Detecting Pattern-Match Failures in Haskell

Neil Mitchell and Colin Runciman
York University

www.cs.york.ac.uk/~ndm/catch
Does this code crash?

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]]
Does this code crash?

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]]
Does this code crash?

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]]

Property:
risers ( _: _) = ( _: _)

Potential crash
Overview

- The problem of pattern-matching
- A framework to solve patterns
- Constraint languages for the framework
- The Catch tool
- A case study: HsColour
- Conclusions
The problem of Pattern-Matching

head \( (x:xs) \) = x

head \( x\_xs \) = \textbf{case} \( x\_xs \) of
  \( x:xs \to x \)
  [] \to \text{error “head []”}

- Problem: can we detect calls to error
Haskell programs “go wrong”

- “Well-typed programs never go wrong”
- But...
  - Incorrect result/actions – requires annotations
  - Non-termination – cannot always be fixed
  - Call error – not much research done
My Goal

- Write a tool for Haskell 98
  - GHC/Haskell is merely a front-end issue
- Check statically that error is not called
  - Conservative, corresponds to a proof
- Entirely automatic
  - No annotations

= Catch
Preconditions

- Each function has a precondition
- If the precondition to a function holds, and none of its arguments crash, it will not crash

\[
\begin{align*}
\text{pre(head } x) &= x \in \{(:) \_ \_\} \\
\text{pre(assert } x \ y) &= x \in \{\text{True}\} \\
\text{pre(null } x) &= \text{True} \\
\text{pre(error } x) &= \text{False}
\end{align*}
\]
Properties

- A property states that if a function is called with arguments satisfying a constraint, the result will satisfy a constraint:

\[ x \in \{(:) \_\_\} \Rightarrow (\text{null } x) \in \{\text{True}\} \]
\[ x \in \{(:) \[]\_\} \Rightarrow (\text{head } x) \in \{\[]\} \]
\[ x \in \{\[]\} \Rightarrow (\text{head } x) \in \{\text{True}\} \]
Checking a Program (Overview)

- Start by calculating the precondition of main
  - If the precondition is True, then program is safe
- Calculate other preconditions and properties as necessary
- Preconditions and properties are defined recursively, so take the fixed point
Checking risers

risers \( r = \text{case } r \text{ of} \)

\[ \begin{align*}
\text{[]} & \to \text{[]} \\
x:xs & \to \text{case } xs \text{ of} \\
\text{[]} & \to (x:[]) : \text{[]} \\
y:etc & \to \text{case } \text{risers} (y:etc) \text{ of} \\
\text{[]} & \to \text{error “pattern match”} \\
s:ss & \to \text{case } x \leq y \text{ of} \\
\text{True} & \to (x:s) : ss \\
\text{False} & \to [x] : (s:ss)
\end{align*} \]
Checking risers

risers r = case r of
  [] → []
  x:xs → case xs of
    [] → (x:[]) : []
    y:etc → case risers (y:etc) of
      [] → error "pattern match"
      s:ss → case x ≤ y of
        True → (x:s) : ss
        False → [x] : (s:ss)
Checking risers

\[
\text{risers } r = \text{case } r \text{ of }
\]
\[
\[
\rightarrow \[
\]
\]
\]\n\[
\text{x:xs } \rightarrow \text{case } xs \text{ of }
\]
\[
\[
\rightarrow \[
\]
\]
\]
\]
\]\n\[
\text{y:etc } \rightarrow \text{case } \text{risers (y:etc) of }
\]
\[
\[
\rightarrow \text{error “pattern match”}
\]
\]
\[
\text{s:ss } \rightarrow \text{case } x <\!\!\!\!\!\!\leq y \text{ of }
\]
\[
\text{True } \rightarrow (x:s) : ss
\]
\[
\text{False } \rightarrow [x] : (s:ss)
\]
Checking risers

risers \( r = \text{case } r \text{ of} \)

\[
\begin{align*}
\text{[]} & \rightarrow \text{[]} \\
x : xs & \rightarrow \text{case } xs \text{ of} \\
\text{[]} & \rightarrow (x : []) : [] \\
y : \text{etc} & \rightarrow \text{case risers } (y : \text{etc}) \text{ of} \\
\text{[]} & \rightarrow \text{error “pattern match”} \\
s : ss & \rightarrow \text{case } x \leq y \text{ of} \\
\text{True} & \rightarrow (x : s) : ss \\
\text{False} & \rightarrow [x] : (s : ss)
\end{align*}
\]
Checking risers

risers r = case r of
    [] → []
x:xs → case xs of
    [] → (x:[]) : []
y:etc → case risers (y:etc) of
    [] → error “pattern match”
s:ss → case x ≤ y of
    True → (x:s) : ss
    False → [x] : (s:ss)

Property:
    r ∈ {(:) _ _} ⇒
    risers r ∈ {(:) _ _}
Checking risers

risers r = case r of
  [] → []
  x:xs → case xs of
    [] → (x:[]) : []
    y:etc → error "pattern match"
  risers (y:etc) ∈ {(_, _)}
  s:ss → case x ≤ y of
    True → (x:s) : ss
    False → [x] : (s:ss)

Property:
  r ∈ {(_, _)} ⇒ risers r ∈ {(_, _)}
Checking risers

risers \( r = \text{case } r \text{ of} \)

- \( [] \rightarrow [] \)
- \( x:xs \rightarrow \text{case } xs \text{ of} \)
  - \( [] \rightarrow (x:[]) : [] \)
  - \( y:\text{etc} \rightarrow \text{error "pattern match" } \)
- \( s:ss \rightarrow \text{case } x \leq y \text{ of} \)
  - \( \text{True} \rightarrow (x:s) : ss \)
  - \( \text{False} \rightarrow [x] : (s:ss) \)

Property:
\( r \in \{(:) \_\_\} \Rightarrow \text{risers } r \in \{(:) \_\_\} \)

\( r \in \{[]\} \lor \)
\( xs \in \{[]\} \lor \)
\( y:\text{etc} \in \{(:) \_\_\} \)
Calculating Preconditions

- **Variables**: $\text{pre}(x) = \text{True}$
  - Always True
- **Constructors**: $\text{pre}(a:b) = \text{pre}(a) \land \text{pre}(b)$
  - Conjunction of the children
- **Function calls**: $\text{pre}(f \ x) = x \in \text{pre}(f) \land \text{pre}(x)$
  - Conjunction of the children
  - Plus applying the preconditions of $f$
  - Note: precondition is recursive
Calculating Preconditions (case)

pre(case on of

\[
\begin{align*}
&\text{[]} \rightarrow a \\
&\text{x:xs} \rightarrow b)
\end{align*}
\]

= pre(on) \land (on \not\in \{[]\} \lor pre(a)) \\
\land (on \not\in \{(:) \_ \_\} \lor pre(b))

- An alternative is safe, or is never reached
Extending Constraints ($\uparrow$)

risers $r = \text{case } r \text{ of}$

$\;\;\;\;\;[] \rightarrow []$

$x:xs \rightarrow \text{case } xs \text{ of}$

$\;\;\;\;\;[] \rightarrow (x:[]) : []$

$y:\text{etc} \rightarrow \ldots$

$xs \in \{(:) \_ \_\} \lor \ldots$

$r^{<(:)-2>} \in \{(:) \_ \_\}$

$r \in \{(:) \_ ((:) \_ \_\})$

$r^{<(:)-2> \uparrow} \{(:) \_ \_\}$

$r^{<(:)-1> \uparrow} \{\text{True}\}$

$\{(:) \text{ True } \_\}$
Splitting Constraints (↓)

risers r = case r of
  [] → []
  x:xs → case xs of
    [] → (x:[]) : []
y:etc → ...

(x:[]):[] ∈ {(:) _ _} ∨ ...
  True

((:) 1 2) ↓ {(:) _ _}
  True

((:) 1 2) ↓ {[]} 
  False

((:) 1 2) ↓ {(:) True []}
  1 ∈ {True} ∧ 2 ∈ {[]}

Summary so far

- Rules for Preconditions
- How to manipulate constraints
  - Extend (↑) – for locally bound variables
  - Split (↓) – for constructor applications
  - Invoke properties – for function application

- Can change a constraint on expressions, to one on function arguments
Algorithm for Preconditions

set all preconditions to True
set error precondition to False
while any preconditions change
  recompute every precondition
end while

Algorithm for properties is very similar

Fixed Point!
Fixed Point

- To ensure a fixed point exists demand only a finite number of possible constraints
- At each stage, $(\wedge)$ with the previous precondition

- Ensures termination of the algorithm
  - But termination $\neq$ useable speed!
The Basic Constraints

- These are the basic ones I have introduced

- *Not* finite – but can bound the depth
  - A little arbitrary
  - Can’t represent infinite data structures

- But a nice simple introduction!
A Constraint System

- Finite number of constraints
- Extend operator ($\uparrow$)
- Split operator ($\downarrow$)
-notin creation, i.e. $x \notin \{(:) _ _\}$
- Optional simplification rules in a predicate
Regular Expression Constraints

- Based on regular expressions
- $x \in r \rightarrow c$
  - $r$ is a regular expression of paths, i.e. $<(:)-1>$
  - $c$ is a set of constructors
  - True if all $r$ paths lead to a constructor in $c$

- Split operator ($\downarrow$) is regular expression differentiation/quotient
RE-Constraint Examples

- head xs
  - $xs \in (1 \rightarrow \{\::\})$

- map head xs
  - $xs \in (\langle\cdot\rangle^{-2} \cdot \langle\cdot\rangle^{-1} \rightarrow \{\::\})$

- map head (reverse xs)
  - $xs \in (\langle\cdot\rangle^{-2} \cdot \langle\cdot\rangle^{-1} \rightarrow \{\::\}) \lor$
  - $xs \in (\langle\cdot\rangle^{-2} \rightarrow \{\::\})$
RE-Constraint Problems

- They are finite (with certain restrictions)
- But there are many of them!
- Some simplification rules
  - Quite a lot (19 so far)
  - Not complete
- In practice, too slow for moderate examples

This fact took 2 years to figure out!
Multipattern Constraints

- Idea: model the recursive and non-recursive components separately

- Given a list
  - Say something about the first element
  - Say something about all other elements
  - Cannot distinguish between element 3 and 4
MP-Constraint Examples

- **head xs**
  - $xs \in \{(\_: _) \ast \{\[], (\:) _\}\}$

  - $xs$ must be $(:)$
  - $xs.\langle(:)-1\rangle$ must be $\_$

- Use the type’s to determine recursive bits

  All recursive tails are unrestricted
More MP-Constraint Examples

- map head xs
  - \{[], (::) ({{::} _} * {[], (::) _})} * \\
  - \{[], (::) ({{::} _} * {[], (::) _})}

- An infinite list
  - \{(::) _} * {(::) _}
MP-Constraint “semantics”

MP = \{set \text{Val}\}

\text{Val} = _ | \{set \text{Pat}\} \ast \{set \text{Pat}\}

\text{Pat} = \text{Constructor} \ [(\text{non-recursive field}, \text{MP})]

Element must satisfy at least one pattern

Each recursive part must satisfy at least one pattern
MP-Constraint Split

- \(((: 1 2) \downarrow (: _) \ast (: \{True\}))\)
  - An infinite list whose elements (after the first) are all true
- \(1 \in _\)
- \(2 \in (: \{True\}) \ast (: \{True\})\)
MP-Constraint Simplification

- There are 8 rules for simplification
  - Still not complete...

- But!
  - \( x \in a \lor x \in b = x \in c \)  union of two sets
  - \( x \in a \land x \in b = x \in c \)  cross product of two sets
MP-Constraint Currying

- We can merge all MP’s on one variable
- We can curry all functions – so each has only one variable
- MP-constraint Predicate \( \equiv \) MP-constraint

\((||)\ a\ b\ \rightarrow\ (||)\ (a,\ b)\)
MP vs RE constraints

- Both have different expressive power
  - Neither is a subset/superset

- RE-constraints grow too quickly
- MP-constraints stay much smaller

- Therefore Catch uses MP-constraints
Numbers

data Int = Neg | Zero | One | Pos

- Checks
  - Is positive? Is natural? Is zero?

- Operations
  - (+1), (-1)

- Work’s very well in practice
Summary so far

- Rules for Preconditions and Properties
- Can manipulate constraints in terms of three operations
- MP and RE Constraints introduced
- Have picked MP-Constraints
Making a Tool (Catch)

- Haskell
- Core
- First-order Core
- Curried
- Analyse

Yhc

In draft paper, see website

This talk
Testing Catch

- The nofib benchmark suite, but

```haskell
main = do
  [arg] <- getArgs
  print $ primes !! (read arg)
```

- Benchmarks have no real users
- Programs without real users crash
Nofib/Imaginary Results (14 tests)

Trivially Safe
Perfect Answer
Good Failures
Bad Failures

Good failure:
Did not get perfect answer, but neither did I!
Bad Failure: Bernouilli

tail (tail x)

- Actual condition: list is at least length 2
- Inferred condition: list must be infinite

drop 2 x
Bad Failure: Paraffins

```haskell
radical_generator n = f undefined
  where f unused = big_memory_result

- array :: Ix a ⇒ (a, a) → [(a, b)] → Array a b
  - Each index must be in the given range
  - Array indexing also problematic
```
Perfect Answer: Digits of E2

e =

("2." ++) $
\text{tail} \cdot \text{concat} $
\text{map (show} \cdot \text{head)} $
\text{iterate (carryPropagate 2} \cdot \text{map (10*)} \cdot \text{tail) }$
2 : [1,1 ..]
Performance of Catch
Case Study: HsColour

- Takes Haskell source code and prints out a colourised version
- 4 years old, 6 contributors, 12 modules, 800+ lines

- Used by GHC nightly runs to generate docs
- Used online by http://hpaste.org
HsColour: Bug 1

```haskell
data Prefs = ... deriving (Read, Show)
```

- Uses read/show serialisation to a file
- readFile prefs, then read result

- Potential crash if the user has modified the file
- Real crash when Pref's structure changed!
HsColour: Bug 1 Catch

> Catch HsColour.hs
Check “Prelude.read: no parse”
Partial Prelude.read$252
Partial Language.Haskell.HsColour
  .Colourise.parseColourPrefs
...
Partial Main.main

Full log is recorded
All preconditions and properties
HsColour: Bug 2

- The latex output mode had:
  \[\text{outToken ("\":xs) = "``" ++ init xs ++ "'''"}\]

- file.hs: “
- hscolour –latex file.hs
- Crash

FIXED
HsColour: Bug 3

- The html anchor output mode had:
  outToken (``':xs) = "<a>" ++ init xs ++ "</a>"

- file.hs: (``)
- hscolour –html –anchor file.hs
- Crash

FIXED
HsColour: Problem 4

- A pattern match without a [] case
- A nice refactoring, but not a crash
- Proof was complex, distributed and fragile
  - Based on the length of comment lexemes!

- End result: HsColour cannot crash
  - Or could not at the date I checked it...
- Required 2.1 seconds, 2.7Mb
Case Study: FiniteMap library

- Over 10 years old, was a standard library
- 14 non-exhaustive patterns, 13 are safe

```haskell
delFromFM (Branch key ..) del_key
  | del_key > key = ...
  | del_key < key = ...
  | del_key ≡ key = ...
```
Case Study: XMonad

- Haskell Window Manager
- Central module (StackSet)
- Checked by Catch as a library

- No bugs, but suggested refactoring
- Made explicit some assumptions about Num
Catch’s Failings

- **Weakest Area: Yhc**
  - Conversion from Haskell to Core requires Yhc
  - Can easily move to using GHC Core (once fixed)

- **2nd Weakest Area: First-order transform**
  - Still working on this
  - Could use supercompilation
**Constraints**

- Could solve more complex problems
- Could retain numeric constraints precisely
- Ideally have a single normal form

- MP-constraints work well, but there is room for improvement
Alternatives to Catch

- Reach, SmallCheck – Matt Naylor, Colin R
  - Enumerative testing to some depth
- ESC/Haskell - Dana Xu
  - Precondition/postcondition checking
- Dependent types – Epigram, Cayenne
  - Push conditions into the types
Conclusions

- Pattern matching is an important area that has been overlooked
- Framework separate from constraints
  - Can replace constraints for different power
- Catch is a good step towards the solution
  - Practical tool
  - Has found real bugs

www.cs.york.ac.uk/~ndm/catch