Chapter 2

A Static Checker for Safe Pattern Matching in Haskell

Neil Mitchell and Colin Runciman [2.1]

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.

2.1 INTRODUCTION

Often it is useful to define pattern matches which are incomplete, for example `head` fails on the empty list. Unfortunately programs with incomplete pattern matches may fail at runtime.

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

[2.1] University of York, UK. http://www.cs.york.ac.uk/∼ndm and http://www.cs.york.ac.uk/∼colin

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!

GHC [The05] 6.4 has a warning flag to detect incomplete patterns, which is named `-fwarn-incomplete-patterns`. Adding this flag at compile time reports:[2.2]

```
Warning: Pattern match(es) are non-exhaustive
```

But the GHC checks are only 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.

Turning the `risers` function 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 `risers` 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.

2.1.1 Roadmap

This paper starts by introducing a reduced language similar to Haskell in §2.2. Next a constraint language is introduced in §2.3 and algorithms are given to manipulate these constraints in §2.4. A worked example is given in §2.5, followed by a range of small examples and a case study in §2.6. This paper is compared to related work in §2.7. Finally conclusions are given in §2.8, along with some remaining tasks – this paper reports on work in progress.

2.2 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.

As shown in example 1, only one defining equation per function is permitted, pattern-matching occurs only in case expressions and every element within a constructor must be uniquely named by a selector (e.g. `hd` and `tl`). A convertor from a reasonable subset of Haskell to this reduced language has been written.

[2.2] The additional flag `-fwarn-simple-patterns` is needed, but this is due to GHC bug number 1075259

Example 2.1

```
data [] a = (:) {hd :: a, tl :: [] a} | []

head x = case x of (a:_) -> a

map f xs = case xs of
               []     -> []
               (a:as) -> f x : map f as

reverse xs = rev xs []

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

□

2.2.1 Higher Order Functions

The current checker is not higher order, and does not allow partial application.

The checker tries to eliminate higher-order functions by specialization. A mutually recursive group of functions can be specialized in their *n*th argument if in all recursive calls this argument is invariant.

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 2.2

```
map f xs = case xs of
               []     -> []
               (a:as) -> f a : map f as

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

is transformed into:

```
map_adds n xs = case xs of
                    []     -> []
                    (a:as) -> add n a : map_adds n as

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 [Hug96], or the defunctionalisation approach by Reynolds [Rey72].

2.2.2 Internal Representation

While the concrete syntax allows the introduction of new variable names, the internal representation does not. All variables are referred to using a *selector path* from an argument to the function.

For example, the internal representation of `map` is:

```
map f xs = case xs of
               []    -> []
               (_:_) -> f (xs·hd) : map f (xs·tl)
```

(Note that the infix · operator here is used to compose paths; it is *not* the Haskell function composition operator.)

2.3 A CONSTRAINT LANGUAGE

In order to implement a checker that can ensure unfailing patterns, it is useful to have some way of expressing properties of data values. A constraint is written as $\langle e, r, c \rangle$, where e 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 e 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 2.3

Consider the function `minimum`, defined as:

```
minimum xs = case xs of
                 [x]      -> x
                 (a:b:xs) -> minimum (min a b : xs)

min a b = case a < b of
              True  -> a
              False -> b
```

Now consider the expression `minimum` e. The constraint that must hold for this expression to be safe is $\langle e, \lambda, \{\texttt{:}\} \rangle$. This says that the expression e must reduce to an application of `:`, i.e. a non-empty list. In this example the path was λ – the empty path. □

Example 2.4

Consider the expression `map minimum` e. In this case the constraint generated is $\langle e, \texttt{tl}^*\cdot\texttt{hd}, \{\texttt{:}\} \rangle$. If we apply any number (possibly zero) of `tl`s to e,

then apply `hd`, we reach a `:` construction. Values satisfying this constraint include `[]` and `[[1],[2],[3]]`, but not `[[1],[]]`. The value `[]` satisfies this constraint because it is impossible to apply either `tl` or `hd`, and therefore the constraint does not assert anything about the possible constructors.

□

Constraints divide up into three parts – the *subject*, the *path* and the *condition*.

The subject in the above two examples was just *e*, 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
$s \cdot t$	concatenation of regular expressions s then t
s^*	any number (possibly zero) occurrences of s
x	a selector, such as `hd` or `tl`
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 result type.

The meaning of a constraint is defined by:

$$\langle e, r, c \rangle \Leftrightarrow (\forall l \in L(r) \bullet \mathit{defined}(e,l) \Rightarrow \mathit{constructor}(e \cdot l) \in c)$$

Here $L(r)$ is the language represented by the regular expression r; *defined* returns true if a path selection is well-defined; 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 no path selection is well-defined then the constraint is vacuously true.

2.3.1 Simplifying the Constraints

From the 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 $\langle$`[]`,`hd`,$\{$`:`$\}\rangle$ the expression `[]` does not have a `hd` path, so this constraint simplifies to true.

Detecting failure: the constraint $\langle$`[]`,λ,$\{$`:`$\}\rangle$ simplifies to false because the `[]` value is not the constructor `:`.

Empty path: in the constraint $\langle e, \phi, c \rangle$, the regular expression is ϕ, the empty language, so the constraint is always true.

Exhaustive conditions: in the constraint $\langle e, \lambda, \{\texttt{:}, \texttt{[]}\}\rangle$ the condition lists all the possible constructors, if e reaches weak head normal form then because of static typing e must be one of these constructors, therefore this constraint simplifies to true.

Algebraic conditions: finally a couple of algebraic equivalences:

$$\begin{array}{rcl} \langle e, r_1, c\rangle \wedge \langle e, r_2, c\rangle & = & \langle e, (r_1 + r_2), c\rangle \\ \langle e, r, c_1\rangle \wedge \langle e, r, c_2\rangle & = & \langle e, r, c_1 \cap c_2\rangle \end{array}$$

2.4 DETERMINING THE CONSTRAINTS

This section concerns the derivation of the constraints, and the operations involved in this task.

2.4.1 The Initial Constraints

In general, a `case` expression, where $\overrightarrow{v}$ are the arguments to a constructor:

`case` e `of` $\texttt{C}_1\ \overrightarrow{v}\ \texttt{->}\ \texttt{val}_1\texttt{;}\ \ldots\texttt{;}\ \texttt{C}_n\ \overrightarrow{v}\ \texttt{->}\ \texttt{val}_n$

produces the initial constraint $\langle e, \lambda, \{\texttt{C}_1\texttt{,}\ldots\texttt{,}\texttt{C}_n\}\rangle$. 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.

2.4.2 Transforming the constraints

For each constraint in turn, if the subject is $\texttt{x}_\texttt{f}$ (i.e. the `x` argument to `f`), the checker searches for every application of `f`, and gets the expression for the argument `x`. On this expression, it sets the existing constraint. This constraint is then transformed using a backward analysis (see §2.4.3), until a constraint on arguments is found.

Example 2.5

Consider the constraint $\langle \texttt{xs}_\texttt{minimum}, \lambda, \{\texttt{:}\}\rangle$ – that is `minimum`'s argument `xs` must be a non-empty list. If the program contains the expression:

```
f x = minimum (g x)
```

then the derived constraint is $\langle \texttt{(g}\ \texttt{x}_\texttt{f}\texttt{)}, \lambda, \{\texttt{:}\}\rangle$. □

2.4.3 Backward Analysis

Backward analysis takes a constraint in which the subject is a compound expression, and derives a combination of constraints over arguments only. This process

$$\varphi\langle e \cdot s, r, c\rangle \to \varphi\langle e, s \cdot r, c\rangle \tag{sel}$$

$$\frac{\bigwedge_{i=1}^{\#\overrightarrow{e}} \varphi\langle e_i, \frac{\partial r}{\partial \mathcal{S}(C,i)}, c\rangle \to P}{\varphi\langle C\ \overrightarrow{e}, r, c\rangle \to (\lambda \in L(r) \Rightarrow C \in c) \wedge P} \tag{con}$$

$$\varphi\langle f\ \overrightarrow{e}, r, c\rangle \to \varphi\langle \mathcal{D}(f, \overrightarrow{e}), r, c\rangle \tag{app}$$

$$\frac{\bigwedge_{i=1}^{\#\overrightarrow{e}} (\varphi\langle e_0, \lambda, \mathcal{C}(C_i)\rangle \vee \varphi\langle e_i, r, c\rangle) \to P}{\varphi\langle \texttt{case}\ e_0\ \texttt{of}\ \{C_1\ \overrightarrow{v}\texttt{->}e_1\texttt{;}\cdots\texttt{;}C_n\ \overrightarrow{v}\texttt{->}e_n\}, r, c\rangle \to P} \tag{cas}$$

FIGURE 2.1. Specification of backward analysis, φ

is denoted by a function φ, which takes a constraint and returns a predicate over constraints. This function is detailed in Figure 2.1.

In this figure, C denotes a constructor, c is a set of constructors, f is a function, e is an expression, r is a regular expression over selectors and s is a selector.

The (sel) rule moves the composition from the expression to the path.

The (con) rule deals with an application of a constructor C. If λ is in the path language the C must be permitted by the condition. This depends on the *empty word property* (ewp) [Con71], which can be calculated structurally on the regular expression.

For each of the arguments to C, a new constraint is obtained from the derivative of the regular expression with respect to that argument's selector. This is denoted by $\partial r/\partial \mathcal{S}(C,i)$, where $\mathcal{S}(C,i)$ gives the selector for the ith argument of the constructor C. The differentiation method is based on that described by Conway [Con71]. It can be used to test for membership in the following way:

$$\begin{array}{rcl} \lambda \in L(r) & = & \mathrm{ewp}(r) \\ s \cdot r' \in L(r) & = & r' \in L(\partial r/\partial s) \end{array}$$

Two particular cases of note are $\partial\lambda/\partial a = \phi$ and $\partial\phi/\partial a = \phi$.

The (app) rule uses the notation $\mathcal{D}(f, \overrightarrow{e})$ to express the result of substituting each of the arguments in $\overrightarrow{e}$ into the body of the function f. 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, see §2.4.4 for details.

The (cas) rule generates a conjunct for each alternative. The function $\mathcal{C}(C)$ returns the set of all other constructors with the same result type as C, i.e.

$\mathcal{C}(\texttt{[]}) = \{\texttt{:}\}$. 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.

2.4.4 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 2.6

```
mapHead xs = case xs of
                 []     -> []
                 (x:xs) -> head x : mapHead xs
```

The function `mapHead` is exactly equivalent to `map head`. Running backward analysis over this function, the constraint generated is $\langle \texttt{xs}_{\texttt{mapHead}}, \texttt{hd}, \{\texttt{:}\}\rangle$, and the only recursive call noted is `mapHead (xs·tl)`. The recursive call is written as `xs` $\hookleftarrow$ `xs·tl`, showing how the value of `xs` changes. Observe that the path in the constraint only reaches the first element in the list, while the desired constraint would reach them all. In effect `mapHead` has been analyzed without considering any recursive applications.

The fixed point for this function can be derived by repeatedly replacing `xs` with `xs·tl` in the subject of the constraint, and joining these constraints with conjunction.

$$\begin{aligned}
& \langle \texttt{xs}, \texttt{hd}, \{\texttt{:}\}\rangle \wedge \langle \texttt{xs·tl}, \texttt{hd}, \{\texttt{:}\}\rangle \wedge \langle \texttt{xs·tl·tl}, \texttt{hd}, \{\texttt{:}\}\rangle \wedge \ldots && (1)\\
\equiv\ & \langle \texttt{xs}, \texttt{hd}, \{\texttt{:}\}\rangle \wedge \langle \texttt{xs}, \texttt{tl·hd}, \{\texttt{:}\}\rangle \wedge \langle \texttt{xs}, \texttt{tl·tl·hd}, \{\texttt{:}\}\rangle \wedge \ldots && (2)\\
\equiv\ & \langle \texttt{xs}, \texttt{hd+tl·hd+tl·tl·hd+}\ldots, \{\texttt{:}\}\rangle && (3)\\
\equiv\ & \langle \texttt{xs}, \texttt{(}\lambda\texttt{+tl+tl·tl+}\ldots\texttt{)·hd}, \{\texttt{:}\}\rangle && (4)\\
\equiv\ & \langle \texttt{xs}, \texttt{tl}^{*}\texttt{·hd}, \{\texttt{:}\}\rangle && (5)
\end{aligned}$$

The justification is as follows. First use the backwards analysis rule given in Figure 2.1 to transform between (1) and (2) – selectors move from the subject to the path. To obtain (3) the first algebraic condition given in §2.3.1 is used. The factorisation of the `hd` element of the regular expression is applied. Finally this can be rewritten using the regular expression *operator as the result. □

More generally, given any constraint of the form $\langle \texttt{x}, r, c\rangle$ and a recursive call of the form `x` $\hookleftarrow$ `x.`p, the fixed point is $\langle \text{x}, p^{*} \cdot r, c\rangle$. A special case is where p is λ, in which case $p^{*}\cdot r = r$.

Example 2.7

Consider the function `reverse` written using an accumulator:

```
reverse x = reverse2 x []

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

Argument `xs` follows the pattern $\mathtt{x} \hookleftarrow \mathtt{x.tl}$, but we also have the recursive call $\mathtt{a} \hookleftarrow \mathtt{(x{\cdot}hd:a)}$. If the program being analyzed contained an instance of `map head (reverse x)`, the part of the condition that applies to `a` before the fixed pointing of `a` is $\langle \mathtt{a}, \mathtt{tl}^*{\cdot}\mathtt{hd}, \{\mathtt{:}\}\rangle$.

In this case a second rule for obtaining a fixed point can be used. For recursive calls of the form $\mathtt{a} \hookleftarrow \mathtt{C}\ \mathtt{x}_1\ \cdots\ \mathtt{x}_n\ \mathtt{a}$, where s is the selector corresponding to the position of a, the rule is:

$$\bigwedge_{r' \in r^{\#}} \left(\left(\lambda \in L(r') \Rightarrow C \in c\right) \wedge \langle a, r', c\rangle \wedge \bigwedge_{i=1}^{n} \langle x_i, \frac{\partial r'}{\partial \mathcal{S}(C,i)}, c\rangle \right)$$

Where:

$$r^{\#} = \{r^0, r^1, \ldots, r^{\infty}\} \qquad r^0 = r \qquad r^{(n+1)} = \frac{\partial r^n}{\partial s}$$

It can be shown that $r^{\#}$ is always a finite set [Law04]. This expression is derived from the (con) rule §2.4.3, applied until it reaches a fixed point.

In the `reverse` example, $r^{\#}$ is $\{\mathtt{tl}^*{\cdot}\mathtt{hd}\}$, since $\partial \mathtt{tl}^*{\cdot}\mathtt{hd}/\partial \mathtt{tl} = \mathtt{tl}^*{\cdot}\mathtt{hd}$. Also $\lambda \notin L(\mathtt{tl}^*{\cdot}\mathtt{hd})$, so the result is:

$$\begin{aligned} & \langle \mathtt{a}, \mathtt{tl}^*{\cdot}\mathtt{hd}, \{\mathtt{:}\}\rangle \wedge \langle \mathtt{x{\cdot}hd}, \frac{\partial \mathtt{tl}^*{\cdot}\mathtt{hd}}{\partial \mathtt{hd}}, \{\mathtt{:}\}\rangle \\ \equiv\ & \langle \mathtt{a}, \mathtt{tl}^*{\cdot}\mathtt{hd}, \{\mathtt{:}\}\rangle \wedge \langle \mathtt{x{\cdot}hd}, \lambda, \{\mathtt{:}\}\rangle \\ \equiv\ & \langle \mathtt{a}, \mathtt{tl}^*{\cdot}\mathtt{hd}, \{\mathtt{:}\}\rangle \wedge \langle \mathtt{x}, \mathtt{hd}, \{\mathtt{:}\}\rangle \end{aligned}$$

Next applying the fixed pointing due to `x`, gives a final condition, as expected: $\langle \mathtt{a}, \mathtt{tl}^*{\cdot}\mathtt{hd}, \{\mathtt{:}\}\rangle \wedge \langle \mathtt{x}, \mathtt{tl}^*{\cdot}\mathtt{hd}, \{\mathtt{:}\}\rangle$ □

While the two rules given do cover a wide range of examples, they are not complete. Additional rules exist for other forms of recursion but not all recursive functions can be handled using the current scheme.

Example 2.8

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

Here the recursive call is $\texttt{y} \hookleftarrow \texttt{x·tl}$, which does not have a rule defined for it. In such cases the checker conservatively outputs `False`, and also gives a warning message to the user. The checker always terminates.

The fixed point rules classify exactly which forms of recursion can be accepted by the checker. Defining more fixed point rules which can capture an increasingly large number of patterns is a matter for future work.

2.5 A WORKED EXAMPLE

Recall the `risers` example in §2.1. The first step of the checker is to transform this into reduced Haskell.

```
risers xs =
    case xs of
         []        -> []
         [x]       -> [[x]]
         (x:y:etc) -> risers2 (x <= y) x (risers (y:etc))

risers2 b x y = case y of
                    (s:ss) -> case b of
                                   True  -> (x:s) : ss
                                   False -> [x] : (s:ss)
```

The auxiliary `risers2` is necessary because reduced Haskell has no `where` clause. The checker proceeds as follows:

Step 1, Find all incomplete case statements. The checker finds one, in the body of `risers2`, the argument `y` must be a non-empty list. The constraint is $\langle \texttt{y}_{\texttt{risers2}}, \lambda, \{\texttt{:}\} \rangle$.

Step 2, Propagate. The auxiliary `risers2` is applied by `risers` with `risers (y:etc)` as the argument `s`. This gives $\langle \texttt{(risers (y:etc))}, \lambda, \{\texttt{:}\} \rangle$. When rewritten in terms of arguments and paths of selectors, this gives the constraint $\langle \texttt{(risers (xs}_{\texttt{risers}}\texttt{·tl·hd : xs}_{\texttt{risers}}\texttt{·tl·tl))}, \lambda, \{\texttt{:}\} \rangle$.

Step 3, Backward analysis. The constraint is transformed using the backward analysis rules. The first rule invoked is (app), which says that the body of `risers` must evaluate to a non-empty list, in effect an inline version of the constraint. Backward analysis is then performed over the case statement, the constructors, and finally `risers2`. The conclusion is that provided $\texttt{xs}_{\texttt{risers}}$ is a `:`, the result will be. The constraint is $\langle \texttt{(xs}_{\texttt{risers}}\texttt{·tl·hd : xs}_{\texttt{risers}}\texttt{·tl·tl)}, \lambda, \{\texttt{:}\} \rangle$, which is true.

In this example, there is no need to perform any fixed pointing.

2.6 SOME SMALL EXAMPLES AND A CASE STUDY

In the following examples, each line represents one propagation step in the checker. The final constraint is given on the last line.

```
head x = case x of
              (y:ys) -> y
main x = head x
```
> $\langle \mathtt{x}_{\mathtt{head}}, \lambda, \{\mathtt{:}\}\rangle$
> $\langle \mathtt{x}_{\mathtt{main}}, \lambda, \{\mathtt{:}\}\rangle$

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

Example 2.9

```
main x = map head x
```
> $\langle \mathtt{x}_{\mathtt{head}}, \lambda, \{\mathtt{:}\}\rangle$
> $\langle \mathtt{x}_{\mathtt{map\_head}}, \mathtt{tl}^*\cdot\mathtt{hd}, \{\mathtt{:}\}\rangle$
> $\langle \mathtt{x}_{\mathtt{main}}, \mathtt{tl}^*\cdot\mathtt{hd}, \{\mathtt{:}\}\rangle$

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 2.10

```
main x = map head (reverse x)
```
> $\langle \mathtt{x}_{\mathtt{head}}, \lambda, \{\mathtt{:}\}\rangle$
> $\langle \mathtt{x}_{\mathtt{map\_head}}, \mathtt{tl}^*\cdot\mathtt{hd}, \{\mathtt{:}\}\rangle$
> $\langle \mathtt{x}_{\mathtt{main}}, \mathtt{tl}^*, \{\mathtt{:}\}\rangle \vee \langle \mathtt{x}_{\mathtt{main}}, \mathtt{tl}^*\cdot\mathtt{hd}, \{\mathtt{:}\}\rangle$

This result may at first seem surprising. The first disjunct of the constraint says that applying `tl` any number of times to $\mathtt{x}_{\mathtt{main}}$ 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 every item in `x` must be a non-empty list. □

Example 2.11

```
main xs ys = case null xs || null ys of
                  True -> 0
                  False -> head xs + head ys
```
> $\langle \mathtt{x}_{\mathtt{head}}, \lambda, \{\mathtt{:}\}\rangle$
> $\langle (\mathtt{null\ xs}_{\mathtt{main}}\ \mathtt{||\ null\ ys}_{\mathtt{main}}), \lambda, \{\mathtt{True}\}\rangle \vee$
> $(\langle \mathtt{xs}_{\mathtt{main}}, \lambda, \{\mathtt{:}\}\rangle \wedge \langle \mathtt{ys}_{\mathtt{main}}, \lambda, \{\mathtt{:}\}\rangle)$
> $\langle \mathtt{xs}_{\mathtt{main}}, \lambda, \{\mathtt{[]}\}\rangle \vee \langle \mathtt{ys}_{\mathtt{main}}, \lambda, \{\mathtt{[]}\}\rangle \vee (\langle \mathtt{xs}_{\mathtt{main}}, \lambda, \{\mathtt{:}\}\rangle \wedge \langle \mathtt{ys}_{\mathtt{main}}, \lambda, \{\mathtt{:}\}\rangle)$
> `True`

This example shows the use of a more complex condition to guard a potentially unsafe application of `head`. The backward analysis applied to `null` and `||` gives precise requirements, which when expanded results in a tautology, showing that no pattern match error can occur. □

Example 2.12

```
main x = tails x
tails x = foldr tails2 [[]] x
tails2 x y = (x:head y) : y
> ⟨x_head, λ, {:}⟩
> ⟨y_tails2, λ, {:}⟩
> ⟨n1_foldr_tails2, λ, {:}⟩ ∨ ⟨n2_foldr_tails2, tl*·tl, {:}⟩
> True
```

This final example uses a fold to calculate the `tails` function. As the auxiliary `tails2` makes use of `head` the program is not obviously 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 the case safety of `foldr tails2` – either its first argument is a `:`, or its second argument is empty or infinite. The way the requirement for empty or infinite length is encoded is by the path $\texttt{tl}^*\cdot\texttt{tl}$. If the list is `[]`, then there are no tails to match the path. 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 `[]` case, or `foldr` recurses infinitely, and therefore the function is never called. Either way, because `foldr`'s second argument is a `:`, and because `tails2` always returns a `:`, the first part of the condition can be satisfied. □

2.6.1 The Clausify Program

Our goal is to check standard Haskell programs, and to provide useful feedback to the user. To test the checker against these 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 [Par92], and was the focus of several papers on heap profiling [RW93]. 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 :: Char} | Not {n :: F}
       | Dis {d1, d2 :: F} | Con {c1, c2 :: F}
       | Imp {i1, i2 :: F} | Eqv {e1, e2 :: F}
```

and the main pipeline is:

```
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:

$$
\begin{aligned}
&> \langle \mathtt{p}_{\mathtt{clause'}}, \mathtt{(d1+d2)^*}, \mathtt{\{Dis,Sym,Not\}} \rangle \wedge \langle \mathtt{p}_{\mathtt{clause'}}, \mathtt{(d1+d2)^*{\cdot}n}, \mathtt{\{Sym\}} \rangle \\
&> \langle \mathtt{p}_{\mathtt{clause'}}, \mathtt{(d1+d2)^*}, \mathtt{\{Dis,Sym,Not\}} \rangle \wedge \langle \mathtt{p}_{\mathtt{clause}}, \mathtt{(d1+d2)^*{\cdot}n}, \mathtt{\{Sym\}} \rangle \\
&> \langle \mathtt{p}_{\mathtt{unicl'}}, \mathtt{(d1+d2)^*}, \mathtt{\{Dis,Sym,Not\}} \rangle \wedge \langle \mathtt{p}_{\mathtt{unicl'}}, \mathtt{(d1+d2)^*{\cdot}n}, \mathtt{\{Sym\}} \rangle \\
&> \langle \mathtt{x}_{\mathtt{foldr\_unicl}}, \mathtt{tl^*{\cdot}hd{\cdot}(d1+d2)^*}, \mathtt{\{Dis,Sym,Not\}} \rangle \wedge \\
&\quad \langle \mathtt{x}_{\mathtt{foldr\_unicl}}, \mathtt{tl^*{\cdot}hd{\cdot}(d1+d2)^*{\cdot}n}, \mathtt{\{Sym\}} \rangle \\
&> \langle \mathtt{x}_{\mathtt{unicl}}, \mathtt{tl^*{\cdot}hd{\cdot}(d1+d2)^*}, \mathtt{\{Dis,Sym,Not\}} \rangle \wedge \\
&\quad \langle \mathtt{x}_{\mathtt{unicl}}, \mathtt{tl^*{\cdot}hd{\cdot}(d1+d2)^*{\cdot}n}, \mathtt{\{Sym\}} \rangle
\end{aligned}
$$

These constraints give accurate and precise requirements for a case error not to occur at each stage. However, when the condition is propagated back over the `split` function, the result becomes less pleasing. None of our fixed pointing schemes handle the original recursive 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:

$$
\begin{aligned}
&> \langle \mathtt{p}_{\mathtt{split}}, \mathtt{(c1+c2)^*}, \mathtt{\{Con,Dis,Sym,Not\}} \rangle \wedge \\
&\quad \langle \mathtt{p}_{\mathtt{split}}, \mathtt{(c1+c2)^*{\cdot}(d1+d2){\cdot}(d1+d2)^*}, \mathtt{\{Dis,Sym,Not\}} \rangle \wedge \\
&\quad \langle \mathtt{p}_{\mathtt{split}}, \mathtt{(c1+c2)^*{\cdot}(d1+d2)^*{\cdot}n}, \mathtt{\{Sym\}} \rangle
\end{aligned}
$$

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 surprising part of this constraint is the `(d1+d2)`·`(d1+d2)`* part of the path in the 2nd conjunct. We might rather expect something similar to `(c1+c2)`*·`(d1+d2)`*{`Dis,Sym,Not`}, but consider what this means. Take as an example the value `(Con (Sym 'x') (Sym 'y'))`. 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 the 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 [RW93] 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 the story comes to an end. Although a constraint is calculated for the new `disin`, this constraint is approximately 15 printed pages long! Initial exploration suggests at least one reason for such a large constraint: there are missed opportunities to simplify paths. We are confident that with further work the Clausify example can be completed.

2.7 RELATED WORK

Viewed as a **proof tool** this work can be seen as following Turner's goal to define a Haskell-like language which is total [Tur04]. Turner disallows incomplete pattern matches, saying this will "force you to pay attention to exactly those corner cases which are likely to cause trouble". Our checker may allow this restriction to be lifted, yet still retain a total programming language.

Viewed as a basic **pattern match checker**, the work on compiling warnings about incomplete and overlapping patterns is quite relevant [JHH$^+$93, Mar05]. As noted in the introduction, these checks are only local.

Viewed as a **mistake detector** this tool has a similar purpose to the classic C Lint tool [Joh78], or Dialyzer [LS04] – a static checker for Erlang. 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.

Viewed as a **soft type system** the checker can be compared to the tree automata work done on XML and XSL [Toz01], which can be seen as an algebraic data type and a functional language. Another soft typing system with similarities is by Aiken [AM91], on the functional language FL. This system tries to assign a type to each function using a set of constructors, for example `head` is given just `Cons` and not `Nil`.

2.8 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 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. Where programs cannot be checked initially, refactoring can increase the checker's success rate. Work in progress includes:

- The checker currently relies on specialization to remove higher order functions.
- The checker is fully polymorphic but it does not currently handle Haskell's type classes; we hope these can be transformed away without vast complication [Jon94].
- Another 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 offer 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 fixed pointing had. Although a more detailed trace would not help an end user, it would help strengthen the understanding of the algorithms.
- The central algorithms of the checker can be refined. In particular a better fixed pointing scheme is being developed. A complete analysis of which programs can be verified would be useful.
- A correctness proof is needed to prove that the checker is sound. This will require a semantics for the reduced Haskell-like language.

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

[AM91] Alex Aiken and Brian Murphy. Static Type Inference in a Dynamically Typed Language. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 279–290. ACM Press, 1991.

[Con71] John Horton Conway. *Regular Algebra and Finite Machines*. London Chapman and Hall, 1971.

[Hug96] John Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, pages 183–215. Springer LNCS 1110, February 1996.

[JHH+93] Simon Peyton Jones, C V Hall, K Hammond, W Partain, and P Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993. `http://www.haskell.org/ghc/`.

[Joh78] S. C. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, 1978.

[Jon94] Mark P. Jones. Dictionary-free Overloading by Partial Evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, June 1994.

[Law04] Mark V. Lawson. *Finite Automata*. CRC Press, first edition, 2004.

[LS04] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, November 2004.

[Mar05] Luc Maranget. Warnings for Pattern Matching. Under consideration for publication in *Journal Functional Programming*, March 2005.

[Par92] Will Partain. The `nofib` Benchmark Suite of Haskell Programs. In J Launchbury and PM Sansom, editors, *Functional Programming, Glasgow 1992*, pages 195–202. Springer-Verlag Workshops in Computing, 1992.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM Press.

[RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, 1993.

[The05] 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`, March 2005.

[Toz01] Akihiko 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.

[Tur04] David Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7):751–768, July 2004.