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
test 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 func-
tions over algebraic data types, matching arguments against alterna-
tive constructor patterns. In the most widely used languages, such
as Haskell and ML, alternative patterns need not exhaust all possi-
ble 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
dans project (Roundy 2005) is one of the most successful large
scale programs written in Haskell. Taking a look at the dans 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 Runci-
man (2007):
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)

A sample application of this function is:
> risers [1,2,3,1,2]
[[1,2,3],[1,2]]

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 Hackage1.

1.1 Contributions
The contributions of this paper include:
• A method for reasoning about pattern-match failures, in terms
  of a parameterisable constraint language. The method calcu-
lates preconditions of functions.
• Two separate constraint languages that can be used with our
  method.
• Details of the Catch implementation which handles the com-
  plete 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. 2008), and for three larger
  examples, investigating the scalability of the checker.

∗ The first author is supported by an EPSRC PhD studentship

[Copyright notice will appear here once 'preprint' option is removed.]
risers \( \times = \text{case } x \text{ of} \)
\[ [ ] \rightarrow [ ] \]
\( (y:ys) \rightarrow \text{case } ys \text{ of} \)
\[ [ ] \rightarrow (y:[]):[] \]
\( (z:zs) \rightarrow \text{risers2 } (\text{risers3 } z zs) \) \((y \leq z) y\)

risers2 \( \times y z = \text{case } y \text{ of} \)
\[ \text{True} \rightarrow (z : \text{snd } x) : (\text{fst } x) \]
\[ \text{False} \rightarrow (z : []) : (\text{snd } x : \text{fst } x) \]

risers3 \( x y = \text{risers4 } (\text{risers } (x : y)) \)

risers4 \( x = \text{case } x \text{ of} \)
\[ (y:ys) \rightarrow (ys, y) \]
\[ [ ] \rightarrow \text{error "Pattern Match Failure, 11:12."} \]

---

**Figure 1.** risers in the Core language.

In Mitchell and Runciman (2007) a pattern-match checker was described with similar aims, which we will refer to as Catch05. For Catch05 risers was towards the boundary of what was possible, for Catch08 it is trivial. Scalability problems have been addressed and Catch08 handles real Haskell programs, with no restriction on the recursion patterns, higher-order functions or type classes. The underlying constraint mechanism of Catch08 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 evaluates Catch08 on programs from the Nofib suite, on a widely-used library and on a larger application program. §6 offers comparisons with related work before §7 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 §3.1.2. The result of converting the risers program to Core Haskell, with identifiers renamed for ease of human reading, is shown in Figure 1.

The type of risers is polymorphic over types in the Ord class. 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. To keep the example simple, we have chosen the latter.

#### 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 constructors 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):

\[
\text{risers4, } x<(\cdot)\]

---

2 Although the paper was completed in 2005, publication was delayed.

---

**Figure 2.** Core Data Type.

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

\[
x<(\cdot) \Rightarrow (\text{risers } x )<(\cdot)
\]

\[
\text{True} \Rightarrow (\text{risers2 } x y z)<(\cdot)
\]

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

### 3. Pattern Match Analysis

This section describes the method used to calculate preconditions for functions. We first give the Core language for our tool in §3.1, then some essential operations on constraints and propositions in §3.2. We then introduce a simple constraint language in §3.3, which we use to illustrate our method. First we define three terms:

- A constraint describes a (possibly infinite) set of values. We say a value satisfies a constraint if the value is within the set.
- A precondition is a proposition combining constraints on the arguments to a function, to ensure the result does not contain \( \bot \). For example, the precondition on tail \( xs \) is that \( xs \) is \((\cdot)\)-constructed.
- An entailment is a proposition combining constraints on the arguments to a function, to ensure the result satisfies a further constraint. For example, \( xs \) is \((\cdot)\)-constructed ensures \( null \) \( xs \) evaluates to False.

#### 3.1 Reduced expression language

The syntax for our Core language is given in Figure 2. Our Core language is 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 expressions to values, written in Haskell. The `hnf` function evaluates an expression to head normal form. The `subst` function substitutes free variables that are the result of a case expression.

#### 3.1.1 Operations on Core

Figure 4 gives the signatures for helper functions over the core data types. In our implementation, these operations are monadic to allow the checking process to be traced – a detail we have omitted here.
data Value = Bottom | Value CtorName [Value]

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

hnf :: Expr → Maybe (CtorName, [Expr])
hnf (Make c xs) = Just (c, xs)
hnf (Call fn xs) |
  fn ≡ "error"   = Nothing
  otherwise      = hnf (subst (zip (args fn) xs) (body fn))
hnf (Case on alts) = listToMaybe |res
  | Just (c, xs) ← [hnf on], Alt n vs e ← alts, c =≡ n
  |    Just res ← [hnf (subst (zip vs xs) e)]

subst :: [(VarName, Expr)] → Expr → Expr
subt r (Var x) = fromMaybe (Var x) (lookup x r)
subt r (Make x cs) = Make x (map (subt r) cs)
subt r (Call x cs) = Call x (map (subt r) cs)
subt r (Case x cs) = Case (subt r x)
  | Alt n vs (subt r' e) | Alt n vs e ← xs
  | let r' = filter ([lx vs) o fst) r |

data Prop α
(∧), (\v) :: Prop α → Prop α → Prop α
andP, orP :: [Prop α] → Prop α
mapP :: (α → Prop β) → Prop α → Prop β
true, false :: Prop α
bool :: Bool → Prop α
lit :: α → Prop α

Figure 5. Proposition data type.

data Sat α = Sat α Constraint
(<) :: α → CtorName → Prop (Sat α)
(>) :: Selector → Constraint → Constraint
(<) :: CtorName → Constraint → Prop (Sat ArgPos)

Figure 6. Constraint operations.

man 2008), which removes higher-order functions without complicating the analysis. Using these steps, we are able to convert the full Haskell 98 language into our Core language.

3.1.3 Algebraic Abstractions of Primitive Types
Our Core language only has algebraic data types. Catch allows for primitive types such as characters and integers by abstracting them into algebraic types. Two abstractions used in Catch are:

data Int = Neg | Zero | One | Pos

data Char = Char

Where knowledge about values is encoded as a set of possible constructions. In our experience, integers 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 Int abstraction models the common properties and operations quite well. For characters, we have found little benefit in any refinement other than considering all characters to be abstracted to the same value.

The final issue of abstraction relates to primitive functions in the IO monad, such as getArgs (which returns the command-line arguments), or read (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
We write Sat x c to assert that the value of expression x must be a member of the set described by the constraint c, i.e. that x satisfies c. If any component of x evaluates to ⊥, the constraint is automatically satisfied: in our method, for a component of x to evaluate to ⊥, some other constraint must have been violated, so an error is still reported. Atomic constraints can be combined into propositions, using the proposition data type in Figure 5.

Several underlying constraint models are possible. To keep the introduction of the algorithms simple we first use basic pattern constraints (§3.3), which are unsuitable for reasons given in §3.7. We then describe regular expression constraints in §4.1 – a variant of the constraints used in Catch08. Finally we present multi-pattern constraints in §4.2 – used in Catch08 to enable scaling to much larger problems.

Three operations must be provided by every constraint model, whose signatures are given in Figure 6. The lifting and splitting operators (> ) and (< ) are discussed in §3.5. The expression x<cs

3 2008/4/2
3.3 Basic Pattern (BP) Constraints

For simplicity, our analysis framework will be introduced using basic pattern constraints (BP-constraints). BP-constraints are defined in Figure 8, and correspond to Haskell pattern matching, where Any represents an unrestricted match. A data structure satisfies a BP-constraint if it matches the pattern. For example, 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.

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

\[
\begin{align*}
\text{a}<\text{x} & : = \text{orP (lit a `Sat` anys x) (x ← x)} \\
\text{where} & \text{ anys x} = \text{Con x (replicate (arity x) Any)}
\end{align*}
\]

So, for example:

\[
\begin{align*}
\text{e}<["\text{True}"] & : = \text{lit (e `Sat` Con "\text{True}" [])} \\
\text{e}<[":"] & : = \text{lit (e `Sat` Con "::" [Any, Any])} \\
\text{e}<[":", "[""] & : = \text{lit (e `Sat` Con "::" [Any, Any])} \lor \text{lit (e `Sat` Con ":["] [Any, Any])}
\end{align*}
\]

3.4 Preconditions for Pattern Safety

Our intention is that for every function, a proposition combining constraints on the arguments forms a precondition to ensure the result does not contain ⊥. The precondition for error is False. A program is safe if the precondition on main is True. Our analysis method derives these preconditions. Given precond which returns the precondition of a function, we can determine the precondition of an expression using the pre function in Figure 9. The intuition behind pre is that in all subexpressions f xs, the arguments xs must satisfy the precondition for f. The only exception is that a case expression is safe if the scrutinee is safe, and each alternative is either safe, or never taken.
reduce :: Prop (Sat Expr) → Prop (Sat ArgPos)
reduce ≝ mapP (λ(Sat x k) → red x k)

red :: Expr → Constraint → Prop (Sat ArgPos)
red (Var x) k ≝ case var x of
  Left i → lit (i ° Sat k)
  Right (on, c) → red on (c ° k)
red (Make c xs ) k ≝ reduce substP xs (c ° k)
red (Call fn xs ) k ≝ reduce substP xs (prePost fn k)
red (Case on alts) k ≝ andP (f c e | Alt c vs e ← alts).
where f c e = reduce (on=ctors c \ [c]) \ ∨ red e k

Figure 11. Specification of constraint reduction, reduce.

Example 1 (revisited)
The precondition for the safeTail function is:
pre' null [xs] ∧ (null xs≡["True"] ∨ pre' tail [xs])
We can use the preconditions computed for tail and null to rewrite the
precondition as:
null xs≡["True"] ∨ xs≡[":"
Now we use an entailment to turn the constraint on null’s result into
a constraint on its argument:
x≡["] ∨ xs≡[":"
Which can be shown to be a tautology.

The Var rule has two alternatives. The first alternative deals with
top-level bound arguments, which are already in the correct form.
The other alternative applies to variables bound by patterns in case
alternatives. It lifts conditions on a bound variable to the scrutinee
top-level bound arguments, which are already in the correct form.

The Make rule deals with an application of a constructor. The ◁
operator splits a constraint on an entire structure into a proposition
combining constraints on each part.
c ◁ Any ≝ true
c ◁ Con c2 xs = bool (c2 ≡ c) \ ∧
  andP (map lit (zipWith Sat [0 ..] xs))

The intuition is that given knowledge of the root constructor of
a data value, we can reformulate the constraint in terms of what the
constructor fields must satisfy. Some sample applications:
"True" ◁ Con "True" [] = true
"False" ◁ Con "True" [] = false
In the next section we introduce two alternative constraint systems. Such a constraint is satisfied by a data structure if every well-defined application to d of a sequence of selectors described by r reaches a constructor in the set cs. If no such sequence of selectors has a well-defined result then the constraint is vacuously true.

Concerning the helper functions needed to define > and in Figure 13, 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, preventing 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 $r_2$
- $r_1 . r_2$ concatenation of regular expressions $r_1$ then $r_2$
- $r_1^*$ any number (possibly zero) occurrences of $r_1$
- $\text{sel}$ a selector, i.e. h for the head of a list
- 0 the language is the empty set
- 1 the language is the set containing the empty string

Catch08 implements REs using the data type RegExp from Figure 13, 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 sufficient 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 $\mapsto c$ always evaluates to True, $<$ replaces Nothing by True.)

Example 3

(\head xs) is safe if xs evaluates to a non-empty list. The RE-constraint generated by Catch is: $\text{xSats} \{ 1 \mapsto \{} \}$. This may be read: from the root of the value xs, after following an empty path of selectors, we reach a (\_)-constructed value.

Example 4

(map head xs) is safe if xs evaluates to a list of non-empty lists. The RE-constraint is: $\text{xSats} \{ \text{tl'} . hd \mapsto \{} \}$. From the root of xs, following any number of tails, then exactly one head, we reach a (\_). If xs is [], it still satisfies the constraint, as the path is not well defined, having no hd selector. If xs is infinite then all its infinitely many elements must be (\_)-constructed.

Example 5

(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: $\text{xSats} \{ \text{tl'} . hd \mapsto \{} \} \lor \text{xSats} \{ \text{tl'} \mapsto \{} \}$. The second term specifies the infinite case: if the list xs is (\_)-constructed, it will have a tl selector, and therefore the tl path is well defined and requires the tail to be (\_;). Each step in the chain ensures the next path is well defined, and therefore the list is infinite.

4. Richer but Finite Constraint Systems

There are many ways of defining a richer constraint system, while also ensuring the necessary finiteness properties. Here we outline two – one adapted from Catch05, one entirely new – both implemented in Catch08. Neither is strictly more powerful than the other; each is capable of expressing constraints that the other cannot express.

When designing a constraint system, the main decision is which distinctions between data values to ignore. Since the constraint system must be finite, there must be sets of data values which no constraint within the system can distinguish between. As the constraint system stores more information, it will distinguish more values, but will likely take longer to obtain fixed points. The two constraint systems in this section were developed by looking at examples, and trying to find systems offering sufficient power to solve real problems, but still remain bounded.

4.1 Regular Expression (RE) Constraints

Catch05 used regular expressions in constraints. An implementation of Catch05 regular expression based constraints (RE-constraints) is given in Figure 13. In a constraint of the form $(r \mapsto cs)$, $r$ is a regular expression and $cs$ is a set of constructors. Such a constraint is satisfied by a data structure if every well-defined application to $d$ of a sequence of selectors described by $r$ reaches a constructor in the set $cs$. If no such sequence of selectors has a well-defined result then the constraint is vacuously true.

Concerning the helper functions needed to define $>$ and $<$ in Figure 13, 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, preventing 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 $r_2$
- $r_1 . r_2$ concatenation of regular expressions $r_1$ then $r_2$
- $r_1^*$ any number (possibly zero) occurrences of $r_1$
- $\text{sel}$ a selector, i.e. h for the head of a list
- 0 the language is the empty set
- 1 the language is the set containing the empty string

Catch08 implements REs using the data type RegExp from Figure 13, 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 sufficient 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 $\mapsto c$ always evaluates to True, $<$ replaces Nothing by True.)

Example 3

(\head xs) is safe if xs evaluates to a non-empty list. The RE-constraint generated by Catch is: $\text{xSats} \{ 1 \mapsto \{} \}$. This may be read: from the root of the value xs, after following an empty path of selectors, we reach a (\_)-constructed value.

Example 4

(map head xs) is safe if xs evaluates to a list of non-empty lists. The RE-constraint is: $\text{xSats} \{ \text{tl'} . hd \mapsto \{} \}$. From the root of xs, following any number of tails, then exactly one head, we reach a (\_). If xs is [], it still satisfies the constraint, as the path is not well defined, having no hd selector. If xs is infinite then all its infinitely many elements must be (\_)-constructed.

Example 5

(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: $\text{xSats} \{ \text{tl'} . hd \mapsto \{} \} \lor \text{xSats} \{ \text{tl'} \mapsto \{} \}$. The second term specifies the infinite case: if the list xs is (\_)-constructed, it will have a tl selector, and therefore the tl path is well defined and requires the tail to be (\_;). Each step in the chain ensures the next path is well defined, and therefore the list is infinite.

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:

\begin{verbatim}
data Constraint = RegExp -> [CtorName]
type RegExp = [RegItem]
data RegItem = Atom Selector | Star [Selector]
\end{verbatim}
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

where rec = length (filter isNothing (concat t))
nonrec = [x | Just x ← concat t]
cctor = length t

The 2 ^ rec term corresponds to the number of possible constraints under Star. The 2 ^ cctor 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 propositions in a proposition, we apply simplification rules to reduce this number. For example, three of the nineteen rules are:

Exhaustion: If in the constraint x ⚫ "Sat" (r → ("":") )) 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 ⚫ "Sat" (r → (c1 ⋈ c2)) can be replaced by e ⚫ "Sat" (r → (c1 ⋈ c2)).

Or merging: The disjunction e ⚫ "Sat" (r → (c1 ⋈ c2)) can be replaced by e ⚫ "Sat" (r → (c1 ⋈ c2)) if c1 ⊆ c2.

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 14) 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 p1 ⋈ p2 if v itself satisfies the pattern p1 and all its recursive components at any depth satisfy p2. We call p1 the root pattern, and p2 the recursive pattern. Each of p1 and p2 is given as a set of matches similar to BP-constraints, but each Pattern only specifies the values for the non-recursive selectors, all recursive selectors are handled by p2. A constraint is a disjunctive list of ⚫ patterns.

The intuition behind the definition of (c, i) ⚫ ps is that if the selector (c, i) is recursive, given a pattern α ⚫ β, the new root pattern requires the value to be c-constructed, and the recursive patterns become merge α β - i.e. all recursive values must satisfy both the root and recursive patterns of the original pattern. If the selector is non-recursive, then each new pattern contains the old pattern within it, as the appropriate non-recursive field. So, for example:

hd ⚫ (α ⚫ β) = {[ (α ⚫ β) ] ⚫ (i : (i : Any))

tl ⚫ (α ⚫ β) = { (i : Any) | (merge α β)
field, and a t field restricted by the constraint on the right of the
* . In this particular case, the right of the * places no restrictions
on the value. This constraint is longer than the corresponding RE-
constraint as it makes explicit that both the head and the recursive
tails are unrestricted.

Example 4 (revisited)
Safe evaluation of (map head xs) requires xs to be a list of non-
empty lists. The MP-constraint on xs is:
\{ [[(\cdot) Any] \ast [\cdot] Any] \ast [\cdot] Any \}
\{ [[(\cdot) Any] \ast [\cdot] Any] \}

Example 5 (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: \{ (\cdot) Any \} \ast
\{ (\cdot) Any \}

MP-constraints also have simplification rules. For example, two
of the eight rules are:
Val-list simplification: Given a Val-list, 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 p1 and p2 cover all constructors and all
their components have Any as their constraint, the constraint p1 \ast p2
can be replaced with Any.

4.2.1 Finitely Many MP-Conestrains per Type
As in §4.1.1, we show there are finitely many constraints per type
by defining a count function:
\begin{align*}
\text{count} & : \text{Type} \to \text{Integer} \\
\text{count} (\text{Type} t) &= 2^\text{\text{val} t} \\
\text{val} t &= 1 + 2 \ast 2^\text{\text{pattern} t} \\
\text{pattern} t &= \text{sum} (\text{map} f t) \\
\text{where} f c &= \text{product} [\text{count} t_2 | \text{Just} t_2 \leftarrow c]
\end{align*}

The val function counts the number of possible Val constructs.
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 atomic constraint:
\begin{align*}
(\text{Sat} e v_1) \lor (\text{Sat} e v_2) &= \text{Sat} e (v_1 \lor v_2) \\
(\text{Sat} e v_1) \land (\text{Sat} e v_2) &= \text{Sat} e [a \land b | a \leftarrow v_1, b \leftarrow v_2]
\end{align*}

This ability to combine constraints on equal expressions can be
exploited further by translating the program to be analysed. After
applying reduce, 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 could be collapsed into
one. We therefore uncurry all functions.

Example 6
\begin{align*}
(\mid) x = \text{case } x \text{ of} \\
& \quad \text{True} \to \text{True} \\
& \quad \text{False} \to \text{y}
\end{align*}
in uncurried form becomes:
\begin{align*}
(\mid) a = \text{case } a \text{ of} \\
& \quad (x, y) \to \text{case } x \text{ of} \\
& \quad \text{True} \to \text{True} \\
& \quad \text{False} \to \text{y}
\end{align*}

Combining MP-constraint reduction rules with the uncurrying
transformation makes Sat ArgPos equivalent in power to Prop (Sat ArgPos).
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, as no additional simplification
rules would apply.

4.3 Comparison of Constraint Systems
As we discussed in §3.7, it is not possible to use BP-constraints, 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 7
Let v1 = (T: [[ ]]) and v2 = (T: [T: [ ]]) and consider the MP-constraint
\{ (\cdot) Any \} \ast \{ [[ ]] \}. This constraint is satisfied by v1 but not by v2.
No proposition over RE-constraints can separate these two values.

Example 8
Consider a data type:
\begin{align*}
\text{data} \quad \text{Tree} \quad \alpha &= \text{Branch} \{ \text{left} :: \text{Tree} \quad \alpha, \text{right} :: \text{Tree} \quad \alpha \} \\
& \quad | \quad \text{Leaf} \quad \{ \text{leaf} :: \alpha \}
\end{align*}

and two values of the type Tree Bool
v1 = Branch \{ \text{Leaf True} \} \quad \text{Leaf False} \\
v2 = \text{Branch} \quad \{ \text{Leaf False} \} \quad \text{Leaf True} \\

The RE-constraint \{ \text{left} (=) \text{leaf} \to \text{True} \} is satisfied by v1 but not v2.
No MP-constraint separates the two values.

We have implemented both constraint systems in Catch. Factors
to consider when choosing which constraint system to use include:
how readable the constraints are, 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. Catch08 uses
MP-constraints by default, as they allow much larger examples to
be checked.

5. Results and Evaluation
The best way to see the power of Catch is by example. §5.1 dis-
cusses in general how some programs may need to be modified to
obtain provable safety. §5.2 investigates all the examples from the
Imaginary section of the Nothib suite (Partain et al. 2008). To il-
strate results for larger and widely-used applications, §5.3 investi-
gates the FiniteMap library, §5.4 investigates the HsColour pro-
gram and §5.5 reports on XMonad.

5.1 Modifications for Verifiable Safety
Take the following example:
\begin{align*}
\text{average} \quad xs &= \text{sum} \quad xs \ \text{div} \ \text{length} \quad xs
\end{align*}
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 $xs = \begin{cases} \text{null } xs & \text{then } 0 \\ \text{else } \text{sum } xs \end{cases} \text{ 'div' length } xs$

Now if $xs \not= \text{[]}$, 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 of definition for the average function. The modification is made in one place only – in the definition of average itself.

Narrow the domain of application: In the example, we narrow the domain of application for the div function. Note that we narrow this domain only for the div application in average – other div 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.

### 5.2 Nofib Benchmark Tests

The entire Nofib suite (Partain et al. 2008) 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.

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 test parameter 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 without modification.

#### Modifications for Checking

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

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

The first unsafe pattern is [arg] <- getArgs, as getArgs is a primitive which may return any value. Additionally, if read fails to parse the value extracted from getArgs, it will evaluate to \_ . Instead, we check the revised program:

```haskell
main = do arg <- getArgs
case map reads args of
[(x, b)] | all isSpace s -> print $ primes !! x
        _ -> putStrLn "Bad command line"
```

Instead of crashing on malformed command line arguments, the modified program informs the user.

**Bernoulli** This program has one instance of tail (tail $xs$). 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 $xs$) with drop 2 $xs$. 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:

```haskell
ratTrans \(a, b, c, d, x) | ((\text{signum } c \equiv \text{signum } d) \land \land (c + d) > q \equiv a + b \land (c + d) | q + (c + d) > a + b = q) \land \text{ratTrans} \(c, d, a - q + c, b - q + d, x) \text{ xs where}
```

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. Widening the domain of application of div allows the program to be 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. There are many ways to crash this program, including entering "", "[" or "]". One potential error comes from head $o$ lines, which can be replaced by takeWhile \((\not= \text{\textbackslash n})\). Two potential errors take the form \((a, \text{\_}; b) = \text{span } f \text{ xs}\). At first glance this pattern definition is similar to the one in isers. But here the pattern is only safe if one of the elements in the list $xs$, f returns True. The test f is actually \((\not= \text{\textbackslash n})\), and the only safe condition Catch can express is that $xs$ is an infinite list. With the amendment \((a, b) = \text{safeSpan } f \text{ xs}\), where safeSpan is defined by:

```haskell
\text{safeSpan } p \text{ xs} = (a, \text{drop } 1 \text{ b}) \text{ where } (a, b) = \text{span } p \text{ 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::[a b] → 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 E 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:

\[ e = \left(\star \cdot \star \right) + 2 \cdot \text{tail} \odot \text{concat} \odot \text{map} \odot \text{show} \odot \text{head} \]

\[ \text{iterate} \left(\text{carryPropagate} 2 \odot \text{map} \odot \left(10^+ \odot \text{tail}\right)\right) 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.

5.3 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:

\[ \text{delFromFM} \left(\text{Branch} \left(\ldots\right) \text{del_key} \mid \text{del_key} > \text{key} = \ldots\right) \mid \text{del_key} < \text{key} = \ldots \mid \text{del_key} \equiv \text{key} = \ldots \]

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 of delFromFM as defined 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.

5.4 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 and analysed it using Catch. HsColour has 12 modules, is 5 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

We can see that parseColourPrefs calls read, which in turn calls error. The read function is specified to crash on incorrect parses, so the blame probably lies in 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.

5.5 The XMonad Program

XMonad (Stewart and Sjanssen 2007) is a window manager, which automatically manages the layout of program windows on the screen. The central module of XMonad contains a pure API, which is used to manipulate a data structure containing information regarding window layout. Catch has been run on this central module.

\[ \text{http://www.cs.york.ac.uk/~ftp/darcs/hscolour/} \]
several times, as XMonad has evolved. The XMonad API contains 36 exported functions, most of which are intended to be total. Within the implementation of these functions, there are a number of incomplete patterns and calls to partial functions.

When the Catch tool was first used, it detected six issues which were cause for concern – including unsafe uses of partial functions, API functions which contained incomplete pattern matches, and unnecessary assumptions about the Ord class. All these issues were subsequently fixed. The XMonad developers have said: “QuickCheck and Catch can be used to provide mechanical support for developing a clean, orthogonal API for a complex system” (Stewart and Sjøssem 2007). In e-mail correspondence, the XMonad developers have summarised their experience using Catch as follows: “[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.”

6. Related Work

6.1 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.

The Catch tool tries to prove that error calls are unreachable. The Reach tool (Naylor and Runciman 2007) also checks for reachability, trying to find values which will cause a certain expression to be evaluated. Unlike Catch, if the Reach tool cannot find a way to reach an expression, this is no guarantee that the expression is indeed unreachable. So the tools are complementary: Reach can be used to find examples causing non-exhaustive patterns to fail, Catch can be used to prove there are no such examples.

6.2 Proving Incomplete Patterns Safe

Despite the seriousness of the problem of pattern matching, there are very few other tools for checking pattern-matching 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 Catch08 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.

The closest other work we are aware of is ESC/Haskell (Xu 2006) and its successor Sound Haskell (Xu et al. 2007). The Sound 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. We eagerly await prototypes of either tool, to permit a full comparison against Catch.

6.3 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

\[
\begin{align*}
data \text{ Cons} \\
data \text{ Unknown} \\
\text{newtype List } \alpha \tau = \text{List } [\alpha] \\
\text{cons :: } \alpha \to [\alpha] \to \text{List } \alpha \text{ Cons} \\
\text{cons } \alpha \alpha \tau = \text{List } (\alpha : \alpha) \\
\text{nil :: List } \alpha \text{ Unknown} \\
\text{nil = List } [] \\
\text{fromList :: } [\alpha] \to \text{List } \alpha \text{ Unknown} \\
\text{fromList } \alpha \alpha \tau = \text{List } \alpha \\
\text{safeTail :: List } \alpha \text{ Cons } \to [\alpha] \\
\text{safeTail } \alpha \text{ Cons } \tau = \alpha \\
\end{align*}
\]

Figure 15. A safeTail function with Phantom types.

GHC compiler (The GHC Team 2007) 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 2007). 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.

6.4 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 nil 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 15. The List type is not exported, ensuring that all lists with a Cons tag are indeed non-empty. The types Cons 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 16. 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.

7. Conclusions and Future Work

We have described the design, implementation and application of Catch, an analysis tool for safe pattern-matching in Haskell 98.
data ConsT α
data NilT

data List α τ where
  Cons :: α → List α τ → List α (ConsT τ)
  Nil :: List α NilT

safeTail :: List α (ConsT τ) → List α τ
safeTail (Cons a b) = b

fromList :: [α] → (τ ★ List α τ → β) → β
fromList [] fn = fn Nil
fromList (x : xs) fn = fromList xs (fn ◦ Cons x)

Figure 16. A safeTail function using GADTs.

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) confirms the 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 expressive 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.

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) 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 expressive 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.

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) 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 expressive 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.

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) 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 expressive 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.