Not All Patterns, But Enough

– an automatic verifier for partial but sufficient pattern matching

Neil Mitchell *
University of York, UK
http://www.cs.york.ac.uk/~ndm/

Colin Runciman
University of York, UK
http://www.cs.york.ac.uk/~colin/

Abstract

We describe an automated analysis of Haskell 98 programs to check statically that, despite the possible use of partial (or non-exhaustive) pattern matching, no pattern-match failure can occur. Our method is an iterative backward analysis using a novel form of pattern-constraint to represent sets of data values. The analysis is defined for a core first-order language to which Haskell 98 programs are reduced. Our analysis tool has been successfully applied to a range of programs, and our techniques seem to scale well. Throughout the paper, methods are represented much as we have implemented them in practice, again in Haskell.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms languages, verification

Keywords Haskell, automatic verification, functional programming, pattern-match errors, preconditions

1. Introduction

Many functional languages support case-by-case definition of functions over algebraic data types, matching arguments against alternative constructor patterns. In the most widely used languages, such as Haskell and ML, alternative patterns need not exhaust all possible values of the relevant datatype; it is often more convenient for pattern matching to be partial. Common simple examples include functions that select components from specific constructions — in Haskell tail applies to (:)-constructed lists and fromJust to Just-constructed values of a Maybe-type.

Partial matching does have a disadvantage. Programs may fail at run-time because a case arises that matches none of the available alternatives. Such pattern-match failures are clearly undesirable, and the motivation for this paper is to avoid them without denying the convenience of partial matching. Our goal is an automated analysis of Haskell 98 programs to check statically that, despite the possible use of partial pattern matching, no pattern-match failure can occur.

The problem of pattern-match failures is a serious one. The *darcs* project (Roundy 2005) is one of the most successful large scale programs written in Haskell. Taking a look at the darcs bug tracker, 13 problems are errors related to the selector function fromJust and 19 are direct pattern-match failures.

Consider the following example taken from Mitchell and Runciman (2007):

$$\begin{array}{l}\mathsf{risers} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow [[\alpha]] \\ \mathsf{risers}\ [\,] = [\,] \\ \mathsf{risers}\ [x] = [[x]] \\ \mathsf{risers}\ (x : y : etc) = \textbf{if}\ x \leqslant y\ \textbf{then}\ (x : s) : ss\ \textbf{else}\ [x] : (s : ss) \\ \quad \textbf{where}\ (s : ss) = \mathsf{risers}\ (y : etc)\end{array}$$

A sample application of this function is:

$$\begin{array}{l}>\mathsf{risers}\ [1,2,3,1,2] \\ [[1,2,3],[1,2]]\end{array}$$

In the last line of the definition, $(s : ss)$ is matched against the result of risers $(y : etc)$. If the result is in fact an empty list, a pattern-match error will occur. It takes a few moments to check manually that no pattern-match failure is possible – and a few more to be sure one has not made a mistake! Turning the risers function over to our analysis tool (which we call CATCH), the output is:

```
Checking "Incomplete pattern on line 5"
Program is Safe
```

In other examples, where CATCH cannot verify pattern-match safety, it can provide information such as sufficient conditions on arguments for safe application of a function.

We have implemented all the techniques reported here. We encourage readers to download the Catch tool and try it out. It can be obtained from the website at http://www.cs.york.ac.uk/~ndm/catch/. A copy of the tool has also been released, and is available on Hackage[1].

1.1 Contributions

The contributions of this paper include:

- Two constraint languages to reason about pattern-match failures, with methods for generating and solving constraints.
- Details of the CATCH implementation which handles the complete Haskell 98 language (Peyton Jones 2003), by transforming Haskell 98 programs to a first-order language.
- Results showing success on a number of small examples drawn from the Nofib suite (Partain et al. 2007), and for two larger examples, investigating the scalability of the checker.

In Mitchell and Runciman (2007) a pattern-match checker was described with similar aims, which we will refer to as CATCH05[2].

* The first author is supported by an EPSRC PhD studentship

[1] http://hackage.haskell.org/

[2] Although the paper was completed in 2005, publication was delayed

```
risers x = case x of
  [] → []
  (y : ys) → case ys of
    [] → (y : []) : []
    (z : zs) → risers2 (risers3 z zs) (y ⩽ z) y

risers2 x y z = case y of
  True → (z : snd x) : (fst x)
  False → (z : []) : (snd x : fst x)

risers3 x y = risers4 (risers (x : y))

risers4 x = case x of
  (y : ys) → (ys, y)
  [] → error "Pattern Match Failure, 11:12."
```

Figure 1. risers in the Core language.

For CATCH05 risers was towards the boundary of what was possible, for CATCH07 it is trivial. Scalability problems have been addressed and CATCH07 handles real Haskell programs, with no restriction on the recursion patterns, higher-order functions or type classes. The underlying constraint mechanism of CATCH07 is radically different.

1.2 Road map

§2 gives an overview of the checking process for the risers function. §3 introduces a small core functional language and a mechanism for reasoning about this language, §4 describes two constraint languages. §5 discusses how to transform Haskell to a first-order core language. §6 evaluates CATCH07 on programs from the Nofib suite, on a widely-used library and on a larger application program. §7 offers comparisons with related work before §8 presents concluding remarks.

2. Overview of the Risers Example

This section sketches the process of checking that the risers function in the Introduction does not crash with a pattern-match error.

2.1 Conversion to a Core Language

Rather than analyse full Haskell, CATCH analyses a first-order Core language, without lambda expressions, partial application or let bindings. A convertor is provided from the full Haskell 98 language to this restricted language – see §5. The result of converting the risers program to Core Haskell, and renaming the identifiers for ease of human reading, is shown in Figure 1.

The type of risers is polymorphic over types in Ord. CATCH can check risers assuming that Ord methods do not raise pattern-match errors, and may return any value. Or a type instance such as Int can be specified with a type signature.

2.2 Analysis of risers – a brief sketch

In the Core language every pattern match covers all possible constructors of the appropriate type. The alternatives for constructor cases not originally given are calls to error. The analysis starts by finding calls to error, then tries to prove that these calls will not be reached. The one error call in risers4 is avoided under the precondition (see §3.4):

risers4, $x \leqslant (:)$

That is, all callers of risers4 must supply an argument x which is a $(:)$-constructed value. For the proof that this precondition holds, two properties are required (see §3.5):

```
type CtorName  = String
type FuncName  = String
type VarName   = String
type Selector  = (CtorName, Int)

data Func = Func FuncName [VarName] Expr

data Expr = Var   VarName
          | Make CtorName [Expr]
          | Call  FuncName [Expr]
          | Case  Expr      [Alt]

data Alt  = Alt CtorName [VarName] Expr
```

Figure 2. Core Data Type.

```
data Value = Bottom | Value CtorName [Value]

eval :: Expr → Value
eval x = case hnf x of
           Nothing     → Bottom
           Just (c, cs) → Value c (map eval cs)

  -- Nothing corresponds to bottom
hnf :: Expr → Maybe (CtorName, [Expr])
hnf (Make x  xs  ) = Just (x, xs)
hnf (Call  x  xs  ) | x ≡ "error" = Nothing
                    | otherwise   = hnf (instantiate x xs)
hnf (Case  on  alts ) = listToMaybe [res
  | Just (c, cs) ← [hnf on], Alt n vs e ← alts, c ≡ n
  , Just res ← [hnf (subst (zip vs cs) e)]]

subst :: [(VarName, Expr)] → Expr → Expr
subst r (Var   x   ) = fromMaybe (Var x) (lookup x r)
subst r (Make x xs) = Make x (map (subst r) xs)
subst r (Call  x xs) = Call  x (map (subst r) xs)
subst r (Case  x xs) = Case (subst r x)
  [Alt n vs (subst r2 e) | Alt n vs e ← xs
  , let r2 = filter ((∉ vs) ∘ fst) r]
```

Figure 3. Semantics for Core expressions.

$x \leqslant (:) \Rightarrow$ (risers x $\quad$) $\leqslant (:)$
True $\Rightarrow$ (risers2 x y z) $\leqslant (:)$

The first property says that if the argument to risers is a $(:)$-constructed value, the result will be. The second states that the result from risers2 is always $(:)$-constructed.

3. Pattern Match Analysis

This section explains the methods used to calculate preconditions for functions. A basic constraint language is introduced, along with operations upon it.

3.1 Reduced expression language

Syntax for the core language is given in Figure 2. Our Core language is a little more restrictive than the core languages typically used in compilers (Tolmach 2001). It is first order, has only simple case statements, and only algebraic data types. All case statements have alternatives for all constructors, with error calls being introduced where a pattern-match error would otherwise occur.

The evaluation strategy is lazy. A semantics is outlined in Figure 3, as an evaluator from expression to values, written in Haskell.

```
ctors       :: CtorName  → [CtorName]
arity       :: CtorName  → Int
var         :: VarName   → Maybe (Expr, Selector)
instantiate :: FuncName  → [Expr] → Expr
isRec       :: Selector  → Bool
```

Figure 4. Operations on Core.

The hnf function evaluates an expression to head normal form. The subst function substitutes free variables that are the result of a Case expression. Laziness is a useful property as it allows β-reduction to be performed by the checker.

3.1.1 Operations on Core

Figure 4 gives the signatures for helper functions over the core data types. Every constructor has an arity, which can be obtained with the arity function. To determine alternative constructors the ctors function can be used; for example ctors `"True"` = [`"False"`, `"True"`] and ctors `"[]"` = [`"[]"`, `":"`]. The var function returns Nothing for a variable bound as an argument of a top-level function, and Just $(e, (c, i))$ for a variable bound as the ith component in the c-constructed alternative of case-expression e. The function instantiate corresponds to β reduction. The isRec (c, n) function returns true if the constructor c has a recursive nth component; for example, let hd = $(\texttt{":"}, 0)$ and tl = $(\texttt{":"}, 1)$ then isRec hd = False but isRec tl = True.

3.1.2 Algebraic Abstractions of Primitive Types

The Core language only has algebraic data types. CATCH allows for primitive types such as characters and numbers by abstracting them into algebraic types. For example, the integer abstraction used in CATCH is simply:

```
data Int = Neg | Zero | One | Pos
```

In our experience, numbers are most often constrained to be a natural, or to be non-zero. Addition or subtraction of one is the most common operation. Though very simple, the abstraction models the common properties and operations quite well.

Possible abstractions of characters include:

```
data Char = Char
data Char = Alpha | Digit | White | Other
data Char = 'A'..'Z' | 'a'..'z' | '0'..'9' | Other
data Char = '\0' | '\1' | '\2'..
```

CATCH employs the first abstraction by default, but provides a flag to allow alternatives. In practice, few additional programs are proven safe with a more refined character abstraction.

The final issue of abstraction relates to primitive functions in the IO monad, such as getArgs (which returns the command-line arguments), or readFile (which reads from the file-system). In most cases an IO function is modelled as returning *any* value of the correct type, using a function primitive to the checker.

3.2 Constraint Essentials and Notation

An expression in our first-order core language evaluates to a data structure. A constraint describes a set of data structures to which a value may belong. If a component within a data structure evaluates to $\bot$, there are no constraints upon it.

If x is an expression and c is a constraint we write $x \lessdot c$ to assert that the value of expression x must be a member of the set satisfying c. It is convenient to have a notation for constraints upon named arguments of named functions: we write f, $x \lessdot c$ to assert that argument x of f must be in the set satisfying c.

```
data Prop α

(∧), (∨), (⟹) :: Prop α → Prop α → Prop α
andP, orP     :: [Prop α] → Prop α
mapP          :: (α → β) → Prop α → Prop β
true, false   :: Prop α
isTrue, isFalse :: Prop α → Bool
bool          :: Bool → Prop α
lit           :: α → Prop α

(⇒) :: Bool → Bool → Bool
```

Figure 5. Property data type

```
data Assert a = a⩽Constraint

notin :: CtorName → Constraint
(▷) :: Selector → Constraint → Constraint
(◁) :: CtorName → Constraint → Prop (Assert Int)
```

Figure 6. Constraint operations.

```
type Constraint = [Pattern]
data Pattern    = Con CtorName [Pattern] | Any
```

Figure 7. Basic pattern constraints.

Atomic constraints can be combined into propositions, using the proposition data type in Figure 5. We also introduce $(\Rightarrow)$, standard boolean implication – Haskell already provides $(\leqslant)$, but we prefer the clarity of a different symbol.

Several underlying constraint models are possible. To keep the introduction of the algorithms simple we first use *basic pattern constraints* (§3.3). We then describe *regular expression constraints* in §4.1 – a variant of the constraints used in CATCH05. Finally we present *multi-pattern constraints* in §4.2 – used in CATCH07 to enable scaling to much larger problems.

Some operations must be provided in every constraint implementation. Signatures are given in Figure 6. Assert is the type of the $(\lessdot)$ condition. The lifting and splitting operators $(\triangleright)$ and $(\triangleleft)$ are discussed in §3.5. The notin function generates a constraint which does *not* match the constructor given.

3.3 Basic Pattern (BP) Constraints

For simplicity, our analysis framework will be introduced using basic pattern constraints (BP-constraints). A BP-constraint directly corresponds to Haskell pattern matching. A constraint is a finite set of patterns represented as in Figure 7. A given data structure d satisfies a constraint ps if d matches at least one pattern in ps. True and false constraints can be represented as [Any] and [] respectively. The requirement for a value to be (:)-constructed would be expressed as (Con `":"` [Any, Any]).

The BP-constraint language is limited in expressivity. For example it is impossible to state that all the elements of a boolean list are True. However, even with this limited constraint language, the Risers example can be proven safe.

As an example of an operator definition for the BP-constraint language, notin can be defined:

```
notin c = map f (delete c (ctors c))
  where f x = Con x (replicate (arity x) Any)
```

```
type PreFun = FuncName → [Expr] → Prop (Assert Expr)

pre :: PreFun → Expr → Prop (Assert Expr)
pre pf (Var  x      ) = true
pre pf (Make _  xs  ) = andP (map (pre pf) xs)
pre pf (Call  fn xs ) = pf fn xs ∧ andP (map (pre pf) xs)
pre pf (Case  on alts) = pre pf on ∧ andP (map f alts)
  where f (Alt c vs e) = lit (on⩽notin c) ∨ pre pf e
```

Figure 8. Precondition of an expression, pre.

```
for fn ∈ funcs do conds (fn) := bool (fn ≢ "error")
loop
  for fn ∈ funcs do
    conds′ (fn) := conds (fn) ∧ pre preFun (instantiate fn vs)
  if conds′ ≡ conds then break
  conds := conds′
end loop
  where
    preFun fn xs = conds (fn) [vs1 / xs1 .. vsn / xsn]
    vs = ...  -- free variables
```

Figure 9. Precondition calculation.

3.4 Preconditions for Pattern Safety

Our intention is that for every function, constraints on the arguments form a precondition for pattern safety. The precondition for error is False. A program is safe if the precondition on main is True. The process of analysis can be seen as deriving these preconditions. Given a function pf that returns the precondition on a function's arguments, a function to determine the precondition for safe evaluation of an *expression* can be specified as pre in Figure 8.

3.4.1 Stable Preconditions

The precondition on each function must be at least as restrictive as the precondition on the body of that function:

```
propRestrictive fn xs = isTrue $
  preFun fn xs ⟹ pre preFun (instantiate fn xs)
```

One conservative (and useless) implementation of preFunc is to always return False. A perfect precondition would be such that preFunc $fn\ xs \equiv$ pre (instantiate $fn\ xs$).

The iterative algorithm for calculating preconditions is given in Figure 9. Initially all preconditions are assumed to be True, apart from the error precondition, which is False. New preconditions are calculated for every function, until no condition changes. In each iteration the preconditions become more restrictive. So *if all chains of increasingly restrictive constraints are finite*, termination is guaranteed. (We return to finiteness in §3.7.) A more efficient algorithm tracks dependencies between preconditions, and performs the minimum amount of recalculation. Finding strongly connected components in the static call graph of a program would allow parts of the program to be checked separately.

3.4.2 Preconditions and Laziness

The pre function defined in Figures 8 does not exploit laziness. The Call equation demands that preconditions hold on *all* arguments – only correct if a function is strict in all arguments. For example, the precondition on False && error "here" is False, when it should be True. In general, preconditions may be more restrictive than necessary. However, investigation of a range of examples suggests that inlining (&&) and (||) captures many of the common cases where laziness would be required.

```
back :: Assert Expr → Prop (Assert Expr)
back (Var  x      ⩽k) = lit $ on⩽(c ▷ k)
  where Just (on, c) = var x
back (Make c  xs  ⩽k) = replaceVars xs (c ◁ k)
back (Case  on alts⩽k) = andP [f c e | Alt c vs e ← alts]
  where f c e = lit (on⩽notin c) ∨ lit (e⩽k)
back (Call  fn xs  ⩽k) = replaceVars xs (property fn k)

backs :: Prop (Assert Expr) → Prop (Assert VarName)
property :: FuncName → Constraint → Prop (Assert Int)

replaceVars :: [Expr] → Prop (Assert Int) → Prop (Assert Expr)
replaceVars xs p = mapP (λ(i⩽k) → (xs !! i)⩽k) p
```

Figure 10. Specification of backward analysis, back.

3.5 Manipulating constraints

Constraints on expressions other than argument variables can be rewritten using the back function, detailed in Figure 10. The backs function repeatedly applies back until all constraints are on arguments. The replaceVars function takes a propositional constraint over a set of index variables, $v_1 \ldots v_n$, and replaces each variable with a corresponding expression from an indexed list.

The Var rule applies to variables bound by patterns in case alternatives. It lifts conditions on a bound variable to the subject of the case expression in which they occur. The $\triangleright$ operator lifts a constraint on one part of a data structure to a constraint on the entire data structure. For BP-constraints $\triangleright$ can be defined:

```
(c, i) ▷ k = map extend k
  where
  extend x = Con c (anys i ++ [x] ++ anys (arity c−i−1))
  anys j = replicate j Any
```

The Make rule deals with an application of a constructor. The $\triangleleft$ operator splits a constraint on an entire structure into a combination of constraints on each part.

```
c ◁ k = orP (map f k)
  where
  f Any = true
  f (Con c2 xs) = bool (c2 ≡ c) ∧
    andP (map lit (zipWith (<) [0 ..] (map (:[]) xs)))
```

The Case rule generates a conjunct for each alternative. An alternative is safe if *either* it is never taken, *or* it meets the constraint when taken. The notin function is used to assert that a particular alternative never matches.

The Call rule relies on the key property:

```
propRestrict fn xs k = isTrue $
  backs (lit $ Call        fn xs⩽k) ⟹
  backs (lit $ instantiate fn xs⩽k)
```

That is, the result is at least as restrictive as if the function was inlined at this point. Two strategies can be used: (1) Use β substitution (instantiate) replacing the call with the body of fn. (2) Create fresh variables for each argument, use β substitution, and then instantiate the results. CATCH05 used method 1. CATCH07 uses method 2, as it is easier to find a fixed point, and allows a cache of results to be built – reducing duplicate computation.

The function property in Figure 11 takes a postcondition, and transforms it to constraints on the argument positions – a precondition to ensure the postcondition. As in the precondition calculation

```
prop := [(fn, k) | fn ← funcs, k ← constraints]
for (fn, k) ∈ ps do prop (fn, k) := true
loop
  for (fn, k) ∈ p do
    prop′ (fn, k) := prop (fn, k) ∧ backs (instantiate fn vs≪k)
  if prop′ ≡ prop then break
  prop := prop′
end loop
  where
    property fn k = prop (fn, k)
    vs = ...   -- free variables
```

Figure 11. Fixed point calculation for property.

in §3.4, each result becomes increasingly restrictive. If refinement chains of constraint/function pairs are finite, termination is guaranteed. Here again, a speed up can be obtained by tracking the dependencies between constraints.

3.6 Semantics of Constraints

A key function in the semantics of constraints tests whether a value satisfies a constraint. The $\triangleright$ operator already provides all the necessary information:

```
satisfies :: Value → Constraint → Bool
satisfies Bottom       k = True
satisfies (Value c xs) k = isTrue $ mapP f (c ◁ k)
  where f (i≪k2) = satisfies (xs !! i) k2
```

The first equation returns True given a value of type Bottom, as for a Bottom to arise some other constraint must have been violated. Now we can express properties that the other constraint operations must have:

```
propExtend v@(Value c xs) k i =
  satisfies v ((c, i) ▷ k) ⇒ satisfies (xs !! i) k
propExtend _ _ _ = True

propNotin v@(Value c xs) = not (satisfies v (notin c))
propNotin _ = True
```

Note that both properties allow for constraints to be more restrictive than necessary.

3.7 Finite Refinement of Constraints

With unbounded recursion in patterns, the BP-constraint language does *not* have only finite chains of refinement. As we saw in §3.4.1, we need this property for termination of the iterative analysis. In the next section we introduce two alternative constraint systems. Both share a key property: *for any type, there are finitely many constraints*.

4. Richer Constraint Systems

There are various ways of defining a richer constraint system, also providing the necessary finiteness properties. Here we outline two – one adapted from CATCH05, one entirely new – both implemented in CATCH07. Neither is strictly more powerful than the other; each is capable of expressing constraints that the other cannot express.

4.1 Regular Expression (RE) Constraints

CATCH05 used regular expressions in constraints. A data type for regular expression based constraints (RE-constraints), along with the essential operations upon it is given in Figure 12. In a constraint of the form $(r \rightsquigarrow cs)$, r is a regular expression and cs is a set of constructors. Such a constraint is satisfied by a data structure

```
data Constraint = RegExp ⇝ [CtorName]
type RegExp     = [RegItem]
data RegItem    = Atom Selector | Star [Selector]

notin :: CtorName → Constraint
notin c = [] ⇝ delete c (ctors c)

(▷) :: Selector → Constraint → Constraint
p ▷ (r ⇝ cs) = integrate p r ⇝ cs

(◁) :: CtorName → Constraint → Prop (Assert Int)
c ◁ (r ⇝ cs) = bool (ewp r ⇒ (c ∈ cs)) ∧
  andP (map f [0 .. arity c−1])
  where
  f i = case differentiate (c, i) r of
          Nothing → true
          Just r2 → lit $ i≪(r2 ⇝ cs)

ewp :: RegExp → Bool
ewp x = all isStar x
  where isStar (Star _) = True
        isStar (Atom _) = False

integrate :: Selector → RegExp → RegExp
integrate p r | not (isRec p) = Atom p : r
integrate p (Star ps : r)     = Star (nub (p : ps)) : r
integrate p r                 = Star [p] : r

differentiate :: Selector → RegExp → Maybe RegExp
differentiate p [] = Nothing
differentiate p (Atom r : rs) | p ≡ r      = Just rs
                              | otherwise  = Nothing
differentiate p (Star r : rs) | p ∈ r      = Just (Star r : rs)
                              | otherwise  = differentiate p rs
```

Figure 12. RE-constraints.

d if every well-defined application to d of a sequence of selectors described by r reaches a constructor in the set cs. If no sequence of selectors has a well-defined result then the constraint is vacuously true.

Concerning the helper functions needed to define $\triangleright$ and $\triangleleft$ in Figure 12, the differentiate function is from Conway (1971); integrate is its inverse; ewp is the empty word property.

CATCH05 regular expressions were unrestricted and quickly grew to an unmanageable size, thwarting analysis of larger programs. In general, a regular expression takes one of six forms:

$r_1 + r_2$	union of regular expressions r_1 and r_2
$r_1 \cdot r_2$	concatenation of regular expressions r_1 then r_2
${r_1}^*$	any number (possibly zero) occurrences of r_1
sel	a selector, i.e. hd for the head of a list
0	the language is the empty set
1	the language is the set containing the empty string

CATCH07 implements REs using the data type RegExp from Figure 12, with RegExp being a list of concatenated RegItem. In addition to the restrictions imposed by the data type, we require: (1) within Atom the Selector is not recursive; (2) within Star there is a non-empty list of Selectors, each of which is recursive; (3) no two Star constructors are adjacent in a concatenation. These restrictions are motivated by three observations:

- Because of static typing constructor-sets must all be of the same type. (In CATCH05 expressions such as hd^* could arise.)
- There are finitely many regular expressions for any type. Combined with the finite number of constructors, this property is enough to guarantee termination when computing a fixed-point iteration on constraints.
- The restricted REs with 0 are closed under integration and differentiation. (The 0 alternative is catered for by the Maybe return type in the differentiation. As $0 \rightsquigarrow c$ always evaluates to True, $\lhd$ replaces Nothing by True.)

Example 1 (head xs) is safe if xs evaluates to a non-empty list. The RE-constraint generated by CATCH is: $xs \leqslant (1 \rightsquigarrow \{:\})$ □

Example 2 (map head xs) is safe if xs evaluates to a list of non-empty lists. The RE-constraint is: $xs \leqslant (\mathsf{tl}^* \cdot \mathsf{hd} \rightsquigarrow \{:\})$ If xs is empty, it still satisfies the constraint. If xs is infinite then the condition applies to all elements, constraining an infinite number. □

Example 3 (map head (reverse xs)) is safe if every item in xs is (:)-constructed, or if xs is infinite – so reverse does not terminate. The RE-constraint is: $xs \leqslant (\mathsf{tl}^* \cdot \mathsf{hd} \rightsquigarrow \{:\}) \vee xs \leqslant (\mathsf{tl}^* \rightsquigarrow \{:\})$ □

4.1.1 Finite Number of RE-Constraints

We require that for any type, there are finitely many constraints (see §3.7). We can model types as:

```
data Type = Type [Ctor]
type Ctor = [Maybe Type]
```

Each Type has a number of constructors. For each constructor Ctor, every component has either a recursive type (represented as Nothing) or a non-recursive type t (represented as Just t). As each non-recursive type is structurally smaller than the original, a function that recurses on the type will terminate. We define a function count which takes a type and returns the number of possible RE-constraints.

```
count :: Type → Integer
count (Type t) = 2^rec*(2^ctor+sum (map count nonrec))
  where
  rec = length (filter isNothing (concat t))
  nonrec = [x | Just x ← concat t]
  ctor = length t
```

The $2 \hat{}$ rec term corresponds to the number of possible constraints under Star. The $2 \hat{}$ ctor term accounts for the case where the selector path is empty.

4.1.2 RE-Constraint Propositions

CATCH computes over propositional formulae with constraints as atomic propositions. Among other operators on propositions, they are compared for equality to obtain a fixed point. All the fixed-point algorithms given in this paper stop once equal constraints are found. We use Binary Decision Diagrams (BDD) (Lee 1959) to make these equality tests fast. Since the complexity of performing an operation is often proportional to the number of atomic constraints in a proposition, we apply simplification rules to reduce this number. Three of the nineteen rules are:

Exhaustion: In the constraint $x \leqslant (r \rightsquigarrow [\texttt{":"}, \texttt{"[]"}])$ the condition lists all the possible constructors. Because of static typing x must be one of these constructors. Any such constraint simplifies to True.

And merging: The conjunction $e \leqslant (r \rightsquigarrow c_1) \wedge e \leqslant (r \rightsquigarrow c_2)$ can be replaced by $e \leqslant (r \rightsquigarrow (c_1 \cap c_2))$.

Or merging: The disjunction $e \leqslant (r \rightsquigarrow c_1) \vee e \leqslant (r \rightsquigarrow c_2)$ can be replaced by $e \leqslant (r \rightsquigarrow c_2)$ if $c_1 \subseteq c_2$.

```
type Constraint = [Val]
data Val        = [Pattern] ⋆ [Pattern] | Any
data Pattern    = Pattern CtorName [Val]

  -- useful auxiliaries, non recursive selectors
nonRecs :: CtorName → [Int]
nonRecs c = [i | i ← [0 .. arity c−1], not (isRec (c, i))]

  -- a complete Pattern on c
complete :: CtorName → Pattern
complete c = Pattern c (map (const Any) (nonRecs c))

notin :: CtorName → Constraint
notin c = [map complete (delete c cs) ⋆ map complete cs]
  where cs = ctors c

(▷) :: Selector → Constraint → Constraint
(c, i) ▷ k = notin c ++ map f k
  where
  f Any = Any
  f (ms1 ⋆ ms2) | isRec (c, i) = [complete c] ⋆ merge ms1 ms2
  f v = [Pattern c [if i ≡ j then v else Any | j ← nonRecs c]]
        ⋆ map complete (ctors c)

(◁) :: CtorName → Constraint → Prop (Assert Int)
c ◁ vs = orP (map f vs)
  where
  (rec, non) = partition (isRec ∘ (, ) c) [0 .. arity c−1]

  f Any = true
  f (ms1 ⋆ ms2) = orP [andP $ map lit $ g vs1
                       | Pattern c1 vs1 ← ms1, c1 ≡ c]
    where g vs = zipWith (◁) non (map (:[]) vs) ++
                 map (⩽[ms2 ⋆ ms2]) rec

(⊓) :: Val → Val → Val
(a1 ⋆ b1) ⊓ (a2 ⋆ b2) = merge a1 a2 ⋆ merge b1 b2
x         ⊓ y         = if x ≡ Any then y else x

merge :: [Pattern] → [Pattern] → [Pattern]
merge ms1 ms2 = [Pattern c1 (zipWith (⊓) vs1 vs2) |
      Pattern c1 vs1 ← ms1, Pattern c2 vs2 ← ms2, c1 ≡ c2]
```

Figure 13. MP-constraints.

4.2 Multipattern (MP) Constraints & Simplification

Although RE-constraints are capable of solving many examples, they suffer from a problem of scale. As programs become more complex the size of the propositions grows quickly, slowing CATCH unacceptably. Multipattern constraints (MP-constraints, defined in Figure 13) are an alternative which scales better.

MP-constraints are similar to BP-constraints, but can constrain an infinite number of items. A value v satisfies a constraint $p_1 \star p_2$ if v itself satisfies p_1 and all its recursive components at any depth satisfy p_2. Each of p_1 and p_2 is given as a set of matches as in §3.3, but each Pattern only specifies the values for the non-recursive selectors, all recursive selectors are handled by p_2. A constraint is a disjunctive list of $\star$ patterns.

Example 1 (revisited) safe evaluation of (head xs) requires xs to be non-empty. The MP-constraint generated by CATCH on xs is: $\{(:)\ \mathsf{Any}\} \star \{[\,], (:)\ \mathsf{Any}\}$ This constraint is longer than the

corresponding RE-constraint as it makes explicit that both the hd field and the recursive components are unrestricted. □

Example 2 (revisited) safe evaluation of (map head xs) requires xs to be a list of non-empty lists. The MP-constraint on xs is:
$\{[], (:)\ (\{(:)\ \mathsf{Any}\} \star \{[], (:)\ \mathsf{Any}\})\} \star$
$\{[], (:)\ (\{(:)\ \mathsf{Any}\} \star \{[], (:)\ \mathsf{Any}\})\}$ □

Example 3 (revisited) (map head (reverse x)) requires xs to be a list of non-empty lists *or* infinite. The MP-constraint for an infinite list is: $\{(:)\ \mathsf{Any}\} \star \{(:)\ \mathsf{Any}\}$ □

MP-constraints also have simplification rules. Two of the eight rules are:

[Val] **simplification:** Given a list of Val, if the value Any is in this list, the list is equal to [Any]. If a value occurs more than once in the list, one copy can be removed.

Val **simplification:** If both p_1 and p_2 cover all constructors and all their components have Any as their constraint, the constraint $p_1 \star p_2$ can be replaced with Any.

4.2.1 Finitely Many MP-Constraints per Type

As in §4.1.1, we show there are finitely many constraints per type by defining a count function:

```
count :: Type → Integer
count (Type t) = 2^val t

val t = 1+2*2^(pattern t)

pattern t = sum (map f t)
  where f c = product [count t2 | Just t2 ← c]
```

The val function counts the number of possible Val constructions. The pattern function performs a similar role for Pattern constructions.

4.2.2 MP-Constraint Propositions and Uncurrying

A big advantage of MP-constraints is that if two constraints on the same expression are combined at the proposition level, they can be reduced into one constraint:

$(e \lessdot v_1) \vee (e \lessdot v_2) = e \lessdot (v_1 +\!\!+ v_2)$
$(e \lessdot v_1) \wedge (e \lessdot v_2) = e \lessdot [a \sqcap b \mid a \leftarrow v_1, b \leftarrow v_2]$

This ability to combine constraints on equal expressions can be exploited further by translating the program to be analysed. After applying backs, all constraints will be in terms of the arguments to a function. So if all functions took exactly one argument then *all* the constraints associated with a function body could be collapsed into one. We therefore *uncurry* all functions.

Example 4

```
(||) x y = case x of
             True  → True
             False → y
```

can be translated into:

```
(||) a = case a of
           (x, y) → case x of
                      True  → True
                      False → y
```
□

Combining MP-constraint reduction rules with the uncurrying transformation makes Assert VarName equivalent in power to Prop (Assert VarName). This simplification reduces the number of different propositional constraints, making fixed-point computations faster. In the RE-constraint system uncurrying would do no harm, but it would be of no use. None of the RE simplification rules is able to reduce distinct components in a tuple.

4.3 Comparison of RE and MP Constraints

As we discussed in §3.7, it is not possible to use BP-constraint, as they do not have finite chains of refinement. Both RE-constraints and MP-constraints are capable of expressing a wide range of value-sets, but neither subsumes the other. We give examples where one constraint language can differentiate between a pair of values, and the other cannot.

Example 5 Let $v_1 = (\mathsf{T} : [])$ and $v_2 = (\mathsf{T} : \mathsf{T} : [])$ and consider the MP-constraint $\{(:)\ \mathsf{Any}\} \star \{[]\}$. This constraint is satisfied by v_1 but not by v_2. No proposition over RE-constraints can separate these two values. □

Example 6 Consider a data type:

```
data Tree α = Branch{left :: Tree α, right :: Tree α}
            | Leaf   {leaf :: α}
```

and two values of the type Tree Bool

v_1 = Branch (Leaf True) (Leaf False)
v_2 = Branch (Leaf False) (Leaf True)

The RE-constraint (left*·leaf ⇝ True) is satisfied by v_1 but not v_2. No MP-constraint separates the two values. □

We have implemented both constraint systems in CATCH. There are various factors to consider when choosing which constraint system to use – how readable the constraints are, the expressive power, implementation complexity and scalability. In practice the issue of scalability is key: how large do constraints become, how quickly can they be manipulated, how expensive is their simplification. CATCH07 uses MP-constraints by default, as they allow much larger examples to be checked.

5. Converting Haskell to a First-order Core

The full Haskell language is unwieldy for analysis. As noted in §3.1, analysis is performed instead on a simplified language, a core to which other Haskell programs can be reduced.

5.1 Yhc Core

To generate core representations of programs, it is natural to start with a full Haskell compiler, and we chose Yhc (Golubovsky et al. 2007), a fork of nhc (Röjemo 1995). The core language of Yhc, PosLambda, was intended only as an internal representation, and exposes certain details that are specific to the compiler. We have therefore introduced a new Core language to Yhc, to which PosLambda can easily be translated. All names are fully qualified. Haskell's type classes have been removed (see §5.2). Only top-level functions remain; all local functions have been lambda lifted. All constructor applications are fully saturated. Pattern matching occurs only in case expressions; alternatives match only the top level constructor and are exhaustive, including an error alternative if necessary.

5.2 The Dictionary Transformation

Most transformations in Yhc operate within a single function definition. The only phases which require information about more than one function are type checking and the transformation used to implement type classes (Wadler and Blott 1989). The dictionary transformation introduces tuples (or *dictionaries*) of methods passed as additional arguments to class-polymorphic functions. Haskell also allows subclassing. For example, Ord requires Eq for the same type. In such cases the dictionary transformation generates a nested tuple: the Eq dictionary is a component of the Ord dictionary.

Example 7

$\mathsf{f} :: \mathsf{Eq}\ \alpha \Rightarrow \alpha \to \alpha \to \mathsf{Bool}$
$\mathsf{f}\ x\ y = x \equiv y \mid\mid x \not\equiv y$

is translated by Yhc into

$\mathsf{f} :: (\alpha \to \alpha \to \mathsf{Bool}, \alpha \to \alpha \to \mathsf{Bool}) \to \alpha \to \alpha \to \mathsf{Bool}$
$\mathsf{f}\ \mathsf{dict}\ x\ y = (\mid\mid)\ (((\equiv)\ \mathsf{dict})\ x\ y)\ (((\not\equiv)\ \mathsf{dict})\ x\ y)$

$(\equiv)\ (a, b) = a$
$(\not\equiv)\ (a, b) = b$

The Eq class is implemented as two selector functions, $(\equiv)$ and $(\not\equiv)$, acting on a method table. For different types of α, different method tables are provided. □

The dictionary transformation is a global transformation. In Example 7 the Eq context in f not only requires a dictionary to be accepted by f; it requires all the callers of f to pass a dictionary as first argument. An alternative approach to implementing type classes, given in Jones (1994), does not rely on higher order functions. Although this approach might suit CATCH better, we re-used the method already implemented in Yhc.

5.3 First-Order Haskell

The analysis presented in §3 operates on a first-order language. In order to analyse full Haskell, we transform Haskell to a first-order language. We briefly consider three alternative methods.

5.3.1 Reynolds style defunctionalization

Reynolds style defunctionalization (Reynolds 1972) is the seminal method for generating a first-order equivalent of a higher-order program.

Example 8

$\mathsf{map}\ fn\ x = \mathbf{case}\ x\ \mathbf{of}$
$\quad [\,] \to [\,]$
$\quad (a : as) \to fn\ a : \mathsf{map}\ fn\ as$

Defunctionalization works by creating a data type to represent all values that fn may take anywhere in the whole program. For instance, it might be:

$\mathbf{data}\ \mathsf{Functions} = \mathsf{Head} \mid \mathsf{Tail}$

$\mathsf{apply\ Head}\ x = \mathsf{head}\ x$
$\mathsf{apply\ Tail}\ \ x = \mathsf{tail}\ \ x$

$\mathsf{map}\ fn\ x = \mathbf{case}\ x\ \mathbf{of}$
$\quad [\,] \to [\,]$
$\quad (a : as) \to \mathsf{apply}\ fn\ a : \mathsf{map}\ fn\ as$

Now all calls to map head are replaced by map Head. □

Defunctionalized code is still type safe, but type checking would require a dependently typed language. This presents no problem for CATCH, which does not use type information. The unacceptable aspect is the creation of an apply function, whose meaning is excessively general, introducing a bottleneck through which various properties must be proven. Asking questions such as *"Is the result of apply an empty-list?"*, requires a lot of computation.

CATCH only uses Reynolds style defunctionalization if all other methods fail.

5.3.2 Specialisation

CATCH05 uses a different technique to remove higher-order functions: specialisation. A mutually recursive group of functions can be specialised if one argument is always passed between functions unmodified. Examples of common functions whose applications can be specialised include map, filter, foldr and foldl. When a function can be specialised, the expression passed as the invariant argument has all its free variables passed as extra arguments, and is expanded in the specialised version. All recursive calls within the new function are then renamed.

Example 9

$\mathsf{adds}\ x\ n = \mathsf{map}\ ((+)\ n)\ x$
$\mathsf{map}\ fn\ xs = \mathbf{case}\ xs\ \mathbf{of}$
$\quad [\,] \to [\,]$
$\quad (a : as) \to fn\ a : \mathsf{map}\ fn\ as$

is transformed into:

$\mathsf{adds}\ x\ n = \mathsf{map\_adds}\ n\ x$
$\mathsf{map\_adds}\ n\ xs = \mathbf{case}\ xs\ \mathbf{of}$
$\quad [\,] \to [\,]$
$\quad (a : as) \to (+)\ n\ a : \mathsf{map\_adds}\ n\ as$ □

Specialisation alone is sufficient for many examples, but it cannot cope with point-free code, and does not deal with many forms of dictionaries.

5.3.3 Specialisation with Inlining

The power of specialisation is greatly increased if it is combined with inlining, and applied selectively to higher-order functions.

In order to permit a higher-order program to be represented, the Core language is augmented with additional constructs, as shown in Figure 14. The Apply constructor represents an unsaturated function call, or a variable to be used as the function. The Part constructor is used to represent unsaturated function calls, leaving the normal Call constructor to represent saturated calls. A Part construction records how many arguments are needed.

The algorithm for removing higher-order functions has two components, specialisation and inlining. We apply the specialise rule until a fixed point is reached, then apply the inline rule once. We repeat these two steps until a fixed point is reached. Given an appropriate fix function, firstify can be implemented as:

$\mathsf{firstify} :: \mathsf{Program} \to \mathsf{Program}$
$\mathsf{firstify} = \mathsf{fix}\ (\mathsf{inline} \circ \mathsf{fix\ specialise})$

The inline stage inlines each Call for which the body passes the isHO test, defined in Figure 15. If this process causes a function to no longer be called from the root of the program, then the function is removed after inlining.

$\mathbf{data}\ \mathsf{Expr} = \ldots$ -- as in Figure 2
$\quad \mid \mathsf{Part}\ \ \mathsf{Int}\ \mathsf{FuncName}\ [\mathsf{Expr}]$
$\quad \mid \mathsf{Apply\ Expr}\ [\mathsf{Expr}]$

-- equivalences
$\mathsf{Part}\ 0\ fn\ xs \equiv \mathsf{Call}\ fn\ xs$
$\mathsf{Apply}\ (\mathsf{Part}\ n\ fn\ xs)\ ys \equiv \mathsf{Part}\ (n - \mathsf{length}\ ys)\ fn\ (xs +\!\!+ ys)$

Figure 14. Augmented Core syntax.

$\mathsf{isHO} :: \mathsf{Expr} \to \mathsf{Bool}$
$\mathsf{isHO}\ (\mathsf{Part}\ n\ \_\ \_) = n{>}0$
$\mathsf{isHO}\ (\mathsf{Make}\ \_\ xs) = \mathsf{any\ isHO}\ xs$
$\mathsf{isHO}\ (\mathsf{Case}\ on\ alts) = \mathsf{any}\ (\mathsf{isHO} \circ \mathsf{snd})\ alts$
$\mathsf{isHO}\ \_ = \mathsf{False}$

Figure 15. Tests for the firstifier.

The specialise stage takes every expression of the form Call *fn xs* where any isHO *xs*, and generates a specialised version of the function *fn* with all functional arguments in *xs* frozen in, and all others passed normally.

In rare circumstances a program may have an infinite number of specialisations. If necessary we revert to Reynolds style defunctionalization.

The combination of specialisation and inlining is powerful. We have encountered few examples where it fails – mainly artificial tests created specifically to break the approach! In the definition of a higher-order function either the entire body is an application of the functional argument (in which case it will be inlined), or it must occur as the argument to a function (in which case it is specialised). There are only two places left for functional arguments to be used: (1) As the subject of a **case** expression. But this situation is impossible as all **case** expressions must choose over a data value. (2) Inside an Apply with another functional argument variable as the function. This situation is rare due to the removal of other functional arguments.

6. Results and Evaluation

The best way to see the power of CATCH is by example. §6.1 discusses in general how some programs may need to be modified to obtain provable safety. §6.2 investigates all the examples from the Imaginary section of the Nofib suite (Partain et al. 2007). To illustrate results for larger and widely-used applications, §6.3 investigates a library (FiniteMap) and §6.4 investigates a complete program (HsColour).

6.1 Modifications for Verifiable Safety

Take the following example:

```
average xs = sum xs `div` length xs
```

If *xs* is [] then a division by zero occurs, modelled in CATCH as a pattern-match error. One small local change could be made which would remove this pattern match error:

```
average xs = if null xs then 0 else sum xs `div` length xs
```

Now if *xs* is [], the program simply returns 0, and no pattern match error occurs. In general, pattern-match errors can be avoided in two ways:

Widen the domain of definition: In the example, we widen the domain definition for the average function. The modification is made in one place only – in the definition of average itself. An alternative approach to the average example would be to widen the domain of definition for division, using a variant such as:

```
safeDiv x y = if y ≡ 0 then 0 else x / y
```

Narrow the domain of application: In the example, we narrow the domain of application for the (/) function. Note that we narrow this domain only for the (/) application in average – other (/) applications may remain unsafe. Another alternative would be to narrow the domain of application for average, ensuring that [] is not passed as the argument. This alternative would require a deeper understanding of the flow of the program, requiring rather more work.

In the following sections, where modifications are required, we prefer to make the minimum number of changes. Consequently, we widen the domain of definition.

6.2 Nofib Benchmark Tests

The entire Nofib suite (Partain et al. 2007) is large. We concentrate on the 'Imaginary' section. These programs are all under a page of text, excluding any Prelude or library definitions used, and particularly stress list operations and numeric computations.

Table 1. Table of results

Name is the name of the checked program (a starred name indicates that changes were needed before safe pattern-matching could be verified); **Src** is the number of lines in the original source code; **Core** is the total number of lines of Yhc Core, including all functions used from libraries; **First** is the number of lines after firstification, just before analysis; **Err** is the number of calls to error (missing pattern cases); **Pre** is the number of functions which have a precondition which is not simply 'True'; **Sec** is the time taken for transformations and analysis; **Mb** is the maximum residency of CATCH at garbage collection time.

Name	Src	Core	First	Err	Pre	Sec	Mb
Bernoulli*	35	1616	652	5	11	4.1	0.8
Digits of E1*	44	957	377	3	8	0.3	0.6
Digits of E2	54	1179	455	5	19	0.5	0.8
Exp3-8	29	220	163	0	0	0.1	0.1
Gen-Regexps*	41	1006	776	1	1	0.3	0.4
Integrate	39	2466	364	3	3	0.3	1.9
Paraffins*	91	2627	1153	2	2	0.8	1.9
Primes	16	302	241	6	13	0.2	0.1
Queens	16	648	283	0	0	0.2	0.2
Rfib	9	1918	100	0	0	0.1	1.7
Tak	12	209	155	0	0	0.1	0.1
Wheel Sieve 1*	37	1221	570	7	10	7.5	0.9
Wheel Sieve 2*	45	1397	636	2	2	0.3	0.6
X2n1	10	2637	331	2	5	1.8	1.9
FiniteMap*	670	1484	1829	13	17	1.6	1.0
HsColour*	823	5379	5060	4	9	2.1	2.7

Results are given in Table 1. Only four programs contain no calls to error as all pattern-matches are exhaustive. Four programs use the list-indexing operator (!!), which requires the index to be non-negative and less than the length of the list; CATCH can only prove this condition if the list is infinite. Eight programs include applications of either head or tail, most of which can be proven safe. Seven programs have incomplete patterns, often in a **where** binding and CATCH performs well on these. Nine programs use division, with the precondition that the divisor must not be zero; most of these can be proven safe.

Three programs have preconditions on the main function, all of which state that the argument must be a natural number. In all cases the generated precondition is a necessary one – if the input violates the precondition then pattern-match failure will occur.

We now discuss general modifications required to allow CATCH to begin checking the programs, followed by the six programs which required changes. We finish with the Digits of E2 program – a program with complex pattern matching that CATCH is able to prove safe.

Modifications for Checking Take a typical benchmark, Primes. The main function is:

```
main = do [arg] ← getArgs
          print $ primes !! (read arg)
```

The first unsafe pattern here is [arg] ← getArgs, as getArgs is a primitive which may return any value. CATCH can only return False as the sufficient precondition to main!

The next step that may fail is when read is applied to an argument extracted from getArgs. This argument is entirely unknown, and read is a sufficiently complicated function that although it can be modelled by CATCH, there is no possibility for getting an ap-

propriate abstraction which models the failure case. As a result, any program which calls read is unsafe, according to CATCH. Using reads, which indicates failure properly, a program may still be checked successfully.

As a result the programs have been rewritten, and CATCH is directed to check the function:

```
compute x = print $ primes !! x
```

Bernoulli This program has one instance of tail (tail x). MP-constraints are unable to express that a list must be of at least length two, so CATCH conservatively strengthens this to the condition that the list must be infinite – a condition that Bernoulli does not satisfy. One remedy is to replace tail (tail x) with drop 2 x. After this change, the program still has several non-exhaustive pattern matches, but all are proven safe.

Digits of E1 This program contains the following equation:

```
ratTrans (a, b, c, d) xs |
  ((signum c ≡ signum d) || (abs c<abs d)) &&
  (c+d)*q ⩽ a+b && (c+d)*q+(c+d)>a+b
    = q : ratTrans (c, d, a−q*c, b−q*d) xs
  where q = b `div` d
```

CATCH is able to prove that the division by d is only unsafe if both c and d are zero, but it is not able to prove that this invariant is maintained. Using safeDiv from §6.1 the program is proved safe.

As the safety of this program depends on quite deep results in number theory, it is no surprise that it is beyond the scope of an automatic checker such as CATCH.

Gen-Regexps This program expects valid regular expressions as input. Ways of crashing the program include entering `""`, `"["`, `"<"` and lots of other inputs. One potential error comes from head∘lines, which can be replaced by takeWhile ($\not\equiv$ `'\n'`). Two potential errors take the form $(a, \_ : b) =$ span f xs. At first glance this pattern definition is similar to the one in risers. But here the pattern is only safe if for one of the elements in the list xs, f returns True. The test f is actually ($\not\equiv$ `'-'`), and the only safe condition CATCH can express is that xs is an infinite list. With the amendment $(a, b) =$ safeSpan f xs, where safeSpan is defined as:

```
safeSpan p xs = (a, drop 1 b) where (a, b) = span p xs
```

CATCH verifies pattern safety.

Wheel Sieve 1 This program defines a data type Wheel, and a function sieve:

```
data Wheel = Wheel Int [Int]

sieve :: [Wheel] → [Int] → [Int] → [Int]
```

The lists are infinite, and the integers are positive, but the program is too complex for CATCH to infer these properties in full. To prove safety a variant of mod is required which does not raise division by zero and a pattern in notDivBy has to be completed. Even with these two modifications, CATCH takes 7.5 seconds to check the other non-exhaustive pattern matches.

Wheel Sieve 2 This program has similar datatypes and invariants, but much greater complexity. CATCH is able to prove very few of the necessary invariants. Only after widening the domain of definition in three places – replacing tail with drop 1, head with a version returning a default on the empty list, and mod with a safe variant – is CATCH able to prove safety.

Paraffins Again the program can only be validated by CATCH after modification. There are two reasons: laziness and arrays. Laziness allows the following odd-looking definition:

```
radical_generator n = radicals undefined
  where radicals unused = big_memory_computation
```

If radicals had a zero-arity definition it would be computed once and retained as long as there are references to it. To prevent this behaviour, a dummy argument (undefined) is passed. If the analysis was more lazy (as discussed in §3.4) then this example would succeed using CATCH. As it is, simply changing undefined to () resolves the problem.

The Paraffins program uses the function array :: Ix $a \Rightarrow (a, a) \to [(a, b)] \to$ Array a b which takes a list of index/value pairs and builds an array. The precondition on this function is that all indexes must be in the range specified. This precondition is too complex for CATCH, but simply using listArray, which takes a list of elements one after another, the program can be validated. Use of listArray actually makes the program shorter and more readable. The array indexing operator (!) is also troublesome. The precondition requires that the index is in the bounds given when the array was constructed, something CATCH does not currently model.

Digits of E2 This program is quite complex, featuring a number of possible pattern-match errors. To illustrate, consider the following fragment:

```
carryPropagate base (d : ds) = ...
  where carryguess = d `div` base
        remainder = d `mod` base
        nextcarry : fraction = carryPropagate (base+1) ds
```

There are four potential pattern-match errors in as many lines. Two of these are the calls to div and mod, both requiring base to be non-zero. A possibly more subtle pattern match error is the nextcarry : fraction left-hand side of the third line. CATCH is able to prove that none of these pattern-matches fail. Now consider:

```
e = ("2."++) $
    tail ∘ concat $
    map (show ∘ head) $
    iterate (carryPropagate 2 ∘ map (10*) ∘ tail) $
    2 : [1, 1 ..]
```

Two uses of tail and one of head occur in quite complex functional pipelines. CATCH is again able to prove that no pattern-match fails.

6.3 The FiniteMap library

The FiniteMap library for Haskell has been widely distributed for over 10 years. The library uses balanced binary trees, based on (Adams 1993). There are 14 non-exhaustive pattern matches.

The first challenge is that there is no main function. CATCH uses all the exports from the library, and checks each of them as if it had main status.

CATCH is able to prove that all but one of the non-exhaustive patterns are safe. The definition found unsafe has the form:

```
delFromFM (Branch key ...) del_key | del_key > key = ...
                                   | del_key < key = ...
                                   | del_key ≡ key = ...
```

At first glance the cases appear to be exhaustive. The law of trichotomy leads us to expect one of the guards to be true. However, the Haskell Ord class does not enforce this law. There is nothing to prevent an instance for a type with partially ordered values, some of which are incomparable. So CATCH cannot verify the safety to delFromFM as defined as above.

The solution is to use the compare function which returns one of GT, EQ or LT. This approach has several advantages: (1) the code is free from non-exhaustive patterns; (2) the assumption of trichotomy is explicit in the return type; (3) the library is faster.

6.4 The HsColour Program

Artificial benchmarks are not necessarily intended to be fail-proof. But a real program, with real users, should *never* fail with a pattern-match error. We have taken the HsColour program[3] and analysed it using CATCH. HsColour has 12 modules, is 4 years old and has had patches from 6 different people. We have contributed patches back to the author of HsColour, with the result that the development version can be proved free from pattern-match errors.

CATCH required 4 small patches to the HsColour program before it could be verified free of pattern-match failures. Details are given in Table 1. Of the 4 patches, 3 were genuine pattern-match errors which could be tripped by constructing unexpected input. The issues were: (1) read was called on a preferences file from the user, this could crash given a malformed preferences file; (2) by giving the document consisting of a single double quote character ", and passing the "-latex" flag, a crash occurred; (3) by giving the document (‘), namely open bracket, backtick, close bracket, and passing "-html -anchor" a crash occurred. The one patch which did not (as far as we are able to ascertain) fix a real bug could still be considered an improvement, and was minor in nature (a single line).

Examining the read error in more detail, by default CATCH outputs the potential error message, and a list of potentially unsafe functions in a call stack:

```
Checking "Prelude.read: no parse"
Partial Prelude.read$252
Partial Language.Haskell.HsColour.Colourise.parseColourPrefs
...
Partial Main.main
```

Using our knowledge of the intention of functions, we can see that although read calls error, the blame probably lies in the caller of read – namely parseColourPrefs. By examining this location in the source code we are able to diagnose and correct the problem. CATCH optionally reports all the preconditions it has deduced, although in our experience problems can usually be fixed from source-position information alone.

7. Related Work

7.1 Mistake Detectors

There has been a long history of writing tools to analyse programs to detect potential bugs, going back at least to the classic C Lint tool (Johnson 1978). In the functional arena there is the Dialyzer tool (Lindahl and Sagonas 2004) for Erlang (Virding et al. 1996). The aim is to have a static checker that works on unmodified code, with no additional annotations. However, a key difference is that in Dialyzer all warnings indicate a genuine problem that needs to be fixed. Because Erlang is a dynamically typed language, a large proportion of Dialyzer's warnings relate to mistakes a type checker would have detected.

7.2 Proving Incomplete Patterns Safe

Despite the seriousness of the problem of pattern matching, there are very few other tools for checking pattern-match safety. This paper has similar goals to Mitchell and Runciman (2007), but some key design decisions are radically different. The difference in practice is that CATCH07 supports full Haskell, can scale to much larger examples and can feasibly be used on real programs. Some of the reasons for these better results include a different fixed-point mechanism, the use of MP-constraints and a superior translation from Core.

[3] http://www.cs.york.ac.uk/fp/darcs/hscolour/

```
data Cons
data Unknown

newtype List a t = List [a]

cons :: a → [a] → List a Cons
cons a as = List (a : as)

nil :: List a Unknown
nil = List []

fromList :: [a] → List a Unknown
fromList xs = List xs

safeTail :: List a Cons → [a]
safeTail (List (a : as)) = as
```

Figure 16. A safeTail function with Phantom types.

The closest other work we are aware of is (Xu 2006), where ESC/Haskell is introduced. The ESC/Haskell approach requires the programmer to give explicit preconditions and contracts which the program obeys. Contracts have more expressive power than our constraints – one of the examples involves an invariant on an ordered list, something beyond CATCH. But the programmer has more work to do. At the time of writing there is no publicly available version of ESC/Haskell. So far as we know, it does not yet support full Haskell, lacks features such as type classes[4], and handles only small examples. A future version of ESC/Haskell may address these limitations, allowing a fuller comparison to be made.

7.3 Eliminating Incomplete Patterns

One way to guarantee that a program does not crash with an incomplete pattern is to ensure that all pattern matching is exhaustive. The GHC compiler (The GHC Team 2006) has an option flag to warn of any incomplete patterns. Unfortunately the Bugs section (12.2.1) of the manual notes that the checks are sometimes wrong, particularly with string patterns or guards, and that this part of the compiler "needs an overhaul really" (The GHC Team 2006). A more precise treatment of when warnings should be issued is given in Maranget (2007). These checks are only local: defining head will lead to a warning, even though the definition is correct; using head will not lead to a warning, even though it may raise a pattern-match error.

A more radical approach is to build exhaustive pattern matching into the design of the language, as part of a total programming system (Turner 2004). The CATCH tool could perhaps allow the exhaustive pattern matching restriction to be lifted somewhat.

7.4 Type System Safety

One method for specifying properties about functional programs is to use the type system. This approach is taken in the tree automata work done on XML and XSLT (Tozawa 2001), which can be seen as an algebraic data type and a functional language. Another soft typing system with similarities is by Aiken and Murphy (1991), on the functional language FL. This system tries to assign a type to each function using a set of constructors, for example head takes the type Cons and not Nil.

Types can sometimes be used to explicitly encode invariants on data in functional languages. One approach is the use of *phantom types* (Fluet and Pucella 2002), for example a safe variant of tail can be written as in Figure 16. The List type is not exported, ensuring that all lists with a Cons tag are indeed non-empty. The types Cons

[4] We have confirmed this information in correspondence with Xu.

```
data ConsT a
data NilT

data List a t where
  Cons :: a → List a b → List a (ConsT b)
  Nil  :: List a NilT

safeTail :: List a (ConsT t) → List a t
safeTail (Cons a b) = b

fromList :: [a] → (∀ t • List a t → r) → r
fromList []       fn = fn Nil
fromList (x : xs) fn = fromList xs (fn ∘ Cons x)
```

Figure 17. A safeTail function using GADTs.

and Unknown are phantom types – they exist only at the type level, and have no corresponding value.

Using GADTs (Peyton Jones et al. 2006), an encoding of lists can be written as in Figure 17. Notice that fromList requires a locally quantified type. The type-directed approach can be pushed much further with *dependent types*, which allow types to depend on values. There has been much work on dependent types, using undecidable type systems (McBride and McKinna 2004), using extensible kinds (Sheard 2004) and using type systems restricted to a decidable fragment (Xi and Pfenning 1999). The downside to all these type systems is that they require the programmer to make explicit annotations, and require the user to learn new techniques for computation.

8. Conclusions and Future Work

We have described the design, implementation and application of CATCH, an analysis tool for safe pattern-matching in Haskell 98. Two key design decisions in CATCH simplify the analysis and make it scalable: (1) the target of analysis is a very small, first-order core language; (2) there are finitely many value-set-defining constraints per type. Decision (1) requires a translation from the full language that avoids the introduction of analysis bottlenecks such as an interpretive apply operator; the combined use of inlining and specialisation has proved very effective. Decision (2) inevitably limits the expressive power of constraints; yet it does not prevent the expression of uniform recursive constraints on the deep structure of values, as in MP-constraints.

Practical evaluation, using CATCH to analyse widely distributed examples in Haskell 98, confirms our claim to give results for programs of moderate size written in the full language. But it does also reveal a frequent need to modify programs, widening definitions or narrowing applications, before CATCH can verify pattern-match safety.

Outcomes of example applications could drive the exploration of more powerful variants of MP-constraints, with a greater (but still finite) number of expressible constraints per type. More demanding tests of scalability could include the application of CATCH to a Haskell compiler, or indeed to CATCH itself.

A tool such as CATCH might do more harm than good if it sometimes wrongly declares that a program cannot fail. We claim that the CATCH analysis is sound, but we have not formally proved soundness with respect to a suitable semantics for the core language.

Like many researchers, we are interested in narrowing the gap between the exactness of constructive mathematics and the scalability of practical programming systems. We hope that CATCH or its successors can provide a small but useful bridge crossing part of that gap.

References

Stephen Adams. Efficient sets – a balancing act. *JFP*, 3(4):553–561, 1993.

Alex Aiken and Brian Murphy. Static Type Inference in a Dynamically Typed Language. In *Proc. POPL '91*, pages 279–290. ACM Press, 1991.

John Horton Conway. *Regular Algebra and Finite Machines*. London Chapman and Hall, 1971.

Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *Proc. TCS '02*, pages 448–460, Deventer, The Netherlands, 2002. Kluwer, B.V.

Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. *The Monad.Reader*, (7):45–61, April 2007.

S. C. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1978.

Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *Proc. PEPM '94*, pages 107–117. ACM Press, June 1994.

C. Lee. Representation of switching circuits by binary decision diagrams. *Bell System Technical Journal*, 38:985–999, 1959.

Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Proc. APLAS '04, LNCS 3302*, pages 91–106. Springer, November 2004.

Luc Maranget. Warnings for pattern matching. *JFP*, 17(3), May 2007.

Conor McBride and James McKinna. The view from the left. *JFP*, 14(1): 69–111, 2004.

Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming (2005 Symposium)*, volume 6, pages 15–30. Intellect, 2007.

Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2007.

Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. *Proc. ICFP '06*, pages 50–61, 2006.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM '72*, pages 717–740, New York, NY, USA, 1972. ACM Press.

Niklas Röjemo. Highlights from nhc - a space-efficient Haskell compiler. In *Proc. FPCA '95*, pages 282–292. ACM Press, 1995.

David Roundy. Darcs: distributed version management in Haskell. In *Proc. Haskell '05*, pages 1–4. ACM Press, 2005.

Tim Sheard. Languages of the future. In *Proc. OOPSLA '04*, pages 116–119. ACM Press, 2004.

The GHC Team. The GHC compiler, version 6.6. `http://www.haskell.org/ghc/`, October 2006.

Andrew Tolmach. An External Representation for the GHC Core Language. `http://www.haskell.org/ghc/docs/papers/core.ps.gz`, September 2001.

Akihiko Tozawa. Towards Static Type Checking for XSLT. In *Proc. DocEng '01*, pages 18–27, New York, NY, USA, 2001. ACM Press.

David Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, July 2004.

Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, second edition, 1996.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM Press.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. POPL '99*, pages 214–227. ACM Press, 1999.

Dana N. Xu. Extended static checking for Haskell. In *Proc. Haskell '06*, pages 48–59. ACM Press, 2006.