Not All Patterns, But Enough

Neil Mitchell, Colin Runciman
York University

Catch

An Example

- Is the following code safe?*

```
risers :: Ord α → [α] → [[α]]
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 "Haskell" = ["Has","k","ell"]
```

* Only people who haven't seen this example in the paper!

Using Catch

```
> catch risers.hs
Incomplete pattern on line 6
Program is safe
```

- Catch is the associated implementation
- Catch has proven the program is safe
  - Without any annotations

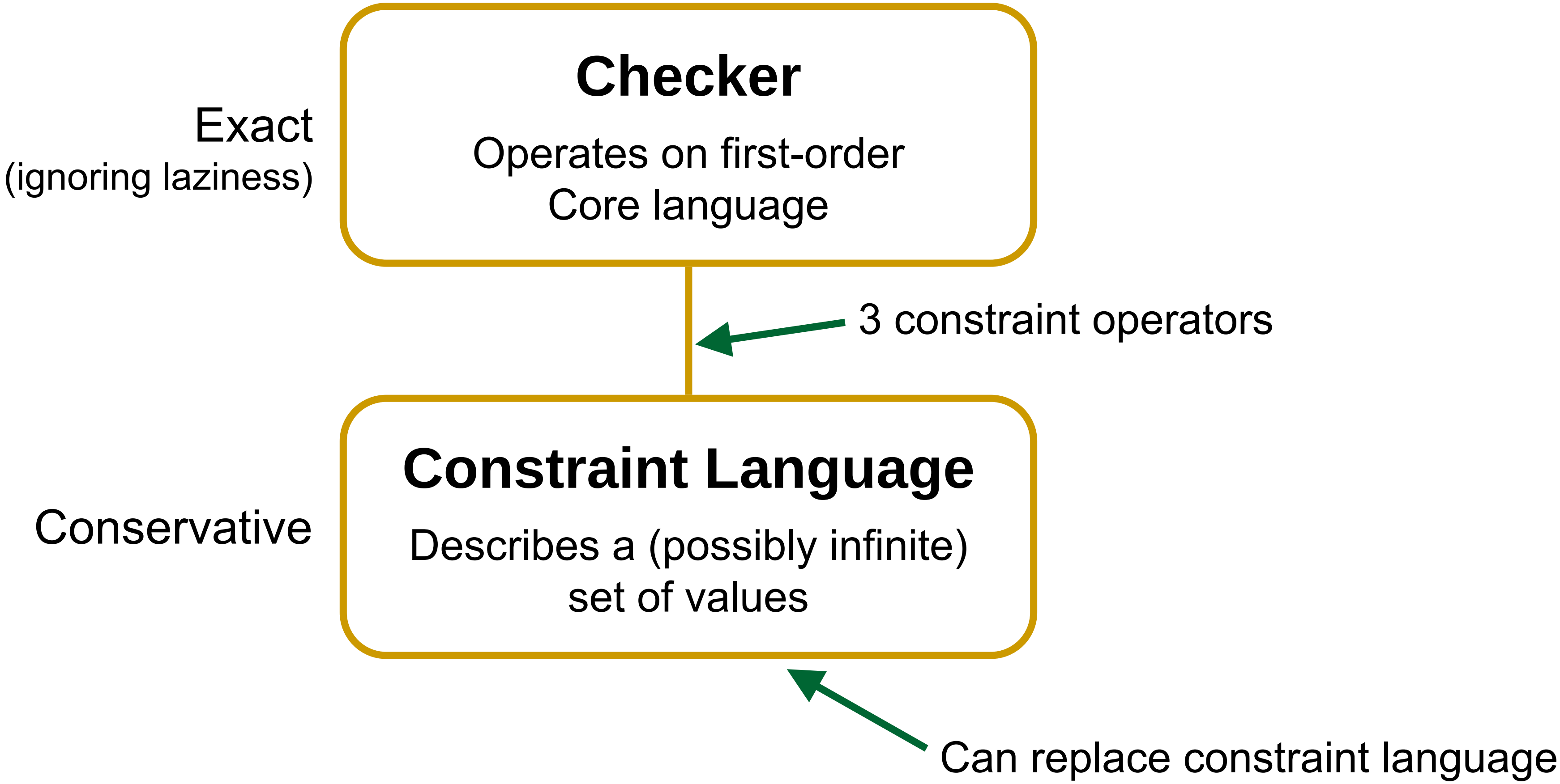
The Pattern-Matching problem

- Will a program crash when run?
  - May call error directly: error “doh!”
  - May call error indirectly: head []
  - Partial pattern match: case False of True → 1

- GHC can warn on partial patterns
- Catch conservatively checks a program will not crash at runtime
  - Even in the presence of partial patterns

How Catch works

First convert Haskell to first-order Core, using Yhc and Firstify



Checker Terms

- A *constraint* describes a set of values
  - x is a (:)-constructed value
- A *precondition* is a constraint on arguments
  - In head x, x must be (:)-constructed
- An *entailment* is a constraint on arguments to ensure a constraint on the result
  - If x is (:)-constructed, null x is False

Checker Types

- Opaque constraint type
  - data Constraint = …
- Does an expression satisfy a constraint?
  - data Sat α = Sat α Constraint
- A proposition (and, or, not)
  - data Prop α = …
- First-order Core expressions
  - data Expr = …

How the Checker works

- Compute the precondition of each function
  - Use a fixed point to deal with recursive functions
  - pre :: Expr → Prop (Sat Expr)
- Reduce constraints on expressions to constraints on function arguments
  - Important for reaching a fixed point
  - reduce :: Prop (Sat Expr) → Prop (Sat ArgPos)
- Empty precondition on main means safe

Preconditions

```
precond :: FuncName → Prop (Sat ArgPos)
precond = reduce . pre . funcBody

pre :: Expr → Prop (Sat Expr)
pre ‹v› = True
pre ‹c xs› = all pre xs
pre ‹f xs› = all pre xs  ∧  (precond f `subst` xs)
pre ‹case on of alts› = pre on  ∧  all alt alts
  where alt ‹c vs → e› = on⩽(ctors c \ [c])  ∨  pre e
```

- ⩽ is a constraint operator

Reduction

- Convert constraints on expressions to constraints on argument positions
  - reduce :: Prop (Sat Expr) → Prop (Sat ArgPos)
  - Implemented in the paper, similar to preconditions

- Requires all three constraint operators
- Also makes use of a fixed point

Constraint Operators

- Constraints must provide 3 operators
  - None mention Expr at all

- Simplest is membership
  - $(\Subset) :: \alpha \rightarrow [CtorName] \rightarrow Prop\ (Sat\ \alpha)$

- One possible implementation:
  - $x \Subset [\text{“:”}] = (x \in \{ \_ : \_ \})$

Zooming Out on Constraints

- Given a constraint on one small part of a value, what is the constraint on all of it
  - (▷) :: Selector → Constraint → Constraint

a ∈ { _ : _ }

Just a ∈ {Just (_ : _)}

Just_1 ▷ { _ : _ } = {Just (_ : _)}

Zooming In on Constraints

- Given a root constructor, what are the constraints on its fields
  - (◁) :: Ctor → Constraint → Prop (Sat ArgPos)

Just a ∈ {Just (_ : _)}

a ∈ { _ : _ }

Just ◁{Just (_ : _)} = (#1 ∈ { _ : _ })

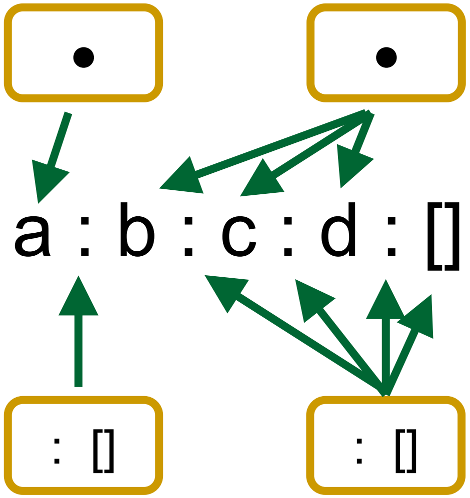
Nothing ◁{Just (_ : _)} = False

Constraint Properties

- Must be consistent
  - “[]” $\lhd$ (a $\leqslant$ [“:”]) = False
- For any type, must be a finite number of constraints (ensures termination)
- The paper presents three constraint models
  - BP-constraints are like pattern-matching
  - RE-constraints use regular expressions
  - MP-constraints are multiple patterns in one

MP-constraints concept

- Like a list of pattern-matches
- But recursive fields (i.e. tail) reuse the parents pattern



MP-constraint Examples

- precondition of head x
  - let cons = {(:) _ } * {[], (:) _ }
- precondition of map head x
  - {[], (:) cons} * {[], (:) cons}
- value is infinite list
  - {(:) _ } * {(:) _ }

Results from the Nofib suite

- Imaginary section, with MP-constraints
- Results are quite good (see paper)
  - Many programs are unsafe, because they are not for real use
- Catch takes around 1-2 seconds normally
  - One example nearly 8 seconds
  - No correlation between program size and speed

Case Study: HsColour

- Takes Haskell source code and prints out a colourised version
- 5 years old, 6 contributors, 12 modules, 800+ lines (not including libraries)

- Used to generate source links from Haddock
- Used online by http://hpaste.org
- Real program, real users!

HsColour: Bug 1

FIXED

data Prefs = … deriving (Read,Show)

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

- Potential crash if the user modifies the file
- Real crash when Prefs 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
```

- Catch pinpoints the error, and a stack trace
  - Can optionally show the constraints

HsColour: Bug 2

FIXED

- The latex output mode had:

outToken ('\"':xs) = "``" ++ init xs ++ "''"

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

HsColour: Bug 3

FIXED

- The html anchor output mode had:

```
outToken (‘`’:xs) = “<a>” ++ init xs ++ “</a>”
```

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

HsColour: Issue 4

CHANGED

- 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 at least couldn't when I last 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

```
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 unexpected bugs found
  - But some nice refactorings
- Made explicit some assumptions about Num

Alternatives to Catch

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

Conclusion

- Pattern matching is an important problem that has been overlooked
  - darcs bugs: 13 fromJust and 19 pattern-matches
- One analysis with several constraint models
  - Can replace constraints for different power
- Catch is a good step towards a solution
  - Has found real bugs

XMonad developers quote

“XMonad made heavy use of Catch in the development of its core data structures and logic. Catch caught several suspect error cases, and helped us improve robustness of the window manager core by weeding out partial functions. It helps encourage a healthy skepticism to partiality, and the quality of code was improved as a result. We’d love to see a partiality checker integrated into GHC.”