Unfailing Haskell: A Static Checker for Pattern Matching

Neil Mitchell and Colin Runciman
http://www.cs.york.ac.uk/∼ndm , http://www.cs.york.ac.uk/∼colin

University of York, UK

Abstract

A Haskell program may fail at runtime with a pattern-match error if the program has any incomplete (non-exhaustive) patterns in definitions or case alternatives. This paper describes a static checker that allows non-exhaustive patterns to exist, yet ensures that a pattern-match error does not occur. It describes a constraint language that can be used to reason about pattern matches, along with mechanisms to propagate these constraints between program components.

1 INTRODUCTION

Programming in Haskell is closer to a mathematical ideal than programming in imperative languages such as C. But many functions in Haskell are partial, not total. Turner [9] argues convincingly that what is needed is a functional programming language in which every program executes to completion, without raising an error. In the discipline of total functional programming that Turner proposes it is impossible to write either programs that generate case errors or that do not terminate. Another way of describing this is that $\perp$ is removed from the language. The question of non-exhaustive patterns is dealt with using the rule that all patterns must be exhaustive. Turner argues that this system will "force you to pay attention to exactly those corner cases which are likely to cause trouble" [9].

But this approach requires sacrifices – one notable casualty is the `head` function. Since `head` can raise an error if the argument is an empty list, the standard definition cannot be used in Turner's total programming language. It would be nice to obtain some of the benefits of a total programming language, namely its unfailing nature, without losing the natural definition of `head`. This paper contributes the design of a pattern-match checker for Haskell.

1.1 Motivating Example

In order to show what the checking tool gives us, consider the following example:

```
risers :: Ord a => [a] -> [[a]]
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 execution of this function would be:

```
> risers [1,2,3,1,2]
[[1,2,3],[1,2]]
```

In the last line of the definition, `(s:ss)` is matched against the output of `risers`. If `risers (y:etc)` returns an empty list this would cause a pattern match error. It takes a few moments to check this program manually – and a few more to be sure one has not made a mistake! Turning the program over to the checker developed in this paper, the output is:

```
> (risers (y:etc)){:}
> True
```

The checker first decides that for the code to be safe the recursive call to `risers` must always yield a non-empty list. It then notices that if the argument in a `riser` application is non-empty, then so will the result be. This satisfies it, and it returns True, guaranteeing that no pattern-match errors will occur.

2 REDUCED HASKELL

The full Haskell language is a bit unwieldy for analysis. In particular the syntactic sugar complicates analysis by introducing more types of expression to consider. The checker works instead on a simplified language, a core to which other Haskell programs can be reduced. This core language is a functional language, making use of case expressions, function applications and algebraic data types.

In this reduced language types and arities are not explicit. Also, there are no named data variables – all variables are referred to by their relationship to an argument. For example, `@1` refers to the first argument of the function in whose body it appears, and $\texttt{@1.Cons}_2$ refers to the second component of the `Cons`-constructed first argument. $\texttt{Cons}_2$ is called a *selector*.

Example 1

```
data List = Cons | Nil

head = case @1 of Cons -> @1.Cons1

map = case @2 of
    Nil -> Nil
    Cons -> Cons (@1 @2.Cons1) (map @1 @2.Cons2)

reverse = rev @1 Nil

rev = case @1 of
    Nil -> @2
    Cons -> rev @1.Cons2 (Cons @1.Cons1 @2)
```

♢

E ::= `arg` m (written `@`m) | `sel` E C m (written $E.C_m$)
| `make` C $E_1 \cdots E_n$ | `func` f
| `apply` E_0 $E_1 \cdots E_n$ | `case` E_0 `of` $\{C_1$ `->` E_1`;` $\cdots$ `;` C_n `->` $E_n\}$
f is the name of a function, C is the name of a constructor, m is a positive integer

FIGURE 1. Abstract Syntax of expressions in reduced Haskell

From now on $\texttt{Cons}_1$ and $\texttt{Cons}_2$ will be written as `head` and `tail` respectively, this is purely to aid understanding for the human reader.

2.1 Values

The values in this reduced language consist only of algebraic data types, and functions. A value is either a function, or a constructor and a list of component values. The type of a value can be deduced statically – whether it is a function or an algebraic value, and in the second case, what are its possible constructors.

2.2 Expressions

Expressions in reduced Haskell are defined in Figure 1. A convertor from a subset of Haskell to this reduced language has been written, and all examples from here onwards use this convertor.

2.3 Higher Order

The current checker is not fully higher order, but some higher-order programs can be checked successfully.

The checker tries to eliminate higher-order functions by specialization. A function can be specialized in its *n*th argument if in all recursive calls this argument is invariant. (There are slight additional complications, due to mutual recursion).

Examples of common functions whose applications can be specialized in this way include `map`, `filter`, `foldr` and `foldl`.

When a function can be specialized, the expression passed as the *n*th argument has all its free variables passed as extra arguments, and is expanded in the specialized version. All recursive calls within the new function are then renamed

Example 2

```
map f xs = case xs of
        [] -> []
        a:b -> f a : map f b

adds x n = map (add n) x
```

is transformed into:

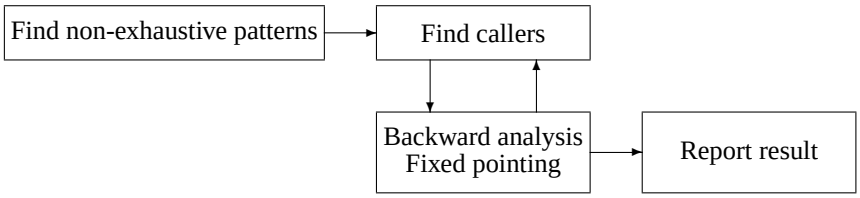

FIGURE 2. Checker Overview

```
map_adds n xs = case xs of
        [] -> []
        a:b -> add n a : map_adds n b

adds x n = map_adds n x
```
♢

Although this firstification approach is not complete by any means, it appears to be sufficient for a large range of examples. Alternative methods are available for full firstification, such as that detailed by Hughes [4].

3 OVERVIEW

The checking process has two main ingredients, a constraint language for expressing properties on data structures and some mechanisms for generating and manipulating constraints to reflect the definition of functions. Both are introduced in detail later on, but first a sketch overview of the checking process is given. A diagram of the process is given in Figure 2

At each stage, the information passed "along the arrows" is a predicate, where the atoms are constraints as introduced in §4. These constraints can either refer to any reduced Haskell expression, or in a special case can refer only to parameters to functions.

The initial stage of finding all non-exhaustive case expressions is done with a basic syntactic check, at a very local level. Initial constraints are generated from these expressions.

Finding all callers is relatively straightforward if all constraints are only on arguments to functions. The result of this stage are constraints on expressions.

Backward analysis and fixed pointing convert back from constraints on expressions into constraints on arguments. Backward analysis performs the translation, but without regard to recursive function calls. Fixed pointing modifies constraints to reflect the recursive calls.

This process continues, until the predicate is reduced to either True or False. If the end result is True, then the system is free from pattern-match errors. If it is False, then the system *may* give rise to pattern errors. The checker is conservative.

In practice False is always accompanied by a history of derivations, ending in

False. These derivations allow the user to gain insight into a possible cause of failure.

4 A CONSTRAINT LANGUAGE

In order to implement a checker that can ensure unfailing patterns, it is useful to have some way of expressing classes of data values. A constraint is written as $\alpha.r\{c\}$, where α is an expression, r is a regular expression over selectors and c is a set of constructors. Such a constraint asserts that any well-defined application to α of a path of selectors described by r must reach a constructor in the set c.

These constraints are used as atoms in a predicate language with conjunction and disjunction, so constraints can be about several expressions and relations between them. The checker does not require a negation operator. We also use the term constraint to refer to logical formulae with constraints as atoms.

Example 3

Consider the function `minimum`, defined as:

```
minimum [x] = x
minimum (a:b:xs) = minimum ((if a < b then a else b) : xs)
```

Now consider the expression `minimum` α. The constraint that must hold for this expression to be safe is $\alpha\{$`:`$\}$. This says that the expression α must reduce to an application of `:`, i.e. a non-empty list. ♢

Example 4

Consider the expression `map minimum` α. In this case the constraint generated is α`.*tail.head`$\{$`:`$\}$. The part following the α is a regular expression, with the `*` operator being applied prefix. If we apply any number (possibly zero) of `tail`s to α, then apply `head`, we reach a `:`. Values satisfying this constraint include `[]` and `[[1],[2],[3]]`, but not `[[1],[]]`. ♢

Constraints divide up into three parts – the *subject*, the *path* and the *condition*. These are usually written as $\alpha.r\{c\}$, but for certain equations writing them as $\langle\alpha, r, c\rangle$ is easier.

The subject in the above two examples was just α, representing any expression – including a call, a construction or even a `case`.

The path is a regular expression over selectors.

A regular expression is defined as:

$s+t$	union of regular expressions s and t
$s.t$	concatenation of regular expressions s then t
$*s$	any number (possibly zero) occurrences of s
C_n	a constructor C and an integer n, being a selector
λ	the language is the set containing the empty string
ϕ	the language is the empty set

The condition is a set of constructors which, due to static type checking, must all be of the same type.

So the first example, $\alpha\{:\}$, could have been written more fully as $\alpha.\lambda\{:\}$ – where λ is the regular expression which describes the language consisting only of the empty string.

The meaning of a constraint is defined by:

$$\alpha.r\{c\} \Leftrightarrow (\forall l \in L(r) \bullet \text{exists}(\alpha, l) \Rightarrow \text{constructor}(\alpha.l) \in c)$$

$$\begin{array}{ll} exists(\_, \Lambda) & = True \\ exists(Ca_1 \ldots a_n, C'_i.\omega) & = C \equiv C' \wedge exists(a_i, \omega) \end{array}$$

Here $L(r)$ is the language represented by the regular expression r; exists returns true if a value has a given path; and constructor gives the constructor used to create the data. Of course, since $L(r)$ is potentially infinite, this cannot be checked by enumeration.

If there are no expressions which can be found following any instance of the path, then the constraint is vacuously true.

4.1 Simplifying the Constraints

From the formal definition of the constraints it is possible to construct a number of identities which can be used for simplification.

Path does not exist: in the constraint `[].head{:}` the expression `[]` does not have a `head` path, so this constraint simplifies to true.

Detecting failure: the constraint `[]{:}` simplifies to false because the `[]` value is not the constructor `:`.

Empty path: in the constraint $\alpha.\phi\{c\}$, the regular expression is ϕ, the empty language, so the constraint is always true.

Exhaustive conditions: in the constraint $\alpha\{:, []\}$ the condition lists all the possible constructors, and because of static typing α must be one of these, therefore this constraint simplifies to true.

Algebraic conditions: finally a few algebraic equivalences:

$$\begin{array}{rcl} \alpha.r_1\{c\} \vee \alpha.r_2\{c\} & = & \alpha.(r_1 + r_2)\{c\} \\ \alpha.r\{c_1\} \vee \alpha.r\{c_2\} & = & \alpha.r\{c_1 \cup c_2\} \\ \alpha.r\{c_1\} \wedge \alpha.r\{c_2\} & = & \alpha.r\{c_1 \cap c_2\} \end{array}$$

5 DETERMINING THE CONSTRAINTS

This section concerns the derivation of the constraints, and the operations involved in this task. An overview of the stages presented here, and how they relate to each other, is given in §3.

5.1 The Initial Constraints

In general, a `case` expression:

`case` α `of` $\texttt{Sel}_1$ `->` $\texttt{val}_1$`;` `...;` $\texttt{Sel}_n$ `->` $\texttt{val}_n$

produces the initial constraint $\alpha\{\texttt{Sel}_1, \ldots, \texttt{Sel}_n\}$. If the case alternatives are exhaustive, then this can be simplified to true. All `case` expressions in the program are found, their initial constraints are found, and these are joined together with conjunction.

5.2 Transforming the constraints

For each constraint in turn, if the subject is `f@n`, the checker searches for every function call of `f`, and gets the expression corresponding to its *n*th argument. On this expression, it sets the existing constraint. This argument is then transformed using a backward analysis (see §5.3), until a constraint on arguments is found.

Example 5

Given the constraint `minimum@1`$\{$`:`$\}$, if the program contains the expression:

```
f = minimum (g @1)
```

then the derived constraint is `(g f@1)`$\{$`:`$\}$. That is, the expression passed as `minimum`'s first argument must evaluate to a non-empty list. $\diamondsuit$

5.3 Backward Analysis

Backward analysis is the process which takes a constraint in which the subject is a compound expression, and changes it to a combination of constraints over arguments only. This process is denoted by a function $\varphi(\alpha, r, c)$ where α is the expression, r is the path and c is the condition. This function is detailed in Figure 3. In order to denote the evaluation of an expression into a value, there is a relation $\mathcal{D}$, which is not defined in this paper.

$$\varphi(\texttt{arg}\ \ n,r,c) \to \langle \text{qual}(n),r,c\rangle$$

$$\frac{\varphi(E,r,c) \to \langle E',r',c'\rangle}{\varphi(\texttt{sel}\ \ E\ \ C\ \ m,r,c) \to \langle E',C_m.r',c'\rangle}$$

$$\frac{\varphi(E_1,\frac{\partial r}{\partial C_1},c) \to E_1',\cdots,\varphi(E_n,\frac{\partial r}{\partial C_n},c) \to E_n'}{\varphi(\texttt{make}\ \ C\ \ E_1\cdots E_n) \to (\lambda \in L(r) \Rightarrow C \in c) \wedge E_1' \wedge \cdots \wedge E_n'}$$

$$\frac{\begin{array}{c}\varphi(\mathcal{D}(E_0),r,c) \to P \\ P[\langle \texttt{arg}\ \ 1,r_1,c_1\rangle/\varphi(E_1,r_1,c_1),\cdots,\langle \texttt{arg}\ \ n,r_n,c_n\rangle/\varphi(E_n,r_n,c_n)] \to P'\end{array}}{\varphi(\texttt{apply}\ \ E_0\ \ E_1\cdots E_n,r,c) \to P'}$$

$$\frac{\begin{array}{c}C = \{x|\text{type}(x) = \text{type}(C_1)\} \\ P = (\varphi(E,\lambda,C\backslash C_1) \vee \varphi(E_1,r,c)) \wedge \cdots \wedge (\varphi(E,\lambda,C\backslash C_n) \vee \varphi(E_n,r,c))\end{array}}{\varphi(\texttt{case}\ \ E\ \ \texttt{of}\ \ \{C_1\texttt{->}E_1\texttt{;}\cdots\texttt{;}C_n\texttt{->}E_n\},r,c) \to P}$$

FIGURE 3. Specification of backward analysis, φ

The `arg` rule qualifies the argument before putting it in the condition. In every expression, all the `arg` references can be qualified with the name of the function they appear in. For example, in the body of the function `f`, `arg` *n* is qualified to `f@`*n*.

The `sel` rule says that if a constraint is satisfied on the expression used before a selector, then following this selector obtains the new constraint.

The `make` rule says that the condition must be true on the constructor used in the `make` expression if λ is in the language represented by the regular expression. This corresponds precisely to the *empty word property* [3], which can be calculated structurally on the regular expression. For each of the arguments to the data structure, it must be true that the condition holds when the derivative of the regular expression with respect to that constructor and argument position is taken. This is denoted by the $\partial r/\partial C_i$. The differentiation method is based on that described in [3].

The `apply` rule uses the result of backward analysis applied to the function to find preconditions on the arguments. While this is fine in theory, it is not necessarily terminating – in fact the naive application of this rule to any function with a recursive call will loop forever. To combat this, if a function

is already in the process of being evaluated with the same constraint, its result is given as true, and the recursive arguments are put into a special pile to be examined later on.

The `case` rule generates a conjunct for each alternative. The generated condition says either the subject of the case analysis has a different constructor (so this particular alternative is not executed in this circumstance), or the right hand side of the alternative is safe given the conditions for this expression. So if the checker can prove a given alternative in a case is not taken in this situation, it can ignore that alternative.

5.4 Obtaining a Fixed Point

We have noted that if a function is in the process of being evaluated, and its value is asked for again with the same constraints, then the call is deferred. After backwards analysis has been performed on the result of a function, there will be a constraint in terms of the arguments, along with a set of recursive calls. If these recursive calls had been analyzed further, then the checking computation would not have terminated.

Example 6

```
mapHead xs = case xs of
        [] -> []
        a:b -> head a : mapHead b
```

The function `mapHead` is exactly equivalent to `map head`. Running the checker over this function, the constraint generated is `mapHead@1.head{:}`, and the only recursive call noted is `mapHead @1.tail`. Observe that the constraint only mentions the first element in the list, while the desired constraint would mention them all. In effect `mapHead` has been analyzed without considering any recursive applications.

Having obtained this constraint and recursive call, the checker attempts to find a fixed point. It does this by noting that the first argument in the recursive call is `@1.tail`. The notation used for this is `@1` $\hookleftarrow$ `@1.tail`. What predicate would have to be satisfied if n recursive calls to the function were performed? Denoting this predicate as P_n, where P_0 is the initial constraint:

$$P_{n+1} = P_n \wedge P_n[\texttt{@1}/\texttt{@1.tail}]$$

So for the `mapHead` function:

$P_0 =$ `@1{:}`
$P_1 =$ `@1{:}` $\wedge$ `@1.tail{:}`
$P_2 =$ `@1{:}` $\wedge$ `@1.tail{:}` $\wedge$ `@1.tail.tail{:}`

The checker attempts to find a fixed point P_∞ such that

$$P_n = P_{n+1} \Rightarrow P_\infty = P_n$$

but in this example there is no fixed point. If a fixed point cannot be established, the system has special rules for dealing with a limited set of common circumstances.

For `mapHead` we have `@1` $\hookleftarrow$ `@1.tail`, so `@1`$_\infty$ = `@1.*tail`. With this knowledge the constraint can be written by replacing `@1` with `@1`$_\infty$. We then obtain the desired constraint, that `mapHead@1.*tail.head{:}`. ♢

In general if an expression exists of the form `@i` $\hookleftarrow$ `@i.path` then `@i`$_\infty$ = `@i.*(path)`. A special case is where `path` is λ. In this case `@i`$_\infty$ = `@i`.

While these special-case rules handle many directly recursive functions, they do not work for all.

Example 7

Consider the function `reverse` written using an accumulator:

```
reverse x = reverse2 x []

reverse2 x y = case x of
        a:b -> reverse2 b (a:y)
        [] -> y
```

Argument `@1` follows the pattern `@1` $\hookleftarrow$ `@1.tail`, but we also have `@2` $\hookleftarrow$ `(@1.head : @2)`. If the program being analyzed contained `main x = map head (reverse x)`, the part of the condition that applies to `reverse2@2` before the fixed pointing is `reverse2@2.*tail.head{:}`.

In this case a second rule for obtaining a fixed point is needed. This second rule handles recursive calls of the form

`@i` $\hookleftarrow$ `C` $x_1 \cdots x_n$ `@i`

(Where the positions of `@i` and `x` within `C` can be reordered, with $R(x)$ giving the position of any variable.) The constraint must be $\langle$`@i`$, r, c\rangle$, with $\partial r / \partial \mathtt{C}_{R(\mathtt{@i})} = r$. In this case, P_∞ is defined to be:

$$(\lambda \in L(r) \Rightarrow \mathtt{C} \in c) \wedge \bigwedge_{i=1}^{n} \langle \mathtt{x}_i, \mathtt{C}_{R(\mathtt{x}_i)} \cdot \frac{\partial r}{\partial \mathtt{C}_{R(\mathtt{x}_i)}}, c \rangle$$

In the `reverse` example the final condition is, as expected:

`reverse2@1.*tail.head{:}` $\wedge$ `reverse2@2.*tail.head{:}` ♢

6 SOME SMALL EXAMPLES AND A CASE STUDY

Example 8

```
head x = case x of
           a:b -> a
main x = head x
> head@1{:}
> False[main@1{:}]
```

This example requires only initial constraint generation, and a simple propagation. ♢

Example 9

```
main x = map head x
> head@1{:}
> map_head@1.*tail.head{:}
> False[main@1.*tail.head{:}]
```

This example shows specialization generating a new function `map_head`, fixed pointing being applied to `map`, and the constraints being propagated through the system. ♢

Example 10

```
main x = map head (reverse x)
-- reverse x is defined with an accumulator
> head@1{:}
> map_head@1.*tail.head{:}
> False[main@1.*tail{:} ∨ main@1.*tail.head{:}]
```

This result may at first seem surprising. The first disjunct of the constraint says that applying `tail` any number of times to `main@1` (also known as `x`) the result must always be a `:`, in other words `x` must be infinite. This guarantees case safety because `reverse` is tail strict, so if its argument is an infinite list, no result will ever be produced, and a case error will not occur. The second disjunct says, less surprisingly, that the list before it is reversed must be a list in which every element is a non-empty list. ♢

Example 11

```
main x = tails x
tails x = foldr tails2 [[]] x
tails2 x y = (x:head y) : y
> head@1{:}
> tails2@2{:}
> fold_tails2@2.*tail.tail{:} ∨ fold_tails2@1{:}
> True
```

This final example uses a fold to calculate the `tails` function. But as the auxiliary `tails2` makes use of `head` – it is not (at first glance) free from pattern-match errors. The first two lines of the output are simply moving the constraint around. The third line is the interesting one. In this line the checker gives two alternative conditions for case safety – either the first argument is a `:`, or the list is either zero length or it is infinite. The way the requirement for zero or infinite length is encoded is by the path `*tail.tail`. If the list is of zero length, then there are no tails, and no words in the regular expression language match. If however, there is one tail, then that tail, and all successive tails must be `:`. So either `foldr` does not call its function argument because it immediately takes the zero case, or `foldr` recurses infinitely, and therefore the function is never called. Either way, because the initial argument to `foldr` is a `:`, and because `tails2` always returns a `:`, the second part of the condition can be satisfied. ♢

6.1 The Clausify Program

Our goal is to check standard Haskell programs, and to provide useful feedback to the user. To test the checker against those objectives we have used several Haskell programs, all written some time ago for other purposes. The analysis of one program is discussed below.

The Clausify program has been around for a very long time, since at least 1990. It has made its way into the `nofib` benchmark suite [5], and was the focus of several papers on heap profiling [6]. It parses logical propositions and puts them in clausal form. We ignore the parser and jump straight to the transformation of propositions. The data structure for a formula is:

```
data F =
    Sym Char | Not F | Dis F F | Con F F | Imp F F | Eqv F F
```

and the main pipeline is:

```
clauses =
    concat . map disp . unicl . split . disin . negin . elim
```

Each of these stages takes a proposition and returns an equivalent version – for example the `elim` stage replaces implications with disjunctions and negation. Each stage eliminates certain forms of proposition, so that future stages do not have to consider them. Despite most of the stages being designed to deal with a restricted class of propositions, the only function which contains a non-exhaustive pattern match is in the definition of `clause` (a helper function for `unicl`).

```
clause p = clause' p ([] , [])
  where
    clause' (Dis p q)       x   = clause' p (clause' q x)
    clause' (Sym s)       (c,a) = (insert s c , a)
    clause' (Not (Sym s)) (c,a) = (c , insert s a)
```

After encountering the non-exhaustive pattern match, the checker generates the following constraints, using $C_?$ as an abbreviation for C_1+C_2:

```
> clause'@1.*Dis?{Dis,Sym,Not} ∧ clause'@1.*Dis?.Not1{Sym}
> clause@1.*Dis?{Dis,Sym,Not} ∧ clause@1.*Dis?.Not1{Sym}
> unicl'@1.*Dis?{Dis,Sym,Not} ∧ unicl'@1.*Dis?.Not1{Sym}
> foldr_unicl@2.*tail.head.*Dis?{Dis,Sym,Not} ∧
  foldr_unicl@2.*tail.head.*Dis?.Not1{Sym}
> unicl@1.*tail.head.*Dis?{Dis,Sym,Not} ∧
  unicl@1.*tail.head.*Dis?.Not1{Sym}
```

These constraints give accurate and precise requirements for a case error not to occur at each stage, and are very useful. However, when the condition is propagated back over the `split` function, the result becomes less pleasing. An error occurs in fixed pointing, because no fixed pointing scheme matches the available function. The original definition of `split`:

```
split p = split' p []
    where
        split' (Con p q) a = split' p (split' q a)
        split' p a = p : a
```

can be transformed manually by the removal of the accumulator:

```
split (Con p q) = split p ++ split q
split p = [p]
```

This second version is accepted by the checker, which generates the constraint:

```
>
(
    split@1.*Con?{Con,Dis,Sym,Not} ∧
    split@1.*Con?.Dis?.*Dis?{Dis,Sym,Not} ∧
    split@1.*Con?.*Dis?.Not1{Sym}
)
```

This constraint can be read as follows: the outer structure of a propositional argument to `split` is any number of nested `Con` constructors; the next level is any number of nested `Dis` constructors; at the innermost level there must be either a `Sym`, or a `Not` containing a `Sym`. That is, propositions are in *conjunctive normal form*.

The one part of this constraint that may be unexpected is the $\texttt{Dis}_?\texttt{.*Dis}_?$ part of the path in the 2nd conjunct. We might rather expect something similar to $\texttt{*Con}_?\texttt{.*Dis}_?\texttt{\{Dis,Sym,Not\}}$, but consider what this means. Take as an example the value `(Con Sym Sym)`. This value meets all 3 conjunctions generated by the tool, but does not meet this new constraint: the path has the empty word property, so the root of the value can no longer be a `Con` constructor.

The next function encountered is `disin` which shifts disjunction inside conjunction. The version in the nofib benchmark has following equation in its definition:

```
disin (Dis p q) =
    if conjunct dp || conjunct dq
    then disin (Dis dp dq)
    else (Dis dp dq)
        where
            dp = disin p
            dq = disin q
```

Unfortunately, when expanded out this gives the call

```
disin (Dis (disin p) (disin q))
```

which does not have a fixed point under the present scheme. Refactoring is required to enable this stage to succeed. Fortunately, in [6] a new version of `disin` is given, which is vastly more efficient than this one, and (as a happy side effect) is also accepted by the checker.

At this point in the story a crisis occurs. Although a constraint is calculated for the new `disin`, this constraint is approximately 15 printed pages long! Initial exploration suggests that there are missed opportunities to simplify regular expressions.

7 RELATED WORK

7.1 Checking for exhaustiveness and usefulness

One way of alerting users to possible spurious pattern matches is by checking for exhaustiveness and usefulness.

Example 12

```
notUseful (x:xs) = xs
notUseful [x]    = []
notUseful []     = []

notExhaustive (Just x:xs) = [x]
notExhaustive []          = []
```

The first function has a redundant second equation – if the argument is of the form `[x]`, then it would have already matched the first equation. Equations defining the second function are not exhaustive: for example, if the argument is `[Nothing]` an error occurs. ♢

When trying to compile these examples using GHC [7] 6.4, the first provokes a warning, but the second does not. There is a compile time flag which can be added and catches both, named `-fwarn-incomplete-patterns`. However, the Bugs (12.2.1) section 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" [7].

The unfortunate problem with these checks is that they are highly local. If the function `head` is defined, then it raises a warning. No effort is made to check the *callers* of `head` – this is an obligation left to the programmer.

7.2 Type Analysis for XML

There are similar problems involving XML [1] and XSLT [2]. XML is a hierarchical data structure, which can be though of as an algebraic data structure. XSLT is a transformation language, with rules given to apply to various XML values. In XSLT there is no destructive assignment, recursion is supported, a form of pattern matching is used – overall it can be seen as a functional language.

A type specification of an XML document is written in a DTD (Document Type Definition), and can express types such as a node of type `html` contains a `head` followed by a `body`. The paper [8] tackles a subset of XSLT named XSLT0. The question the paper attempts to address is: Given a DTD for an output document, and an XSLT0 transformation, what is the DTD for the input document?

The advantage of this knowledge is that a document can be checked to meet an output DTD without the cost of transformation first, and the errors can be determined in the input (or source) document, which the user wrote – not a document generated by a transformation.

The paper treats this as a question of backward type inference. A type is synthesized as a finite tree automaton, and is deduced compositionally. Correctness proofs are presented, along with an efficient algorithm for inference.

8 CONCLUSIONS AND FURTHER WORK

A static checker for potential pattern-match errors in Haskell has been specified and implemented. This checker is capable of determining the preconditions under which a program with non-exhaustive patterns executes without failing due to a pattern-match error. A range of small examples has been investigated successfully, along with some larger programs. Where programs cannot be checked initially, refactoring can increase the checker's success rate.

The checker relies on specialization to remove higher order functions. Where higher order functions do remain, provided they do not have any pattern match failures, the remaining part of the program can be checked.

The checker is fully polymorphic but it does not currently handle classes; we hope these can be transformed away without vast complication.

The checker is a prototype only, and various enhancements could be made.

- The next challenge is to translate from full Haskell into the reduced language. This work has been started: we have a converter for a useful subset.
- The checker should output fuller traces that can be manually verified. Currently the predicate at each stage is given, without any record of how it was obtained, or what effect fixedpointing had.

- The central algorithms of the checker can be refined. A better fixed pointing procedure could perhaps make additional use of backward analysis.

With these improvements we hope to check larger Haskell programs, and to give useful feedback to the programmer.

ACKNOWLEDGEMENT

The first author is a PhD student supported by a studentship from the Engineering and Physical Sciences Research Council of the UK.

REFERENCES

[1] T. Bray. Extensible Markup Language (XML) 1.0 (Third Edition). `http://www.w3.org/TR/2004/REC-xml-20040204/`, Feb. 2004.

[2] J. Clark. XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`, Nov. 1999.

[3] J. H. Conway. *Regular Algebra and Finite Machines*. London Chapman and Hall, 1971.

[4] J. Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, pages 183–215. Springer LNCS 1110, Feb. 1996.

[5] W. Partain. The `nofib` Benchmark Suite of Haskell Programs. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*, pages 195–202. Springer-Verlag Workshops in Computing, 1992.

[6] C. Runciman and D. Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, 1993.

[7] The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4. `http://www.haskell/org/ghc/docs/latest/html/users_guide`, Mar. 2005.

[8] A. Tozawa. Towards Static Type Checking for XSLT. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering*, pages 18–27, New York, NY, USA, 2001. ACM Press.

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