), r, c\rangle \qquad \text{(app)}$$

$$\frac{\bigwedge_{i=1}^{\#\overrightarrow{e}} (\varphi\langle e_0, \lambda, \mathcal{C}(C_i)\rangle \vee \varphi\langle e_i, r, c\rangle) \to P}{\varphi\langle \texttt{case}\ e_0\ \texttt{of}\ \{C_1\ \overrightarrow{v}\texttt{->}e_1\texttt{;}\cdots\texttt{;}C_n\ \overrightarrow{v}\texttt{->}e_n\}, r, c\rangle \to P} \qquad \text{(cas)}$$

λ 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