Not All Patterns, But Enough

an automatic verifier for partial but sufficient pattern matching

Neil Mitchell *
University of York, UK
ndmitchell@gmail.com

Colin Runciman
University of York, UK
colin@cs.york.ac.uk

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 pattern-match failures in darcs functions.

Consider the following example taken from Mitchell and Runciman (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 Hackage[1]. We have also given an argument for the soundness of our method in (Mitchell 2008), showing that if Catch declares a program free from pattern-match errors then that program is guaranteed not to crash with a pattern-match error.

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 calculates *preconditions* of functions.
- Two separate constraint languages that can be used with our method.
- Details of the Catch implementation which supports the full Haskell 98 language (Peyton Jones 2003), by transforming Haskell 98 programs to a first-order language.

* The first author is supported by an EPSRC PhD studentship

Haskell'08, September 25, 2008, Victoria, BC, Canada.

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

```
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.

- 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.

This paper has similar aims to our previous work (Mitchell and Runciman 2007), which we will refer to as Catch05[2]. For Catch05 risers is towards the limit of what is possible, for Catch08 it is trivial. Catch05 can only deal with small examples that use a few restricted forms of recursion. Catch08 uses substantially improved algorithms, along with radically different constraint mechanisms, to deal with real Haskell programs.

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 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⩽(:)

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

x⩽(:) ⇒ (risers x)⩽(:)
True ⇒ (risers2 x y z)⩽(:)

The first line 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.

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

```
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.

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 that if no part of any argument is ⊥ then no part of the result is ⊥. For example, the precondition on tail xs is that xs is (:)-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 (:)-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.

```
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)

hnf :: Expr → Maybe (CtorName, [Expr])
hnf (Make c  xs  ) = Just (c, xs)
hnf (Call  f xs  )
  | f ≡ "error"  = Nothing
  | otherwise    = hnf (subst (zip (args f) xs) (body f))
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
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 r′ e) | Alt n vs e ← xs
  , let r′ = filter ((∉ vs) ∘ fst) r]
```

Figure 3. Semantics for Core expressions.

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

Figure 4. Operations on Core.

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. 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 the 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 a case-expression whose scrutinee is e. The functions body and args obtain the body and argument names of a function. The isRec (c, i) function returns true if the constructor c has a recursive ith component; for example, let hd = (":", 0) and tl = (":", 1) then isRec hd = False but isRec tl = True.

3.1.2 Transformation From Haskell to Core

To generate core representations of programs, we start with Yhc, the York Haskell Compiler (Golubovsky et al. 2007). Yhc can transform a Haskell program into a single Yhc Core file, containing all necessary libraries. Yhc Core is higher-order and has let expressions, neither of which are permitted in our Core language. The let expressions can be removed using simple transformations provided by the Yhc Core library. The standard way to make a program

```
data Prop α

(∧), (∨)   :: 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.

first-order is Reynolds style defunctionalisation (Reynolds 1972). This method embeds a mini-interpreter into the resultant program, which would complicate our analysis method considerably. Instead, we use an alternative defunctionalisation method (Mitchell 2008, Chapter 5), which removes most higher-order functions without complicating the analysis. If any higher-order functions remain we then use Reynolds method. 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
```

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 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

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 $\bot$, the constraint is automatically satisfied: in our method, for a component of x to evaluate to $\bot$, 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 Catch05. 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 ($\triangleright$) and ($\triangleleft$) are discussed in §3.5. The expression x⩽cs generates a proposition ensuring that the value x must be constructed by one of the constructors in cs.

The type signatures for the functions calculating preconditions and entailments are given in Figure 7. The precond function (see §3.4) takes a function name, and gives a proposition imposing

```
data Sat α = Sat α Constraint

(⩽) :: α → [CtorName] → Prop (Sat α)
(▷) :: Selector → Constraint → Constraint
(◁) :: CtorName → Constraint → Prop (Sat Int)
```

Figure 6. Constraint operations.

```
precond :: FuncName → Prop (Sat VarName)
prePost :: FuncName → Constraint → Prop (Sat VarName)
reduce  :: Prop (Sat Expr) → Prop (Sat VarName)

substP :: Eq α ⇒ [(α, β)] → Prop (Sat α) → Prop (Sat β)
substP xs = mapP (λ(Sat i k) → lit $ Sat (f i) k)
  where f i = fromJust $ lookup i xs
```

Figure 7. Operations to generate preconditions and entailments.

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

Figure 8. Basic pattern constraints.

constraints on the arguments to that function. The prePost function (see §3.5) takes a function name and a postcondition, and gives a precondition sufficient to ensure the postcondition. During the manipulation of constraints, we often need to talk about constraints on expressions, rather than argument variables: the reduce function (see §3.5) converts propositions of constraints on expressions to equivalent propositions of constraints on arguments. The substP function performs substitution, by argument name or position, over propositions of constraints.

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:

```
a⩽xs = orP [lit (a `Sat` anys x) | x ← xs]
  where anys x = Con x (replicate (arity x) Any)
```

So, for example:

```
e⩽["True"]     = lit (e `Sat` Con "True" [])
e⩽[":"]        = lit (e `Sat` Con ":" [Any, Any])
e⩽[":", "[]"] = lit (e `Sat` Con ":" [Any, Any]) ∨
                lit (e `Sat` Con "[]" [])
```

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

```
pre :: Expr → Prop (Sat Expr)
pre (Var  v      ) = true
pre (Make c xs   ) = andP (map pre xs)
pre (Call  f xs  ) = pre′ f xs ∧ andP (map pre xs)
  where pre′ f xs = substP (zip (args f) xs) (precond f)
pre (Case  on alts) = pre on ∧ andP (map alt alts)
  where alt (Alt c vs e) = on⩽(ctors c \ [c]) ∨ pre e
```

Figure 9. Precondition of an expression, pre.

```
precond :: FuncName → Prop (Sat VarName)
precond0   f = if f ≡ "error" then false else true
precondn+1 f = precondn f ∧
               reduce (pre{precondn}(body f))
```

Figure 10. Precondition calculation.

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.

Example 1

```
safeTail xs = case null xs of
                True  → []
                False → tail xs
```

The precondition for safeTail, after removing conjunctions with true, is computed as:

```
pre′ null [xs] ∧ (null xs⩽["True"] ∨ pre′ tail [xs])
```

This predicate states that the invocation of null xs must be safe, and either null xs is True or tail xs must be safe. □

3.4.1 Stable Preconditions

The iterative algorithm for calculating preconditions is given in Figure 10. Initially all preconditions are assumed to be true, apart from the error precondition, which is false. In each iteration we calculate the precondition using the pre function from Figure 9, using the previous value of precond. Each successive precondition is conjoined with the previous one, and is therefore more restrictive. So *if all chains of increasingly restrictive propositions of constraints are finite*, termination is guaranteed – a topic we return to in §3.7.

We can improve the efficiency of the algorithm by tracking dependencies between preconditions, and performing the minimum amount of recalculation. Finding strongly connected components in the static call graph of a program allows parts of the program to be checked separately.

3.4.2 Preconditions and Laziness

The pre function defined in Figure 9 does not take laziness into account. 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.

```
reduce :: Prop (Sat Expr) → Prop (Sat VarName)
reduce = mapP (λ(Sat x k) → red x k)

red :: Expr → Constraint → Prop (Sat VarName)
red (Var    v       ) k = case var v of
                            Nothing     → lit (v `Sat` k)
                            Just (on, s) → red on (s ▷ k)
red (Make c   xs    ) k = reduce $ substP (zip [0 ..] xs) (c ◁ k)
red (Call   f   xs  ) k = reduce $ substP (zip (args f) xs)
                                          (prePost f k)
red (Case  on alts) k = andP [alt c e | Alt c vs e ← alts]
  where alt c e = reduce (on⩽(ctors c \ [c])) ∨ red e k
```

Figure 11. Specification of constraint reduction, reduce.

3.5 Manipulating constraints

The pre function generates constraints in terms of expressions, which the precond function transforms into constraints on function arguments, using reduce. The reduce function is defined in Figure 11. We will first give an example of how reduce works, followed by a description of each rule corresponding to an equation in the definition of red.

Example 1 (revisited)

The precondition for the safeTail function is:

$$\mathsf{pre'\ null}\ [\mathsf{xs}]\ \wedge\ (\mathsf{null\ xs} \leqslant [\texttt{"True"}]\ \vee\ \mathsf{pre'\ tail}\ [\mathsf{xs}])$$

We can use the preconditions computed for null and tail to rewrite the precondition as:

$$\mathsf{null\ xs} \leqslant [\texttt{"True"}]\ \vee\ \mathsf{xs} \leqslant [\texttt{":"}]$$

The reduce function changes constraints on expressions to constraints on function arguments. It makes use of an entailment to turn the constraint on null's result into a constraint on its argument:

$$\mathsf{xs} \leqslant [\texttt{"[]"}]\ \vee\ \mathsf{xs} \leqslant [\texttt{":"}]$$

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 of the case expression in which they occur. The ▷ operator lifts a constraint on one part of a data structure to a constraint on the entire data structure. For BP-constraints, ▷ can be defined as:

```
(c, i) ▷ k = Con c  [ if i ≡ j then k else Any
                    | j ← [0 .. arity c − 1]]
```

Example 2

```
case xs of
    []     → []
    y : ys → tail y
```

Here the initial precondition will be y⩽[":"], which evaluates to the result y `Sat` Con ":" [Any, Any]. The var function on y gives Just (xs, (":", 0)). After the application of ▷ the revised constraint refers to xs instead of y, and will be xs `Sat` Con ":" [Con ":" [Any, Any], Any]. We have gone from a constraint on y, using the knowledge that y is bound to a portion of xs, to a constraint on xs. □

```
prePost :: FuncName → Constraint → Prop (Sat VarName)
prePost_0   f k = true
prePost_n+1 f k = prePost_n f k ∧
                  reduce{ prePost_n }(lit $ body f `Sat` k)
```

Figure 12. Fixed point calculation for prePost.

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 field of a constructor.

```
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. For example, Sat 0 k requires that the first field satisfies k. Some sample applications:

```
"True"  ◁ Con "True" [] = true
"False" ◁ Con "True" [] = false
":" ◁ Con ":" [Con "True" [], Any] =
  lit (0 `Sat` Con "True" []) ∧ lit (1 `Sat` Any)
```

The Case **rule** generates a conjunct for each alternative. An alternative satisfies a constraint if *either* it is never taken, *or* it meets the constraint when taken.

The Call **rule** relies on the prePost function defined in Figure 12. This function calculates the precondition necessary to ensure a given postcondition on a function, which forms an entailment. Like the precondition calculation in §3.4, the prePost function works iteratively, with each result becoming increasingly restrictive. Initially, all postconditions are assumed to be true. The iterative step takes the body of the function, and uses the reduce transformation to obtain a predicate in terms of the arguments to the function, using the previous value of prePost. 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, and additionally caching all calculated results.

3.6 Semantics of Constraints

The semantics of a constraint are determined by which values satisfy it. We can implement a satisfies function using the ◁ operator:

```
satisfies :: Value → Constraint → Bool
satisfies Bottom      k = True
satisfies (Value c xs) k =
  satisfiesP $ substP (zip [0 ..] xs) (c ◁ k)

satisfiesP :: Prop (Sat Value) → Bool
satisfiesP x = (true :: Prop ()) ≡
               mapP (λ(Sat v k) → bool $ satisfies v k) x
```

The first equation returns True given a value of type Bottom, as if a value contains ⊥ then any constraint is true. In order to be consistent with ◁, the other operations must respect certain properties, here expressed as boolean-valued functions that should always return True.

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

The propExtend property requires that if a constraint satisfies a value after they have both been extended, then the original

value must have satisfied the original constraint. For example, if Just α `Sat` (("Just", 0) $\triangleright$ k) is true, then α `Sat` k must be true.

```
propOneOf v@(Value c xs) cs
  | c ∉ cs = not (satisfiesP (v⩽cs))
propOneOf _ _ = True
```

The propOneOf property requires that v⩽cs must not match values constructed by constructors not in cs. Note that both properties allow for constraints to be more restrictive than necessary.

In (Mitchell 2008) we give a detailed argument that if a constraint language satisfies these two properties, the algorithms presented in Figures 9 to 12 are sound. We also show that both BP-constraints and MP-constraints (introduced in §4.2) satisfy these properties.

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 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. Figure 13 gives an implementation of Catch08 regular expression based constraints (RE-constraints). 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 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 such 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 13, the differentiate function is from Conway (1971); integrate is its inverse; ewp is the empty word property.

Example 3

(head xs) is safe if xs evaluates to a non-empty list. The RE-constraint generated by Catch is: xs `Sat` (1 $\rightsquigarrow$ {:}). 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: xs `Sat` (tl*·hd $\rightsquigarrow$ {:}). 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 there are no well defined paths containing a hd selector. If xs is infinite then all its infinitely many elements must be (:)-constructed. □

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

(⩽) :: α → [CtorName] → Prop (Sat α)
e⩽cs = lit $ e `Sat` ([] ⇝ cs)

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

(◁) :: CtorName → Constraint → Prop (Sat Int)
c ◁ (r ⇝ cs) = bool (not (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 `Sat` (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 13. RE-constraints.

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: xs `Sat` (tl*·hd $\rightsquigarrow$ {:}) ∨ xs `Sat` (tl* $\rightsquigarrow$ {:}). 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. □

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
$r_1 \cdot r_2$	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

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 RexExp 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 $0 \rightsquigarrow \mathsf{c}$ always evaluates to True, $\lhd$ replaces Nothing by True.)

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. For example, three of the nineteen rules are:

Exhaustion: In the constraint x `Sat` $(\mathsf{r} \rightsquigarrow$ [":", "[]"]) 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` $(\mathsf{r} \rightsquigarrow \mathsf{c}_1) \wedge$ e `Sat` $(\mathsf{r} \rightsquigarrow \mathsf{c}_2)$ can be replaced by e `Sat` $(\mathsf{r} \rightsquigarrow (\mathsf{c}_1 \cap \mathsf{c}_2))$.

Or merging: The disjunction e `Sat` $(\mathsf{r} \rightsquigarrow \mathsf{c}_1) \vee$ e `Sat` $(\mathsf{r} \rightsquigarrow \mathsf{c}_2)$ can be replaced by e `Sat` $(\mathsf{r} \rightsquigarrow \mathsf{c}_2)$ if $\mathsf{c}_1 \subseteq \mathsf{c}_2$.

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.

```
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))

(⋞) :: α → [CtorName] → Prop (Sat α)
e⋞cs = lit $ Sat e  [ map complete cs
                    ⋆ map complete (ctors (head cs))
                    | not (null cs)]

(▷) :: Selector → Constraint → Constraint
(c, i) ▷ k = 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 (Sat 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 Sat non (map (:[]) vs) ++
                 map ( `Sat` [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 14. MP-constraints.

MP-constraints are similar to BP-constraints, but can constrain an infinite number of items. A value v satisfies a constraint $\mathsf{p}_1 \star \mathsf{p}_2$ if v itself satisfies the pattern p_1 and *all its recursive components at any depth* satisfy p_2. We call p_1 the root pattern, and p_2 the recursive pattern. Each of p_1 and p_2 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 p_2. A constraint is a disjunctive list of $\star$ patterns.

The intuition behind the definition of $(\mathsf{c}, \mathsf{i}) \rhd \mathsf{ps}$ is that if the selector (c, i) is recursive, given a pattern $\alpha \star \beta$, 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:

$$\mathsf{hd} \rhd (\alpha \star \beta) = \{(:)\ (\alpha \star \beta)\} \star \{[\,], (:)\ \mathsf{Any}\}$$
$$\mathsf{tl} \rhd (\alpha \star \beta) = \{(:)\ \mathsf{Any}\ \} \star (\mathsf{merge}\ \alpha\ \beta)$$

For the $\lhd$ operator, if the root pattern matches, then all non-recursive fields are matched to their non-recursive constraints, and all recursive fields have their root and recursive patterns become their recursive pattern. In the result, each field is denoted by its argument position. So, for example:

$$\texttt{":"} \lhd (\{[\,]\ \} \star \beta) = \mathsf{false}$$
$$\texttt{":"} \lhd (\{(:)\ \alpha\} \star \beta) = 0\ \texttt{`Sat`}\ \alpha\ \wedge\ 1\ \texttt{`Sat`}\ (\beta \star \beta)$$

Example 3 (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 can be read in two portions: the part to the left of $\star$ requires the value to be (:)-constructed, with an unrestricted hd field; the right allows either a [] or a (:) with an unrestricted hd field, and a tl field restricted by the constraint on the right of the $\star$. In this particular case, the right of the $\star$ 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:

$$\{[\,], (:)\ (\{(:)\ \mathsf{Any}\} \star \{[\,], (:)\ \mathsf{Any}\})\} \star$$
$$\{[\,], (:)\ (\{(:)\ \mathsf{Any}\} \star \{[\,], (:)\ \mathsf{Any}\})\}$$ □

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: $\{(:)\ \mathsf{Any}\} \star \{(:)\ \mathsf{Any}\}$ □

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 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
  where 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 atomic constraint:

$$(\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_1)\ \vee\ (\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_2) = \mathsf{Sat}\ \mathsf{e}\ (\mathsf{v}_1 \dashv\vdash \mathsf{v}_2)$$
$$(\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_1)\ \wedge\ (\mathsf{Sat}\ \mathsf{e}\ \mathsf{v}_2) = \mathsf{Sat}\ \mathsf{e}\ [\mathsf{a} \sqcap \mathsf{b} \mid \mathsf{a} \leftarrow \mathsf{v}_1, \mathsf{b} \leftarrow \mathsf{v}_2]$$

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

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

in uncurried form becomes:

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

□

Combining MP-constraint reduction rules with the uncurrying transformation makes Sat α equivalent in power to Prop (Sat α). 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 $\mathsf{v}_1 = (\mathsf{T}{:}[\,])$ and $\mathsf{v}_2 = (\mathsf{T}{:}\mathsf{T}{:}[\,])$ 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 values. □

Example 8

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

$$\mathsf{v}_1 = \mathsf{Branch}\ (\mathsf{Leaf}\ \mathsf{True}\ )\ (\mathsf{Leaf}\ \mathsf{False})$$
$$\mathsf{v}_2 = \mathsf{Branch}\ (\mathsf{Leaf}\ \mathsf{False})\ (\mathsf{Leaf}\ \mathsf{True}\ )$$

The RE-constraint $(\mathsf{left}^* \cdot \mathsf{leaf} \rightsquigarrow \mathsf{True})$ is satisfied by v_1 but not v_2. No MP-constraint separates the two values. □

We have implemented both constraint systems in Catch. 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 discusses 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 Nofib suite (Partain et al. 2008). To illustrate results for larger and widely-used applications, §5.3 investigates the FiniteMap library, §5.4 investigates the HsColour program and §5.5 reports on XMonad. In all cases our defunctionalisation method successfully removes all higher-order functions.

5.1 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 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 using MP-constraints are given in Table 1. Using RE-constraints, only 8 programs can be proven safe within 10 minutes, even after the modifications described later in this section. 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.

Table 1. Table of results

Name is the name of the checked program (a starred name indicates that changes were made before safe pattern-matching could be verified); **Src** is the number of lines in the original source code; **Core** is the number of lines of first-order Core, *including all needed Prelude and library definitions*, 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	Err	Pre	Sec	Mb
Bernoulli*	35	652	5	11	4.1	0.8
Digits of E1*	44	377	3	8	0.3	0.6
Digits of E2	54	455	5	19	0.5	0.8
Exp3-8	29	163	0	0	0.1	0.1
Gen-Regexps*	41	776	1	1	0.3	0.4
Integrate	39	364	3	3	0.3	1.9
Paraffins*	91	1153	2	2	0.8	1.9
Primes	16	241	6	13	0.2	0.1
Queens	16	283	0	0	0.2	0.2
Rfib	9	100	0	0	0.1	1.7
Tak	12	155	0	0	0.1	0.1
Wheel Sieve 1*	37	570	7	10	7.5	0.9
Wheel Sieve 2*	45	636	2	2	0.3	0.6
X2n1	10	331	2	5	1.8	1.9
FiniteMap*	670	1829	13	17	1.6	1.0
HsColour*	823	5060	4	9	2.1	2.7

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

```
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:

```
main = do args ← getArgs
          case map reads args of
               [[(x, s)]] | 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 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. 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 ∘ lines, which can be replaced by takeWhile ($\not\equiv$ '\n'). Two potential errors take the form $(a, \_ : b) = \text{span f xs}$. 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 $(a, b) = \text{safeSpan f xs}$, where safeSpan is defined by:

$$\text{safeSpan p xs} = (a, \text{drop } 1\ b) \textbf{ where } (a, b) = \text{span p xs}$$

Catch verifies pattern safety.

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

$$\textbf{data } \text{Wheel} = \text{Wheel Int } [\text{Int}]$$

$$\text{sieve} :: [\text{Wheel}] \to [\text{Int}] \to [\text{Int}] \to [\text{Int}]$$

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:

$$\text{radical\_generator n} = \text{radicals undefined}$$
$$\textbf{where } \text{radicals unused} = \text{big\_memory\_computation}$$

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 $\text{array} :: \text{Ix a} \Rightarrow (a, a) \to [(a, b)] \to \text{Array a b}$ 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:

$$\text{carryPropagate base } (d : ds) = \ldots$$
$$\textbf{where } \text{carryguess} = d \text{ `div` base}$$
$$\text{remainder} = d \text{ `mod` base}$$
$$\text{nextcarry} : \text{fraction} = \text{carryPropagate } (\text{base} + 1)\ ds$$

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 fails. Now consider:

$$\begin{aligned} e = {} & (\text{"2."} +\!\!+) \$ \\ & \text{tail} \circ \text{concat} \$ \\ & \text{map } (\text{show} \circ \text{head}) \$ \\ & \text{iterate } (\text{carryPropagate } 2 \circ \text{map } (10*) \circ \text{tail}) \$ \\ & 2 : [1, 1 \ldots] \end{aligned}$$

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:

$$\begin{aligned} \text{delFromFM } (\text{Branch key } \ldots)\ \text{del\_key} \mid {} & \text{del\_key} > \text{key} = \ldots \\ \mid {} & \text{del\_key} < \text{key} = \ldots \\ \mid {} & \text{del\_key} \equiv \text{key} = \ldots \end{aligned}$$

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 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.

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[3] 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 of the checking process 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).

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

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, 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 Sjanssen 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-match safety. This paper has similar goals to Mitchell and Runciman (2007), but many 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 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 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 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.

```
data Cons
data Unknown
newtype List α τ = List [α]

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

nil :: List α Unknown
nil = List []

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

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

Figure 15. A safeTail function with Phantom types.

```
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 []       f = f Nil
fromList (x : xs) f = fromList xs (f ∘ Cons x)
```

Figure 16. A safeTail function using GADTs.

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. 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 a mini-interpreter. 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 make minor modifications to programs, 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.

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.

Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core – from Haskell to Core. *The Monad.Reader*, 1(7):45–61, April 2007.

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

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):1–35, May 2007.

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

Neil Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, 2008.

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.

Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *Proc. SCAM '07*, pages 133–142. IEEE Computer Society, September 2007.

Will Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2008.

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. In *Proc. ICFP '06*, pages 50–61. ACM Press, 2006.

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

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.

Don Stewart and Spencer Sjanssen. XMonad. In *Proc. Haskell '07*, pages 119–119. ACM Press, 2007.

The GHC Team. The GHC compiler, version 6.8.2. `http://www.haskell.org/ghc/`, December 2007.

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. ACM Press, 2001.

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.

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.

Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *Proc. IFL 2007*, pages 382–399, 2007.