The University of York
Department of Computer Science
Programming Languages and Systems Group

Unfailing Haskell

Qualifying Dissertation

30th June 2005

Neil Mitchell

Abstract

Programs written in Haskell may fail at runtime with either a pattern match error, or with non-termination. Both of these can be thought of as giving the value $\perp$ as a result. Other forms of failure, for example heap exhaustion, are not considered.

The first section of this document reviews previous work, including total functional programming and sized types. Attention is paid to termination checkers for both Prolog and various functional languages.

The main result from work so far is a static checker for pattern match errors that allows non-exhaustive patterns to exist, yet ensures that a pattern match error does not occur. It includes a constraint language that can be used to reason about pattern matches, along with mechanisms to propagate these constraints between program components.

The proposal deals with future work to be done. It gives an approximate timetable for the design and implementation of a static checker for termination and pattern match errors.

Contents

Chapter 1

Introduction

1.1 Total Programming

Programming, particularly in a functional style, can be thought of as writing down a specification for a function. Normally programs correspond to partial functions (they may crash, or loop forever). However, it may be desirable to restrict the class of programs allowed to ensure that those permitted are total.

Turner [Turner 04] argues convincingly that what is needed is a functional programming language in which every program executes to completion, without raising an error. This language would then inherit a lot of the mathematical properties that are currently lacking in Haskell. In the discipline of total functional programming that Turner proposes it is impossible to write either programs that generate pattern match errors or that do not terminate. Another way of describing this restriction is that $\perp$ is removed from the language.

1.2 Benefits of Totality

If a particular program is known to be unfailing, this provides a number of benefits. The most obvious is that this knowledge allows the programmer to ignore the possibility of error conditions. The user will not encounter such a situation. Testing does not have to be carried out to check for totality.

One of the first uses a totality checker would probably see in real life would be to help the understanding of a programmer. Once a program has been written, usually it is the intention that it is total. Having this property checked automatically draws attention to any areas of the program which do not follow this assumption. These are likely to be the areas where bugs lurk.

To reason formally about a partial program requires the consideration of the $\perp$ value at every point. If $\perp$ is known not to occur, then proofs can be simplified. In addition, certain source level transformations that are suited to functional programs are more readily applicable.

Another use of a totality proof is that it can be used in a greater proof of correctness. Take an algorithm for sorting a list of integers: alongside the proofs of orderedness and permutation, a proof of totality would be desirable. If this part of the proof can be done automatically, then this is be of use.

A situation in which totality is particular useful is in embedded systems. In these systems, if a program crashes it is usually not possible for the end user to reboot the device. This means that a higher level of assurance is needed in the quality of the code.

A final benefit of totality relates to program optimization. If a compiler can determine that the program is total then the choice of evaluation strategy, namely eager or lazy, is not an issue of correctness. If a program is total, then under all evaluation strategies it will terminate. This means the compiler can pick the fastest or most memory efficient evaluation strategy. In addition, error messages can be removed and certain functions can make assumptions about their arguments, without requiring run-time tests.

1.3 Drawbacks of Totality

It is impossible in any computable framework to accept *all* total functions, yet reject *all* partial functions [Turing 37]. If all functions must be total, then there must be an undecidable set of total functions that cannot be described in this scheme.

There are also lots of functions that are not total, yet are still correct with respect to their specification. Consider a Universal Turing Machine, this cannot be written in a total manner. Operating systems are also non-terminating.

It is also likely that any scheme proposed will require refactoring of some programs. Programmers will have to specify their program in a way that meets a certain set of criteria, rather than in a more natural way. The programmer must then make the choice: is the effort spent refactoring the code worthwhile to obtain the benefits of provable totality.

1.4 A Totality Checker

My research goal is a totality checker for the functional programming language Haskell [Peyton Jones 03]. One particular aspect which requires deep consideration is Haskell's use of lazy evaluation – this complicates both pattern matching and to a larger degree, termination.

As totality is undecidable, so my tool must err on the side of caution – only claiming totality when a complete proof can be produced.

Chapter 2

Field Survey and Review

This review covers aspects of unfailing functional programming. The chapter starts off with the background properties required to understand the material. Static analysis is then covered. Next a motivation is given for total functional programming, followed by methods for ensuring termination, and for guaranteeing pattern match safety.

2.1 Background

This section covers the properties of functional programming that are required to understand fully the following sections. Where specific code fragments are given, usually this is done using the Haskell [Peyton Jones 03] programming language.

2.1.1 Bottom $\perp$

In Haskell, the semantic value for error is bottom, $\perp$. It can be thought of that every type is augmented with the additional $\perp$ value. However, unlike normal values, $\perp$ cannot be tested for. There are two different ways in which the $\perp$ value can be generated by a Haskell program, these are *pattern match failure* and *non-termination*. Other functional languages, for instance ML, do distinguish between these two classes, and allow the programmer to test for pattern match failure.

Pattern Match Failure

One example of possible pattern match failure is the `head` function, defined by the single equation:

```
head (x:xs) = x
```

Looking at the above equation, `head` is only defined if the argument is an application of a Cons constructor when evaluated; it is undefined for the empty list. If an empty list is passed as the argument to `head`, then an error such as `Program error:  pattern match failure:  head []` will be printed.

Non-termination

One example of non-termination is a function `non_term`, defined by the single equation:

```
non_term x = non_term x
```

In this case, the function `non_term` will never return a value. The exact behaviour is implementation dependent, sometimes a circular dependency will be detected at runtime, other times a stack overflow is caused and in other cases the program will become unresponsive.

Detecting whether a function, when applied to a particular argument, will terminate is undecidable, and is known as the halting problem [Turing 37].

2.1.2 Normal Form

An expression which can be rewritten or reduced at its root is called a *redex*. If no part of an expression can be reduced any further, then it is said to be in Normal Form (NF). The expressions in a language such as Haskell that are not redexes are applications of data constructors, unsaturated applications, literal values and λ expressions.

A weaker form than normal form is called Weak Head Normal Form (WHNF). In this form the leftmost, outermost expression cannot be reduced any further. This does not mean that the entire expression is in normal form – one example of an expression in WHNF, but not NF, is `(4 + 5) : [1..]`.

With functional programming languages, if there is more than one way to reduce an expression to a normal form, then all reduction strategies will result in the same normal form. It has also been shown that should a normal form exist, then the strategy of reducing the leftmost, outermost redex will find it. This strategy corresponds to lazy evaluation. It should be noted that innermost reduction – the strategy taken by strict languages – does not provide any guarantee about reaching a normal form should one exist.

2.1.3 Laziness

Programming languages can be divided into those which are *strict* and those which are *lazy*. In a strict language, when a function call is made, the arguments are all evaluated to normal form, then the function is executed until the result is in normal form. Typically a strict functional language will have certain non-strict elements within it – for example Lisp [Steele 90] defines `if`, `and` and `or` as lazy in all but their first arguments.

In contrast a lazy language doesn't have any special built in lazy elements. Every function is lazy. When a value is needed, it is then computed – if a value is never needed, it is never computed. This allows for the possibility of infinite data structures – for example the infinite list of prime numbers.

Consider the following example, which generates the infinite list of natural numbers, and then outputs the first 5 numbers.

```
nats = nats' 0
       where nats' x = x : nats' (x+1)

result = take 5 nats
```

Lazy evaluation of `result` yields `[0,1,2,3,4]`. Using strict evaluation, no result would ever be generated, as every single natural number would have to be generated first.

2.1.4 Higher Order

In higher order languages functions can be passed as arguments to other functions. This allows functions such as:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Note that the type of the first argument `f :: a -> b`, is functional.

First-order languages do not permit functions to be passed as values. Most languages have some form of higher-order expressions, for example the standard C [Ame 89] language does. However, in contrast to Haskell, the C form of higher-order functions consists of passing named functions by address – there is no λ construct. In C use of a higher-order function is a rarity, while in functional languages it is the norm.

Higher order functions allow greater reuse of components. Consider the definitions of `sum` and `product`. They are both very similar, but in a first order language it is hard to abstract over them. This can be achieved in Haskell using the following definitions:

```
sum     = foldr1 (+)
product = foldr1 (*)

foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

2.1.5 Peano Numbers

Peano's representation of the natural numbers uses the constant 0 and function S, where $S(x) = x + 1$. For example number 4 is represented as $S(S(S(S(0))))$. It is possible, and perhaps more intuitive, to uses these numbers instead of integers for most purposes in functional programming – a topic discussed in [Runciman 89].

2.2 Static Analysis

Static analysis a term used to describe the analysis of a program, and determining properties about it, without executing the code directly. It analyzes all branches of the code, and the analysis itself is guaranteed to terminate. Static analysis is used for many purposes, including checking for safety of programs, and for generating higher performance software. Much of the information in this section comes from [Mitchell 04].

Often in static analysis it is not possible to completely determine a property. In this circumstance, the analysis can either ignore a potential problem, or generate a possibly false warning. The choice of which to do depends on the purpose of the static analyzer.

There are many different models of static analysis, some of the more common ones are now described. More information on this topic can be obtained from sources such as [Nielson 99].

In order to show the difference between the various following approaches, I will use the imperative-programming example of calculating $z = x \times y$ without the use of multiplication, where all the variables are positive integers, and x is destroyed. This can be done by the following code:

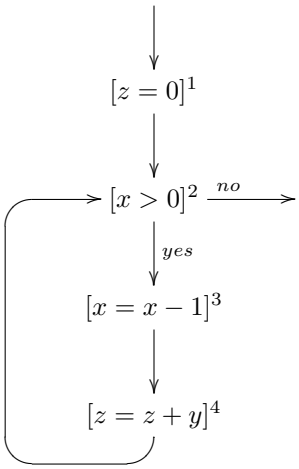

Figure 2.1: Multiplication example as a dataflow graph

```
[z = 0]^1;
while ([x > 0]^2) {
    [x = x - 1]^3;
    [z = z + y]^4;
}
```

2.2.1 Data Flow Analysis

For data-flow analysis, the program can be considered as a graph. Each statement in the program then corresponds to a node, and the flow between the statements as directed arcs on this graph. Program statements such as `if` and `while` lead to situations where two arcs either arrive at or leave a single node. The example given above would translate into a graph as shown in Figure 2.1.

Using the graph representation, equations are then set up to define the reaching definitions, where a variable obtains its value from. Separate definitions are provided for the entry and exit to each node, allowing the output to be modelled in terms of the input. For example, statement 2 does not modify the state so $RD_{entry}(2) = RD_{exit}(2)$. Before the program is executed, on entry to statement 1, none of the values have a known reaching definition so $RD_{entry}(1) = \{(x, ?), (y, ?), (z, ?)\}$. However, after statement 1 has been executed $RD_{exit}(1) = \{(x, ?), (y, ?), (z, 1)\}$. Note that the value associated with z is 1, corresponding to the statement that set it, and not the value it was set to.

After these reaching definitions have been created, they are then solved to determine where a variables value at any point comes from. This information can then help show what the value of the variable is, and how it changes. In the course of solving the equations, it may be necessary to decrease the precision of the analysis, to ensure termination.

2.2.2 Constraint Based Analysis

Constraint based analysis uses a similar graph representation to data flow analysis, however it focuses on the node instead of its entry and exit points. Each statement is reformulated as a constraint, describing the

effect of the node. These constraints are then solved using general constraint solving techniques. This type of analysis seems to be more common when working with functional programs, and vast simplifications rely on the the absence of side effects.

The example given for data-flow analysis could be reformulated as a constraint based problem. The exit from the first statement can be represented as $RD_{exit}(1) \supseteq \{(z, 1)\}$. Note the use of $\supseteq$ as a constraint, many of these can be generated and then solved to find a fixed point.

2.2.3 Forward Analysis

In this method the program is executed on a generic state, which represents all the possible states the program could be in just after a particular statement. As the statements are executed, the state is modified, and the end state can be used to deduce properties on. Where a statement can be executed more than once, for example statement 3, the generic state must cater for all these possibilities.

Finding suitable abstractions for each variable is often specific to particular problems. One method often used for numeric variables is range analysis, determining which variables may be between which values. For example, after statement 1 has been executed $0 \leq z \leq 0$, while all other variables are unknown.

2.2.4 Backwards Analysis

Backwards analysis takes a postcondition at the end of the procedure, and transforms it over all the statements in reverse order to obtain a precondition. This is particularly useful when the exact properties required at the end can be determined before analysis starts.

The disadvantage of using backward analysis compared to the other methods is that sometimes it becomes counter intuitive. When an error is reached, it is not always easy to pinpoint where the error is – only that it exists. On the other hand, forward analysis is generally able to give the precise statement which caused the error.

Using backward analysis on the above example, if the postcondition for statement 4 was that $z \geq 0$ then the precondition would be $z + y \geq 0$. When transformed over the entire program, $z \geq 0$ would become $y \geq 0$. However, taking the postcondition $z > 0$ would generate $y > 0 \wedge x > 0$ as a precondition to the function.

2.3 Total Functional Programming

Total Functional Programming is about designing a new computer programming language, which from the very lowest level has no possibility of runtime errors, or at a semantic level the value $\perp$ is removed [Turner 04]. When performing proofs on functional programs currently the value bottom must be considered, and often the resulting proofs are only valid if the computation completes successfully.

Take for example the well typed expression:

```
loop :: Int -> Int
loop n = 1 + loop n
```

It is tempting to manipulate this definition as if it were an equation, in the following steps.

```
loop n = 1 + loop n     //from the definition
loop 1 = 1 + loop 1     //instantiate with 1 for n
loop 1 - loop 1 = 1     //subtract loop 1 from both sides
0 = 1                   //using x - x = 0
```

Unfortunately now a falsehood has been proven. If the definition being manipulated was known to be total, then the above steps would have preserved correctness. In a partial language these type of manipulations are not safe.

Checking for totality can be decomposed into two separate checks: checking for termination and checking for pattern match safety.

2.4 Termination Arguments

There are many arguments for ensuring termination. Most rely on a decreasing value, which has a lower bound and a minimum decrease at each step. Some of the more common termination arguments are presented here.

2.4.1 Primitive Recursion

There is a strict subset of recursive functions over the natural numbers called *primitive recursive*. Every function in this subset always terminates, however there are still many non-primitive recursive functions that terminate. A basic primitive recursive function is defined as one of:

- The constant function 0
- The successor function S, from Peano Numbers
- The projection function P_i^n, which takes n arguments, and returns the ith argument

From these, some operators are defined to combine them. A function is primitive recursion if it can be constructed from the basic functions, applying either composition or recursion a finite number of times.

Composition

Composition takes a primitive recursive function f, of arity i, and i primitive recursive functions of arity j being $g_1, \ldots, g_i$, and creates a function h of arity j where:

$$h(x_1, \ldots, x_j) = f(g_1(x_1, \ldots, x_j), \ldots, g_i(x_1, \ldots, x_j))$$

Recursion

Recursion takes a primitive recursive function f, of arity i, and a primitive recursive function g, of arity $i + 2$, and creates a function h of arity $i + 1$ where:

$$\begin{aligned} h(0, x_1, \ldots, x_i) &= f(x_1, \ldots, x_i) \\ h(S(n), x_1, \ldots, x_i) &= g(h(n, x_1, \ldots, x_i), n, x_1, \ldots, x_i) \end{aligned}$$

This is the only form of recursion is allowed, and it can be seen that the first argument to n is monotonically decreasing, with a lower bound, which ensures termination.

Ackermann's function

Despite the restriction in power of primitive recursion, most useful functions can be written in primitive recursive form. One such function for which this is *not* the case is Ackermann's function:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{otherwise} \end{cases}$$

This function cannot be rewritten in a primitive recursive manner, and it grows in size faster than any primitive recursive function ever could. However, this is only a first order result. In a higher order language, it is possible to automatically rewrite this to a primitive recursive form.

A Haskell implementation of Ackermann's function following the above definition is:

```
ack 0     n     = n+1
ack (m+1) 0     = ack m 1
ack m     (n+1) = ack (m-1) (ack m n)
```

This definition can be converted automatically to:

```
ack 0     n = n+1
ack (m+1) n = ack' (ack m) n

ack' f 0     = f 1
ack' f (n+1) = f (ack' f n)
```

This second version is indeed primitive recursive. The original definition of `ack` uses nested structural recursion, where one argument to the recursive function is the result of that function. All cases of nested structural recursion can be automatically transformed into higher-order primitive recursive functions. In a higher-order system all recursive functions whose totality can be proved in first order logic can be written with primitive recursion [Turner 04].

2.4.2 Syntactic Termination

One way of guaranteeing a terminating function is for all recursive calls to have syntactic descent, provided there are no infinite data structures.

```
length [] = 0
length (x:xs) = 1 + length xs
```

Turner suggests that this restriction "does not deprive us of any functions that we need" [Turner 04]. He does note that refactoring is likely to be required. He also states that many definitions that are not primitive recursive have an equivalent definition of the same time complexity which is.

Beyond Structural Recursion

It is possible to extend from using simple structural recursion to Walther Recursion [McAllester 96]; though no more powerful than primitive recursion, it is often more convenient. The concept of Walther recursion is to

go beyond the standard primitive recursive test of simply decrementing a natural number. Functions in this scheme can either act as reducers or conservers. Reducers decrease some measure, for example subtraction on natural numbers where both numbers are non-zero is a reducer. Conservers make no change to the measure, for example `id`.

Unfortunately this is only semi-decidable, so a reduced class based on syntactic constraints has been defined. In this scheme a projection can be used as either a reducer or a consumer. A projection takes some element of data from a constructor, for example `pred` and `tail` and both projections.

A particular example of an operation that is well founded in Walther Recursion but not in Structural Recursion is the fast exponentiation algorithm.

$$\exp(x, n) = \begin{cases} x & \text{if } n = 1 \\ \exp(x^2, n/2) & \text{if } n \text{ is even} \\ x \times \exp(x, n-1) & \text{otherwise} \end{cases}$$

Note that this uses $n' = n/2$ as its termination guarantee – rather than the more standard $n' = n - 1$. It is instructive to give this in standard code, on natural numbers, and follow the termination argument.

```
data Nat = Z | S Nat
pred (S x) = x

exp x Z     = S Z
exp x (S Z) = x
exp x n | isEven n  = exp (x ^ 2) (div2 n)
        | otherwise = exp x (pred n)

div2 (S Z) = Z
div2 (S (S x)) = S (div2 x)
```

The details of `(^)` are ignored here. The interesting thing to note is the proof of termination for `exp` is in its second argument. The first two clauses for `exp` have no recursive call, and are therefore terminating. The `otherwise` branch uses simple primitive recursive reduction. Using Walther recursion `pred` is classed as a reducer, which shows that this equation is terminating. The most interesting case is that a termination argument is needed to show that $\forall n \bullet n <$ `div2` n, namely that `div2` is a *reducer*.

The first equation of `div2` can be seen to be a reducer, since the argument and result are both concrete values, and the argument is greater than the result. The second equation is also reducing, and hence `div2` is a reducer. This means the entire program is terminating by use of Walther recursion. Unfortunately, this hides one important caveat. The function `div2` is terminating, but it is not total. The application `div2 Z` is undefined. In fact, `div2` is never called in a situation where the first argument is zero, but if a simple exhaustive pattern requirement is imposed, this does not help. Also note that if a equation had been given of the form `div2 Z = Z` (as you might expect in a standard definition), then `div2` would *not* be a reducer, and the program would *not* be proved terminating.

While Walther recursion does increase the number of recognized definitions, there are still many natural definitions of functions which can be written as primitive recursive functions, that are not accepted as Walther recursive.

2.4.3 Data vs Codata

In a total functional programming language it would be nice to have potentially infinite data, along with normal finite data. One particular area where this would be useful is for embedded systems. Often an

embedded system can be modeled as a stream of inputs, and a stream of output – with a function sitting in the middle generating the outputs based on the inputs. In this view, the stream of inputs and outputs are both potentially infinite, so an infinite (and hence non-terminating) stream is required.

With codata [Telford 97], instead of having a requirement that the entire data structure must terminate, the requirement is that the generation of any subsequent element must terminate. The computation is *productive* – it never fails to perform needed further work. For example, a data structure for infinite linked lists, along with the definition of the even numbers, could be given as:

```
codata CoList a = CoCons a (CoList a)
                | CoNil

evens :: CoList Int
evens = f 2
    where f x = CoCons x (f (x+2))
```

This definition will clearly produce an infinite list – there is no terminating case. It is however also possible to show that this function will never stall; it is always possible to compute another element. The argument for productivity would be formulated by noting that the first part of the result `f x` produces is `CoCons x`, before recursing. This means that a `CoCons` constructor is always applied to a value at every step, therefore `evens` is productive.

2.4.4 Sized Types

The idea behind *sized types* [Pareto 98, Hughes 96b] is very similar to that of total functional programming, in that functional programs can be written in a manner which ensures termination. The way this is achieved is by ensuring termination at the types level. The standard type system is expanded by including sizes of items such as linked lists. Natural numbers are represented as Peano numbers. The size of the result of a function is based on the size of its inputs, in a linear combination. For example, the type of `append` becomes:

```
append :: [x] -> [x] -> [x]
append ::  a  ->  b  -> a + b
```

The first line corresponds to the standard list type signature – taking two lists, and returning a third one. The second line is the sized type definition, where a and b are the lengths of the two lists, and $a + b$ is at most the length of the result.

Most sized type checkers only permit linear constraints, and use upper bounds [Pareto 98]. For example the `filter` function which takes a condition, and a list, and returns those items which meet the condition. The derived type of `filter` states that it takes a list of size n and returns a list of size at most n.

Termination is then proved on each individual function, and composes upwards quite naturally.

To show the power and weaknesses of the sized types approach, take the functions insertion sort and quick sort. Insertion sort could be defined by:

```
isort []     = []
isort (x:xs) = insert x (isort xs)

insert n []     = [n]
insert n (x:xs) = if n<=x then n:x:xs else x:insert n xs
```

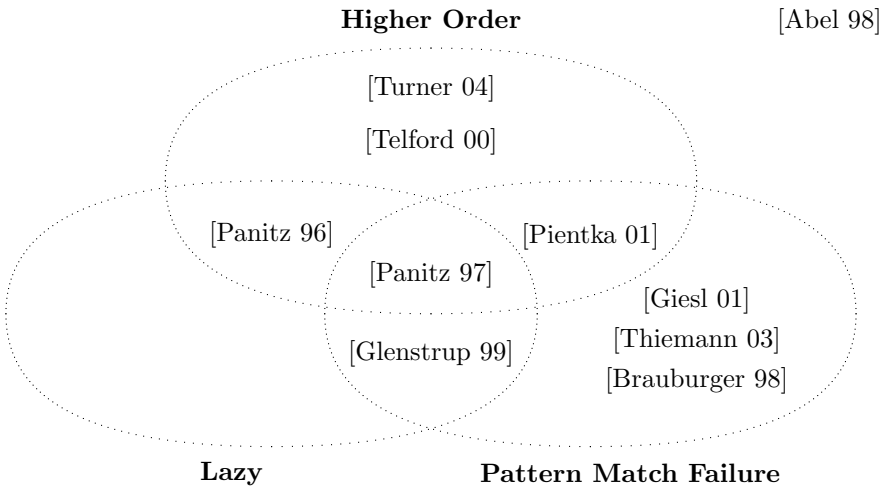

Figure 2.2: Termination Checkers

The sized types mechanism would first assign the type of `insert` as `_ -> n -> n + 1` – namely it adds one element to the list. Using this fact, it can derive that the type of `isort` is `n -> n`. These can all be seen on a case by case basis.

Taking the more complex quick sort:

```
qsort []     = []
qsort (x:xs) = qsort l++[x]++qsort h
    where l = filter (<= x) xs
          h = filter (>  x) xs
```

This definition relies on `filter`, which has the type `_ -> n -> n`. The sized type analysis discards too much information to succeed. Within the types, this merely means that the length of the result of `filter` is less than or equal to the length of its list argument. Following this through, `l` and `h` both have the type `n-1`, where `n` is the length of `(x:xs)`. This means that `qsort` then has a type of `n -> n`^{2}, which cannot be represented, so is changed to `n -> w`.

In the particular case of `qsort`, a human can deduce that length `l` plus the length of `h` equals the length of `x`, but the sized types cannot. This is the deduction required to prove that the type of `qsort` is `n -> n`.

2.5 Termination Checkers

There are quite a few termination checkers for functional languages. Figure 2.2 classifies them by which features they support. Some researches developing termination checkers invent their own language [Abel 98], while others work on existing languages such as Lisp [Glenstrup 99], Haskell [Panitz 97] or Erlang [Giesl 01, Armstrong 93].

It is not perfectly clear in all cases as to what is a termination checker. The Total Functional Programming papers by Turner [Turner 04] specify an entirely new language which, as part of the compilation process, is

checked for termination. This can either be seen as a compiler which checks that the program is well formed according to the language definition, or it can be seen as a compiler and a termination checker.

From the diagram in Figure 2.2, it is easy to see that there has been quite a lot of work on languages with pattern match failures. Almost every functional language has non-exhaustive pattern matching, and essential functions such as `head` or `car` depend on this behaviour. Rather less work has been done on higher order functional languages, although still a reasonable amount – again modeling the fact that most functional languages have higher order features. When it comes to laziness, there is a distinct lack of termination checkers; this may either be because the task is far harder, or because fewer functional languages are lazy.

When classifying the checkers, it is often not clear cut which properties a language has, or does not have. Although [Panitz 97] does its checking on Haskell programs, and Haskell is a lazy language, it is not clear whether this checker accepts programs that do not terminate if evaluated strictly. Also the treatment of pattern match failure is varied: most checkers treat this as a successful termination – possibly not the appropriate model, depending on the semantics of the language. Some checkers also disallow non-exhaustive patterns entirely.

2.5.1 Prolog Termination Checkers

While there are some termination checkers for functional languages, when it comes to the logic programming language Prolog [ISO 95], there are far more checkers available. The question then arises as to which of the three properties discussed above appear in Prolog.

Non-Exhaustive Patterns

Prolog clearly has non-exhaustive patterns, for example `List = [Head | Tail]` will not unify if `List` is the empty list (or in fact, any non-list type either). However, in Prolog this is not the same kind of error as in a functional language. A failed unification is only a soft failure, and causes the computation to backtrack.

Higher Order

There is a literature on higher order logic programming [Naish 96]. Using the `call/N` primitive, it is possible to write definitions such as `filter` in the following way:

```
filter(_P, [], []).
filter(P, A0.As0, As) :-
   (call(P, A0) -> As = A0.As1 ; As = As1),
   filter(P, As0, As1).
```

Lazy Evaluation

While Prolog is not lazy, it is not strict either, but implements a backtracking approach. This is similar to lazy evaluation in many ways, and lazy evaluation can be encoded as free variables [Antoy 00]. It is also possible to use lazy evaluation to encode backtracking, by generating a lazy list of successes [Wadler 85].

Termination Checkers

As mentioned earlier, there are many termination checkers for Prolog. Most of them attempt to find some size-change measure, which can be used to prove termination. Prolog termination checkers are quite advanced,

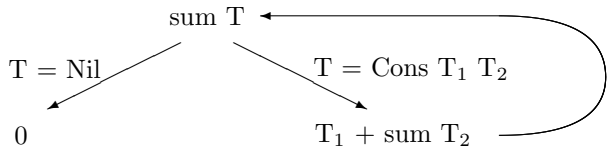

Figure 2.3: Tableau proof for termination of `sum`

with many tools using tabulation, constraint solvers and other complicated techniques. Out of various papers has emerged a set of standard problems which solvers are checked on. It should be noted that no solver successfully manages to discharge all of them. The standard set includes:

- List manipulations - map, reverse, append, concatenate, sort
- Peano numbers - equality, add, divide, greatest common divisor, Ackermann
- String functions - prefix, matching
- Graphs - graph colouring, path finding
- Parsing - parsing expressions, type checking
- Interpreters - boolean expressions

Unfortunately it appears that when encoding laziness and higher-order functions in Prolog, most of these checkers seem to fail to find termination proofs.

2.5.2 A detailed look at TEA [Panitz 97]

Out of all the termination analyzers examined, the only one that could have been said to have any support for all three properties is TEA, which operates on the Haskell programming language.

The way TEA works is by translating Haskell into a core language, which preserves the semantics but has a simpler representation. After this it uses *tableau proof* [D'Agostino 99] to generate orderings over various variables. When it finds errors, including those generated by pattern match failure, it treats them as successful termination. TEA's authors claim to have a "90% success rate" [Panitz 97] on programs, including functions from the prelude, and those used to perform the analysis.

The type of termination that TEA checks for is normal form (NF) termination. A function f of arity i is *NF-terminating*, if when given i arguments, $v_1, \ldots, v_i$, all of which are in normal form, the result of $f(v_1, \ldots, v_i)$ ends up in normal form. While this is one form of termination, it is a more natural definition for a strict language. In a lazy language, something similar to WHNF termination would be more appropriate.

To take a simple example, the following definition of a function `sum`:

```
sum Nil = 0
sum (Cons x xs) = x + sum xs
```

is transformed into the tableau proof shown in Figure 2.3. For the `sum` function to terminate, there must be an ordering under which $T > T_2$. Since T was originally substituted, it can be replaced with Cons T_1 T_2 > T_2, which can be solved quite easily. The ordering in this case is that the value of a list is its length.

2.6 Pattern Match Checking

The other side to the totality problem is that of checking that pattern match errors do not occur. Most functional languages have some sort of `case` expression at their core – although they may present this in a variety of ways. In Haskell for example, the `if` expression can be seen as a sugared `case` expression, and so can pattern matching.

2.6.1 Simplifying Case Statements

Often `case` expressions are composite – testing for more than just one case at a time. For example, the following expression:

```
case x of
    [y] -> f y
    _   -> g x
```

the first branch tests that the root of `x` is a Cons application, and that the tail of `x` is a Nil constructor. This can be expanded to:

```
case x of
    (y:ys) -> case ys of
                  (z:zs) -> g x
                  []     -> f y
    []     -> g x
```

In this expanded expression all case expressions check the root constructor, and all case expressions are exhaustive. A case matching error can then be modelled as a call to a special `error` primitive as the result. Details of how to perform this conversion are covered in [Augustsson 85] and [Wadler 86].

2.6.2 Checking for exhaustiveness and usefulness

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

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

notExhaustive (Just x:xs) = Just x
notExhaustive [] = Nothing
```

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.

The rationale behind flagging the first function as a probable error should be quite simple to understand: the programmer would not have written the equation if they did not think it was needed, and this implies that the programmer has misunderstood something.

The second function is less obviously an error. Consider the function `head`; when written correctly this would be flagged as non-exhaustive. While this may be a useful warning message, it is not as severe as the redundancy error message.

These warnings are implemented in ML [Sestoft 96], using the internal compiler representation of the case expressions. Another mechanism which is suitable for compilers which use alternative representations is presented in [Maranget 05].

When trying to compile these examples using GHC 6.4 [The GHC Team 05], the second provokes a warning, but the first 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"* [The GHC Team 05].

These checks are *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.

2.6.3 Requiring Exhaustive Patterns

The pattern match failure problem could be solved by simply requiring all pattern matches to consider all possible values – disallowing the natural definition of `head`, for example. Turner suggests that this would "force you to pay attention to exactly those boundary cases which are likely to cause trouble" [Turner 04].

One way of rehabilitating the `head` function is to let each individual situation where `head` would have been required perform its own case matching. If a function such as head is still required, then there are several ways to rewrite it in a case complete version, some of which are discussed below.

Maybe wrapping

```
head :: [x] -> Maybe x
head []     = Nothing
head (x:xs) = Just x
```

Here the result is returned as a wrapped type. The disadvantage is that a case match must be used to check for `Nothing` or `Just`. This style of programming is explored in [Spivey 90].

Error value

```
head :: x -> [x] -> x
head no []     = no
head no (x:xs) = x
```

In this version an additional parameter is passed in, being the value to return for failure. The failure value must have the same type as the elements of the list.

Continuation passing

```
head :: z -> (x -> z) -> [x] -> z
head no yes []     = no
head no yes (x:xs) = yes x
```

In this system `yes` is a function. If the list argument is non-empty, then the function is applied to the first element. Otherwise the `no` value is returned. Note that the `no` value does not have to be the same type as the elements of the list.

Discussion

One property that all of these functions have in common is that they change the type of `head`. This means that existing code that calls `head` must be refactored to take this into account. All the approaches are useful in different situations, with the `Maybe` approach being perhaps the least invasive in existing code.

2.6.4 Addition of `error`

In Total Functional Programming there are functions for which there is no sensible answer, for example $n/0$. It is possible to augment the total language with a special error construct, which indicates that this value can never arise. Starting from an initial total functional programming language, it is possible to add the `error` keyword. There are no associated semantics with this keyword, and it is the analyzers job to check that an error value cannot occur.

An analysis that checks for just this is presented [Telford 00]. This works by attempting to give subtypes to expressions, and ensure that the `error` keyword cannot be reached.

2.6.5 Testing for $\perp$

One widely used tool for testing properties in Haskell programs is QuickCheck [Claessen 00]. Properties are written which the program must satisfy. By randomly generating data values, then exercising these properties, this will likely provoke any pattern match errors. This can be combined with Chasing Bottoms [Danielsson 04], a function which can test for $\perp$. A generator can be given for the data, so an appropriate subset of possible data values can be used.

Another option is to use lazy assertions [Chitil 03]. Assertions are embedded within Haskell programs, but their evaluation is lazy, so does not affect the behaviour of the program. Asserting properties about the shape of a data structure can trap potential pattern match errors at an earlier stage, even if a particular execution does not trigger them.

These approaches all rely on checking at runtime, so the problem is that they are not complete (only specific runs are checked) and their execution may be slow if multiple runs are required. Also, if automatically generating suitable input is hard, certain classes of data structure may never be generated, and any errors associated with them will go undetected.

2.6.6 Soft Typing

Another field of research which bares some similarities to the pattern match checking problem is that on soft types [Aiken 91, Aiken 94]. This looks at assigning some static type information to a program in a dynamically typed programming language. A lot of the work in this field is by Aiken, and concerns the dynamically typed language FL [Backus 90].

The advantage of giving some static type to a dynamically typed language is that some runtime checks can be removed, and certain specializations can be performed. Take for instance the multiplication function. When multiplying two arbitrary numbers this is slower than when multiplying two integers, as most computers

have a special integer multiplication instruction. If static type information can be recovered, then this optimization can be taken.

The language representing type expressions at any point is a set of all possible constructors – including the empty set, and the universal set. Types can be joined with union and intersection. In one paper [Aiken 94] there is some notion of conditional types, which allows types to be dependent on the value and type of other expressions.

Since individual constructors are represented, in this soft typing mechanism it is possible to differentiate between a list (being both Cons and Nil), and an empty list (being just Nil). This gives the power to statically check functions such as `head`.

A final important point about this approach is that if some area of the program cannot be assigned an accurate type, it does not break down. Instead a most general type is assigned to this one specific area, and the result of the program remains typed.

2.6.7 Type Analysis for XML

Outside of the functional programming literature there are other problems which are very similar to pattern match failure detection – primarily those involving XML [Bray 04] and XSLT [Clark 99]. 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 first order 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 [Tozawa 01] 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.

2.7 Conclusion

In the literature review I have presented quite a lot of work on termination checkers, but none have really tackled the question of lazy termination with any satisfactory results. There has been some work on detecting pattern match errors statically, but only in a very local context – not using a global program analysis.

The underlying theory of termination has been explored extensively, particularly in relation to Prolog. The question of pattern match failure has not had the same depth of attention given to it, and that theory requires further development. The work on soft typing and other type analysis seems relevant, but has not been used for this particular purpose.

The other area that has not been explored concerns the needs of the programmers. If a programmer had a mechanism for proving totality available, would they take it? Given that it may take a significant amount of work to obtain the required proofs, at what point would the programmer stop? The only way to answer

these questions is to make a totality checker available, and see just how much effort is required, and if this has any measurable change in the way a programmer operates.

Chapter 3

Results

3.1 Overview

The majority of this section comes from a recently submitted paper [Mitchell 05]. The goal of this first stage is to write a static checker for a Haskell subset which finds any non-exhaustive patterns that may cause a runtime error.

3.1.1 Motivating Example

In order to show what this 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)
```

In the third equation, the `where` matches 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 this program over to the checker developed in this paper, the output from the checker is:

```
> (risers (y:etc)):Cons
> True
```

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

$$
\begin{array}{rcl}
E & ::= & \texttt{arg}\ m \\
& | & \texttt{path}\ E\ \ C\ \ m \\
& | & \texttt{make}\ C\ \ E_1 \cdots E_n \\
& | & \texttt{func}\ f \\
& | & \texttt{apply}\ E_0\ \ E_1 \cdots E_n \\
& | & \texttt{case}\ E_0\ \texttt{of}\ \{C_1\ \texttt{->}\ E_1\texttt{;}\ \cdots\ \texttt{;}\ C_n\ \texttt{->}\ E_n\}
\end{array}
$$

f is the name of a function
C is the name of a constructor
m is a positive integer

Figure 3.1: Abstract Syntax of expressions in reduced Haskell

3.2 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. The abstract syntax of expressions is given in Figure 3.1.

Example 1

```
data List = Cons | Nil

head = case #1 of Cons -> #1.Cons#1

reverse = rev #1 Nil

rev = case #1 of
    Nil -> #2
    Cons -> rev #1.Cons#2 (Cons #1.Cons#1 #2)

map = case #2 of
    Nil -> Nil
    Cons -> Cons (#1 #2.Cons#1) (map #1 #2.Cons#2)
```

□

Note that 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.Cons#2` refers to the second construct of the `Cons`-constructed first argument. (From here onwards `Cons#1` and `Cons#2` will be written as `head` and `tail` respectively, but this is purely to aid understanding for the human reader.)

This reduced language is *not* intended for people to program in. Rather programs in this language are obtained by transforming programs in standard Haskell.

3.2.1 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.

```
plus x y = case x of
       Positive -> case y of
              Positive -> Positive
              Zero -> Positive
              Negative -> anyInt
       Zero -> case y of
              Positive -> Positive
              Zero -> Zero
              Negative -> Negative
       Negative -> case y of
              Positive -> anyInt
              Zero -> Negative
              Negative -> Negative
```

Figure 3.2: The `plus` function in reduced Haskell

There are no primitive data types. A type such as `Int` is modelled as `data Int = Positive|Zero|Negative`, by replacing the primitively defined sections of the prelude. In this representation, the `plus` function can be written as in Figure 3.2. The 'magic' symbol `anyInt` may evaluate to any of the `Int` constructors.

An alternative modelling would be using Peano numbers, perhaps with a flag to indicate negative vs positive status. This would give more information, but could lead to an explosion in the size of the conditions.

The `Char` basic type can be modelled as an enumeration, since there are only a finite number of characters. An alternative would be a reduced representation such as `data Char = Upper | Lower | Digit | Other`.

3.2.2 Expressions

Expressions in reduced Haskell are defined in Figure 3.1. Some are very similar to those in Haskell, although none is quite the same:

`arg` m denotes the mth argument of the function in whose body it appears

`path` E C m follows the mth field of the expression E – which must be a value with constructor C. Static typing and `case` expressions ensure that this is always a safe operation to perform.

`make` C $E_1 \cdots E_n$ creates a new data object, with the given fields, all the required fields must be supplied.

`func` f`;` simply refers to a top level named function.

`apply` E_0 $E_1 \cdots E_n$ applies function E_0 to arguments $E_1 \cdots E_n$. It is not necessary that all the arguments to the function are supplied. E_0 must evaluate to a function.

`case` E_0 `of {`C_1 `->` E_1`; ⋯ ;` C_n `->` E_n`}` has patterns that are just single constructors with no binding variables. If projections are needed they can be expressed using `path`. E_0 must be an algebraic value.

Due to static typing of programs before translation, all expressions except `case` are incapable of causing any error, apart from non-termination. The things that are lost in this reduced language include local definitions such as `where` and `let`, deep patterns, complex case statements and lambda expressions. None of these add any power to the language; they are merely convenient – even though to program without them would be highly challenging.

3.2.3 Converting from Haskell

A converter from a subset of Haskell to this reduced language is used as the main input mechanism for the checking program. The main difference between the subset of Haskell and the reduced language is the removal of all variable names as arguments to functions, and in `case` options – and the replacing of those with variables and paths. These are relatively simple transformations.

A better target would be a converter from full Haskell. By combining a number of existing utilities, such as Haskell All-In-One [Daumé III 03], and GHC's ability to generate Core [Tolmach 01], after let lifting and lambda floating one obtains a language very similar to that currently accepted by the checker.

3.2.4 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 nth 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 nth 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
       Nil -> Nil
       Cons a b -> Cons (f a) (map f b)

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

is transformed into:

```
map_adds n xs = case xs of
       Nil -> Nil
       Cons a b -> Cons (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 [Hughes 96a].

3.3 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. For this purpose we introduce a constraint language by a few examples, followed by a more formal description.

Example 3

Consider the expression `head` α. The constraint that must hold for this expression to be safe is α`:Cons`. This says that the expression α must reduce to an application of `Cons`, i.e. a non-empty list. □

Example 4

Consider the expression `map head` α. In this case the constraint is α`.*tail.head:Cons`. 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 `Cons`. 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*.

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 subtype 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
$\langle C, n\rangle$	a constructor C and an integer n
λ	the language is the set containing the empty string
ϕ	the language is the empty set

The elementary regular expressions are of the form $\langle C, n\rangle$ where C is a constructor and n is its nth field.

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

So the first example, α`:Cons`, could have been written more fully as α`.`λ`:Cons` – where λ is the regular expression which describes the language consisting only of the empty word.

A constraint of the form α`.`r`:`c can also be expressed as a tuple $\langle \alpha, r, c\rangle$. Using the tuple form is more convenient when describing certain properties formally. The meaning of a constraint is defined by:

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

$$\text{exists}(e, l) = \begin{cases} True & \text{where } \#l = 0 \\ \text{constructor}(e) = C \ \wedge & \\ \quad \text{exists}(e.\langle C, n\rangle, \omega) & \text{where } l = \langle C, n\rangle.\omega \\ False & \text{otherwise} \end{cases}$$

Where $L(r)$ is the (possibly infinite) 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.

Example 5

`(A C C).B#1:C` is true because in the expression `A C C`, there is no `B#1` selector to follow (only `A#1` and `A#2` are available), and so no condition is applied. □

Another way in which a tautology can occur is if the condition lists all the constructors of a type. For example, `x:(Nil,Cons)` is always true. Because of static typing, it must be the case that the type of `x`, and the type of the conditions match. Furthermore, because the condition lists all possible constructors for a list, it must either be the case that `x` is an application of one of them, or `x` has the value $\perp$. If the $\perp$ arose through an incomplete case statement, this would be picked up at the source. If the $\perp$ is the result of non-termination, then the case statement will never be executed, as the program will still be trying to evaluate `x`, and hence no case error will be raised.

A final special case is α`.`ϕ`:`c, because the regular expression is ϕ, which is the empty language, this condition is always true.

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.

3.4 Determining the Constraints

The checker's job is to generate constraints, and then propagate them round the program, transforming them as necessary. 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.

3.4.1 The Initial Constraints

The first step of the checker is to determine an initial set of constraints by detecting all non-exhaustive case statements.

Example 6

For the following definition of `head`

```
head = case #1 of Cons -> #1.head
```

the generated constraint is `head#1:Cons`. □

In general, a `case` expression:

```
case α of
       Sel1 -> val1
       ...
       Seln -> valn
```

produces the condition α `:` `(`$\mathtt{Sel}_1$`,`$\mathtt{Sel}_2$`,...,`$\mathtt{Sel}_n$`)`. Of course, if the case alternatives are exhaustive, then this can be simplified to True.

All `case` expression in the program are found, their initial constraints are found, and these are joined together with conjunction.

If α is a complex expression, and not a simple argument reference, then it must be replaced with an equivalent expression on arguments. The process of transforming from a constraint on an expression, to one only on arguments is a kind of backward analysis (see §3.4.3).

If a constraint on an argument to a function is generated, where the function has a recursive call, then the constraint must have a fixed point found (see §3.4.4).

3.4.2 Transforming the constraints

Once a set of initial constraints has been generated, they are transformed. At each stage the constraints are expressed as a logical formula, where each atom is a constraint, and every subject of a constraint is an argument reference.

For each constraint `f#n` in turn, the checker searches for every function call of `f`, and gets the expression corresponding to its nth argument. On this expression, it sets the existing constraint. This argument is then transformed using a backward analysis (see §3.4.3), until a constraint on arguments is found.

Example 7

Given that `head#1:Cons`, if the program contains the expression:

```
f = head (g #1)
```

Then the derived constraint is `(g f#1):Cons`. i.e. the expression passed as `head`'s first argument must be a `Cons`. □

If the subject of a constraint is an argument to the `main` function, this cannot be discharged, and is output to the user. In certain cases, either through the checker being conservative, or through a genuine coding error, it is possible that a condition that evaluates to False can be generated. These are also recorded as errors, and reported to the user.

At each stage of the computation constraints obtained are reported to the user, to help determine the steps the checker performed. Eventually the constraint becomes either True or False, and the checker terminates.

3.4.3 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 3.3. 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.

The `arg` rule is quite simple: the only thing that changes is that the argument is qualified before being put 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 `path` rule says that if a constraint is satisfied on the expression used at the end the path, then following this bit of the path will give the constraint.

The `make` rule is more complex. 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 (ewp) [Conway 71], and can be calculated structurally on the regular expression. For

$$\varphi(\texttt{arg}n, r, c) \rightarrow \text{qual}(n).r : c$$

$$\frac{\varphi(E, r, c) \rightarrow \langle E', r', c' \rangle}{\varphi(\texttt{path}E\ \ C\ \ m, r, c) \rightarrow \langle E', \langle C, m \rangle .r', c' \rangle}$$

$$\frac{\varphi(E_1, \frac{\partial r}{\partial \langle C,1 \rangle}, c) \rightarrow E'_1, \cdots, \varphi(E_n, \frac{\partial r}{\partial \langle C,n \rangle}, c) \rightarrow E'_n}{\varphi(\texttt{make}C\ \ E_1 \cdots E_n) \rightarrow (\lambda \in L(r) \Rightarrow C \in c) \wedge E_1 \wedge \cdots \wedge E_n}$$

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

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

Figure 3.3: Specification of backward analysis, φ

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 $\partial r / \partial \langle C, i \rangle$. The differentiation method is based on that described in [Conway 71].

After differentiating a regular expression with respect to an input symbol, the result can be thought of as the regular expression that would match the rest of the string. This can be used to test for membership in the following way:

$$\begin{array}{rcll} \omega \in L(r) & = & \text{ewp}(r) & \text{where } \#\omega = 0 \\ a.\omega \in L(r) & = & w \in L(\partial r / \partial a) & \text{where } a \in \Sigma \wedge \omega \in \Sigma* \end{array}$$

If no string starting with this input symbol could be accepted then the result of the differentiation is ϕ. So $\alpha.\phi : c$ is always true, matching the semantics of the constraints quite nicely.

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 reason for automatically assuming that any constraint on the result of a function in the process of being evaluated is true can be best demonstrated with an example. Consider the following function:

```
f x = case x of
         A a -> f a
         B b -> b
```

Consider a constraint `(f x)`.r:c. If `f x` does not meet the imposed constraint, then the part that is responsible cannot be `f a`, because the success or failure of `f a` is equal to that of `f x`. If then `b` meets the conditions, `f x` will succeed – meaning that the constraint on `f a` is true. Note what this means in the following function:

```
f x = f x
```

Any constraint with `f x` as the subject is always true. Regardless of what case selection is applied to `f x`, there will never be a crash. The reason for the lack of a case error may be entirely down to a non-termination error, but for the purposes of a a pattern match error checker, this is acceptable.

The `case` rule is the final one. For each case alternative, the generated condition says either the value being selected on is always of a different constructor (so this particular alternative is never executed), 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 never taken, it can ignore that alternative.

3.4.4 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 8

```
mapHead xs = case xs of
       Nil -> Nil
       Cons a b -> Cons (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:Cons`, 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:

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

For the `mapHead` function:

$P_0 =$ `#1:Cons`
$P_1 =$ `#1:Cons` $\wedge$ `#1.tail:Cons`
$P_2 =$ `#1:Cons` $\wedge$ `#1.tail:Cons` $\wedge$ `#1.tail.tail:Cons`

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

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

However, 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:Cons`. □

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 9

Consider the function `reverse` written using an accumulator:

```
reverse x = reverse2 x Nil

reverse2 x y = case x of
       Cons a b -> reverse2 b (Cons a y)
       Nil -> y
```

Argument `#1` follows the pattern `#1` $\hookleftarrow$ `#1.tail`, but we also have `#2` $\hookleftarrow$ `Cons #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:Cons`.

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` $\mathtt{x}_1 \cdots \mathtt{x}_n$ `#i`

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

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

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

`reverse2#1.*tail.head:Cons` $\wedge$ `reverse2#2.*tail.head:Cons`

□

3.5 Some Simple Examples

Now that the system has been specified, let's see how far it can take us. The following examples are all based on standard prelude functions. Most use the `List` data type, but could equally use any other data type.

Programs are written in the subset of Haskell accepted by the tool – with the exception of the syntactic list sugar (`:` and `[]`), which the tool does not support. For the examples in this paper, removal of list sugar was performed manually, and then specialization and other transformations were performed automatically by the checker. Outputs are shown with ASCII symbols converted to their mathematical equivalents, but results are essentially as given by the checker. The lines in the result are prefixed with `>`, the last line is the final result, and all lines before that correspond to stages of the checking process. In each case the check is run on the `main` function.

The examples are in order of increasing complexity.

Example 10

```
head x = case x of
           Cons a b -> a
main x = head x
> head#1:Cons
> False[main#1:Cons]
```

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

Example 11

```
list x = case x of
            (a:as) -> head x : tail x
            _ -> []
main x = list x
> head#1:Cons ∧ tail#1:Cons
> True
```

In this example `list` is about as useful as `id`, but it does show some nice features of the system. Even though internally `list` uses the non-exhaustive functions `head` and `tail`, the checker is able to tell that their arguments are not the empty list. This example shows the use of non-exhaustive components in a program which is proved to be free from pattern match errors. □

Example 12

```
main x = map head x
> head#1:Cons
> map_head#1.*tail.head:Cons
> False[main#1.*tail.head:Cons]
```

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 13

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

This result may at first seem surprising. The first disjunct of the condition says that applying `tail` any number of times to `main#1` (also known as `x`) the result must always be a `Cons`, in other words must be infinite. This guarantees case safety because `reverse` is tail strict, so if an infinite list is passed in, 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 where every element is a non-empty list. □

Example 14

```
main x = main2 (map box x)
main2 x = map head (reverse x)
box x = [x]
> head#1:Cons
> map_head#1.*tail.head:Cons
> main2#1.*tail:Cons ∨ main2#1.*tail.head:Cons
```

```
> True
```

The same constraint as for the previous example is generated. It is solved by mapping `box` over the list. The second condition evaluates to true, so the first becomes irrelevant. □

Example 15

```
main x = main2 (repeat x)
main2 x = map head (reverse x)
repeat x = x : repeat x
> head#1:Cons
> map_head#1.*tail.head:Cons
> main2#1.*tail:Cons ∨ main2#1.*tail.head:Cons
> True
```

Although the constraints in this example are identical to those above, they have been solved in a very different way. In this case the checker spots that `repeat` generates an infinite list, so the program will loop forever and not cause an error. □

Example 16

```
main x = tails x
tails x = foldr tails2 [[]] x
tails2 x y = (x:head y) : y
> head#1:Cons
> tails2#2:Cons
> fold_tails2#2.*tail.tail:Cons ∨ fold_tails2#1:Cons
> 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 `Cons`, 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 `*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 `Cons`. This is an argument for saying either the `foldr` does not call its function argument because it immediately takes the zero case, or the `foldr` recurses infinitely, and therefore the function is never called. Either way, because the initial argument to `foldr` is a `Cons`, and because `tails2` always returns a `Cons`, the second part of the condition can be satisfied. □

3.6 Properties of the system

This section explores the boundaries of the system, what the current checker can, and cannot, solve.

3.6.1 Correctness

What does it mean for the checker to be correct? If an application of `main` is evaluated and its argument constraints are satisfied, then the program will not fail with a pattern match error. One special case is that if the constraint is True, then the program is always safe.

The checker proceeds in steps. If it can be shown that the first step is correct, and that the transformation at each stage preserves correctness, then the checker is sound. This is an argument by induction.

The first stage is easy to show correct – the checker considers all non-exhaustive pattern matches, which are all the possible causes of a pattern match error, and the generated constraints are all that is required.

The fixed point stage is either successful in a finite number of iterations, or a safe approximation is generated. The approximations are done on a case by case basis, and each case can be checked for correctness individually. This process is also conservative: if all else fails the approximation is False, which is always safe.

At each stage, if the input constraint is correct, then the output constraint must be too. Since the program first finds all callers to functions with constraints on them, and moves the constraint on to their arguments, the constraint is preserved. The backward analysis proceeds case by case, so again it can be shown to be correct.

3.6.2 Unconditional Success

If all case expressions have exhaustive patterns, the checker determines the program is safe. This is entirely independent of the structure of the code – higher order, laziness, monads – all of these are entirely ignored by the checker. Many total programming approaches [Turner 04, Hughes 96b] demand exhaustive patterns from the outset, perhaps using this checker that restriction could be relaxed?

3.6.3 Laziness

Haskell is a lazy language. In this section we consider three functions. In a lazy language all three execute without error; in a strict language only one does. Currently the checker only discharges the first (which is strict). With modifications, the second could be solved. Significant work would be required to prove the final example.

It is important to remember that if a program is safe under strict evaluation, then it is also safe under lazy evaluation. Because of this, the checker remains sound.

Example 17

```
main x = case null x of
           True -> []
           False -> tail x
```

Checking this function, which can be thought of as a safe variant of `tail`, gives the result `True`. The constraint derived and its simplification are as follows:

```
> ((null x):False ∨ True) ∧
  ((null x):True ∨ x:Cons)
> (null x):True ∨ x:Cons
> x:Nil ∨ x:Cons
> x:(Nil,Cons)
> True
```

Either the `False` branch will not be taken because `(null x):True`, or it is safe, because `x:Cons`. By backward analysis `(null x):True` is equivalent to `x:Nil`. The two constraints can be collapsed into one, `x:(Nil,Cons)`. Because all constructors are covered, this constraint simplifies to `True`. □

Example 18

The next function to consider is equivalent, but defined in a different way.

```
main x = cond (null x) [] (tail x)

cond c t f = case c of
               True -> t
               False -> f
```

The checker generates the constraint `x:Cons`, because if that is not the case then `tail x` will fail. Unfortunately, the checker does not see under what conditions `f` is evaluated. This might be solved with an additional forward analysis pass, which tries to determine the situations in which an expression will be used. In this case inlining would also bring the condition to the surface, and allow it to be satisfied. □

Example 19

```
main x = head [x, head []]
```

Now `main` is an obfuscated version of `id`. The checker returns `False`. Evaluation of `head []` would cause the system to crash if evaluated, and there is no conditional guarding this expression, so the checker does not realize that it will not be evaluated. As `head` acts as a projection, discarding a portion of its input, we need a more sophisticated analysis. □

3.6.4 Higher Order Functions

The current system can only check higher order functions if they can be reduced to first order versions by a simple specialization procedure. This includes many commonly used higher order functions such as `map`, `filter`, `foldr` and `foldl`.

One place where specialization falls down is when functions are placed inside a data structure, such as a list.

Example 20

```
applyAll :: [a -> a] -> a -> a
applyAll [] x = x
applyAll (f:fs) x = f (applyAll fs x)
```

We cannot specialize `applyAll`. However, if all functions appearing in `applyAll`s arguments have exhaustive patterns, the checker succeeds. For example, if `applyAll [sin, cos] x` is the only application, the program passes. Because none of the named functions have non-exhaustive patterns, the system does not even attempt to analyze inside `applyAll`.

However, a non-exhaustive function in an `applyAll` argument does not necessarily cause the checker to fail. Consider the application `applyAll [(/ 3), sin, cos]`. In this case the function `(/)` is not exhaustive: if its second argument is 0, then the function crashes. But as the second argument is known to be 3, the system succeeds without any higher order analysis. □

A point-free programming style also poses problems. Take for instance the following function:

Example 21

```
odd :: (Integral a) => a -> Bool
odd = not . even
```

As this definition does not mention the argument, it is hard to assert things about it. This style of programming is very common, however it can typically be removed very easily. For example:

```
odd x = not (even x)
```

The current checker requires manual arity raising, but it can be automated in common cases without much effort. □

3.6.5 Fixed Pointing

The final major part of the system that limits what the checker can do is the fixed pointing. If a function makes no recursive calls, or if the predicate at the point when a recursive call is made is True, there is no problem.

More interesting to explore are functions where the checker tries but fails to find a fixed point.

Example 22

```
intersperse x y = case x of
       Nil -> y
       Cons a b -> Cons a (intersperse y b)

main x = map head (intersperse x x)
```

The checker gives the output `False [Could not fixed point intersperse]`. The reason is that in the recursive call it reaches the situation where `#1` $\hookleftarrow$ `#2` and `#2` $\hookleftarrow$ `#1.tail`, for which an approximation is not defined. It is possible to unfold the program one level to produce:

```
intersperse2 x y = case x of
  Nil -> y
  Cons x1 xs -> Cons x1
    (case y of
       Nil -> xs
       Cons y1 ys -> Cons y1 (intersperse2 xs ys)
     )

main x = map head (intersperse2 x x)
> main#1.*tail.head:Cons
```

In this new program `#1` $\hookleftarrow$ `#1.tail` and `#2` $\hookleftarrow$ `#2.tail`, and the checker does indeed obtain a fixed point.[1] □

Example 23

Another example where fixed pointing fails is derived from a function in Clausify [Runciman 93], modified to create a minimal failing case. A naive version is first presented, followed by the accumulating version which the checker rejects.

```
data Data = A Data Data | B | C

allB x = case x of
           A a1 a2 -> allB a1 ++ allB a2
           B -> [x]
```

[1] Although currently the tool does not give the expression shown above, it does give one which is equivalent, but longer (4 constraints, with accompanying predicates). This is purely a weakness in the predicate and regular expression simplifiers, and so the reduced version has been given, by performing manual simplification.

The application context requires `(allB x).*tail.head:B` and the generated condition is `x.*(A#1+A#2):(A,B)`, which is indeed correct. It may seem strange that the constraint is `:(A,B)`, but disallows `A` in the result. However, consider a value – if it is an `A` then because of `A#1+A#2` it would have both its inner elements followed. Eventually it must reach a `B`, or be a non-terminating sequence of `A`s – both of which are allowed by this constraint, and both of which are safe.

However, this version of `allB` is $O(n^2)$ in the length of the result list; by using an accumulator, an $O(n)$ version can be obtained.

```
allB x = allB2 x []

allB2 x y = case x of
             A a1 a2 -> allB2 a1 (allB2 a2 y)
             B -> x:y
```

Unfortunately, this definition has a recursive call such that `#2` $\hookleftarrow$ `allB2 #1.A#2 #2`, which cannot be approximated under the current fixed pointing scheme. □

Clearly the current fixed pointing system is a key limitation. One way to improve it would be to add a backward analysis step if the result is not immediately of a form that can be fixed pointed. This requires further work, but in our experience most recursive functions that cause fixed pointing problems can be refactored to avoid them.

3.7 Case Studies

Our goal is to check standard Haskell programs, and to provide useful feedback to the user. To test the checker against those objectives this section looks at several example Haskell programs. The programs were all written some time ago, for other purposes.

3.7.1 Adjoxo

The Adjoxo program is an adjudicator for a noughts and crosses game. The user supplies a file, formatted in a certain way, and the program parses this file and determines who has won a game, or if the game is a draw. There are only two non-exhaustive pattern matches (or instances where non-exhaustive prelude functions are called), and these are discussed separately.

One incomplete pattern match is the `opp` function, defined as:

```
opp :: Char -> Char
opp 'O' = 'X'
opp 'X' = 'O'
```

This function is used to report who has won. If a loss is detected then the message printed is that the opposite person has won. The generated constraint for `opp` is `opp#1:('O','X')`, as expected.

The `opp` function is called from `report`, with the player as second argument. It is only invoked when the first argument is `Loss`, so the generated condition is `report#1:(Win,Draw)` $\vee$ `report#2:('O','X')`. In all calls to `report`, the second argument is a literal `'X'` or `'O'`, so the constraint is satisfied.

A second source of pattern failure is a use of `foldr1`, with the constraint `foldr1#2:Cons`, in the following expression:

```
if gridFull ap pp then Draw
else foldr1 bestOf
    (map moveval (([1..9] `dif` ap) `dif` pp))
```

The `gridFull` and `dif` functions are defined by:

```
gridFull ap pp = length ap + length pp == 9

dif [] ys = []
dif xs [] = xs
dif xs@(x:xs') ys@(y:ys') =
    case compare x y of
        LT -> x : dif xs' ys
        EQ -> dif xs' ys'
        GT -> dif xs ys'
```

Backward analysis applied to the `map` application results in the condition `(([1..9] `dif` ap) `dif` pp):Cons`. Backwards analysis is then applied to `dif`, yielding the constraint that the result of `dif` is a `Cons`.

At this point the analysis starts to break down. Currently the function `compare` is modelled as returning any of `LT`, `GT` and `EQ`. Hence the initial constraint is `dif#1:Cons`, and the recursive calls give `#1` $\hookleftarrow$ `#1.tail`, `#1` $\hookleftarrow$ `#1` and `#2` $\hookleftarrow$ `#2.tail`. The checker concludes that `dif#1.*tail:Cons` – the first argument must be infinite. Without any knowledge of the values this is a reasonable deduction, but too much information has been lost for the analysis to complete successfully.

However, the program can be refactored so that it checks successfully. If the list passed to `foldr1` is empty, then the grid is full, and this check can be used directly instead of calling `gridFull`.

```
case (([1..9] `dif` ap) `dif` pp) of
    [] -> Draw
    xs -> foldr1 bestOf (map moveval xs)
```

Now `dif` is not even examined.

3.7.2 Soda

The Soda program solves word search problems. A grid of letters is given, along with words to look for, in any direction, including diagonals.

The program has three initial non-exhaustive patterns. Two are solved with relative ease. But the final one deserves a little more discussion.

The first and easiest non-exhaustive pattern is a use of `tail`, which can be satisfied directly by looking at the case structure.

The next instance of incomplete patterns is in the `diagonals` function:

```
diagonals [r] = map (:[]) (reverse r)
diagonals (r:rs) = zipinit (reverse r) ([]:diagonals rs)
```

The checker generates the constraint that `diagonals#1:Cons`. When a fixed point is attempted, because the argument in the recursive call of `diagonals` is already known to be a `Cons`, no additional conditions are generated. This constraint is propagated to applications of diagonals:

```
dr = diagonals grid
ur = diagonals (reverse grid)
```

These give the condition that `grid` is non-empty, and in the second call, that `grid` may be either non-empty or infinite – which can be simplified to just non-empty. A quick check of grid shows it to be a constant data structure, which is indeed non-empty.

The final, most complex instance of non-exhaustive patterns is:

```
zipinit [] ys = ys
zipinit (x:xs) (y:ys) = (x : y) : zipinit xs ys
```

This function, `zipinit`, fails if the first argument is a `Cons` and the second is a `Nil`. The checker obtains the fixed point `zipinit#1.*tail:Nil` $\vee$ `zipinit#2.*tail:Cons`. The first disjunct can be simplified to `zipinit#1:Nil`. The condition now reads: either the first argument is an empty list, or the second argument is an infinite list. Both alternatives are indeed safe, but both miss the point. The accurate condition is that the length of the first argument must be less than or equal to the length of the second argument. This condition cannot be derived in the current constraint framework.

Information is lost in the fixed pointing: the two recurrences to be fixed are `#1` $\hookleftarrow$ `#1.tail` and `#2` $\hookleftarrow$ `#2.tail`; these expand to `#1.*tail` and `#2.*tail`. A more accurate constraint would be `#1.tail`^{n} and `#2.tail`^{n}, where the n's are constrained to be the same length. Unfortunately this cannot be represented by a finite expression in the constraint language.

Following the accurate constraint backwards, it eventually becomes apparent that in `grid :: [[Char]]`, each element of the outer list must be a list of the same length. This condition took some substantial thought to derive, and is beyond the ability of the checker.

Once again, we can resort to a simple refactoring, adding an extra equation to the definition `zipinit`:

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

Now the program checks without any problems. An additional benefit is that if the grid is changed, say by deleting the two bottom right elements, the program does not crash. The original program fails, but with the modified version of `zipinit` the correct answer is given.

3.7.3 Clausify

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 [Partain 92], and was the focus of several papers on heap profiling [Runciman 93]. It parses logical propositions and puts them in clausal form.

We ignore the parser and jump straight to the transformation of propositions. The main pipeline is:

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

and the data structure is:

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

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 expressions, 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 `clause` definition.

```
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)
```

The checker generates large constraint formulae containing many expressions such as `(Dis#1+Dis#2)`. Directly motivated by this example, a special shorthand has been introduced to output `C#?` to mean the `(C#1+C#2)`. This translation is only performed during output function – internally the conditions remain the same.

For the last five stages of clausify the checker generates the following constraints:

```
> clause'#1.*Dis#?:(Dis,Sym,Not) ∧ clause'#1.*Dis#?.Not#1:Sym

> clause#1.*Dis#?:(Dis,Sym,Not) ∧ clause#1.*Dis#?.Not#1:Sym

> unicl'#1.*Dis#?:(Dis,Sym,Not) ∧ unicl'#1.*Dis#?.Not#1:Sym

> foldr_unicl#2.*tail.head.*Dis#?:(Dis,Sym,Not) ∧
  foldr_unicl#2.*tail.head.*Dis#?.Not#1:Sym

> unicl#1.*tail.head.*Dis#?:(Dis,Sym,Not) ∧ unicl#1.*tail.head.*Dis#?.Not#1:Sym
```

These constraints give accurate and precise requirements for a case error not to occur at each stage, and are very useful. When the condition is propagated back over the `split` function, the result becomes less pleasing. An error occurs in fixed pointing, for reasons discussed in §3.6.5. 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 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 condition:

```
>
(
    split#1.*Con#?:(Con,Dis,Sym,Not) ∧
    split#1.*Con#?.Dis#?.*Dis#?:(Dis,Sym,Not) ∧
    split#1.*Con#?.*Dis#?.Not#1:Sym
)
```

These constraints 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 `Dis#?.*Dis#?` part of the regular expression in the 2nd conjunct. A natural constraint might be `*Con#?.*Dis#?:(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, 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 [Runciman 93] 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.

3.7.4 Execution Times

The checker is still a prototype – no profiling has been performed, and neither has much thought been given to efficiency. Even so, checking all the short examples is virtually instant. On a Pentium 4 2GHz machine, with 1Gb of RAM, running Linux, and compiling the checker with GHC 5.04.2, the elapsed times for the examples in §3.5 (all in seconds):

0.018, 0.018, 0.021, 0.021, 0.022, 0.021, 0.020

The motivating example given in the introduction, `risers`, took 0.007 seconds. Adjoxo took 0.037 seconds. Soda took 0.032 seconds. Clausify took 0.047 seconds up until after the constraint for `split` had been determined. When run until after `disin` 126Kb of output is generated in 3.192 seconds.

3.8 Conclusions

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 can execute and not raise a pattern match error. A range of small examples has been investigated successfully, along with several larger examples. Where programs cannot be checked directly, various refactoring steps have been shown that can increase the checker's success rate.

The checker does not take account of laziness in its analysis, but we suspect that this is not a severe limitation. While counterexamples can be constructed, they are not of a very natural style.

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, and very little concern is given to types. It does not handle classes, but these can be transformed away without vast complication.

Chapter 4

Proposal

4.1 Motivation

In order to set into context the work I intend to do, it is useful to set out the overall goals. My primary aim is to create a static checker for Haskell 98. If the checker reports that a program is safe, this will guarantee that particular program is unfailing. This primarily means that the program will not fail with a pattern match error, and that it will terminate.

My hope is that with future work this checker could become a standard part of a Haskell toolkit – including compilers, debuggers and static checkers.

4.2 Common Infrastructure

I am hoping to build two checking mechanisms – a pattern match checker and a termination checker. While these do different tasks, I hope that underneath they will share a common infrastructure.

The following goals are both short term and long term – to enable the checking to proceed, a short term version of most of these elements is needed. To increase the quality of the checkers it will then be necessary to investigate each element individually, and improve them.

Reduction to a Haskell core The full Haskell language is too big to analyze, so a reduced one will be designed, along with a convertor from Haskell 98. This work has already been started upon in §3.2, but needs further development.

Program Transformation There are many program transformations that can be applied to ease the analysis. Some of these include specialization, lambda lifting, let floating and inlining of functions. It is likely that other transformations could also be useful.

Regular Expression Simplification Various problems covered in the results chapter require a firm understanding of regular expressions. Some work has already been done in the area of regular expression simplification [Mitchell 03], but more work is required. Ideally properties such as regular expression equality should be derivable, without generating finite state machines.

Predicate Simplification The predicates managed by the systems to be developed will become large. As such, as robust predicate simplifier is required, which can combine simplifications on atomic terms with those using standard boolean logic identities.

4.3 Pattern Match Checker

An initial pattern match checker has been written, and is detailed in the results section. This tool is at an early stage of development, but is a good base to build upon.

The following tasks follow on from this work.

Diagnosis 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 any individual step had.

Proof The checker has not been formally proved with respect to any semantics. This is required to give confidence that the interpretation given to a program by the checker matches that of the compiler/interpretter.

Theory The central algorithms of the checker can be made more powerful. The current framework has a lot of untapped power. A forward analysis pass could make the system more lazy. A better fixed pointing procedure could perhaps make additional use of backward analysis.

Safety Annotations I hope to generate automatically information about the prelude functions, and under what circumstances they are safe. This would make a useful reference.

User Assertions Currently the system derives constraints based upon non-exhaustive case statements. In some circumstances the programmer has stronger requirements in mind. By accepting user annotations, the program could then check these elements. This could help with refactoring, by narrowing down the scope of an error.

4.4 Termination Checking

The initial research has pushed in the direction of pattern match errors, and termination has been deliberately ignored. I suspect that the same framework can be used in the termination checking. Implementing a simple termination checker that accepts only primitive recursion, and ignores laziness entirely, should be an easily achievable first step.

There are lots of methods for termination checking, in a variety of different disciplines including term rewriting, graph rewriting and logic programming. I will continue to study the existing literature, before devising an strategy that seems most appropriate to a lazy language.

4.5 Timeline

I intend to work on the common infrastructure next, in the order of regular expressions then full Haskell reduction – with predicate simplification being investigated as required by other aspects. I expect this to take about three months.

After the infrastructure has been completed, I will revisit the static pattern match checker, seeing what additional power the new base adds. Hopefully the effort spent at lower levels will allow some improvements to be obtained. This will take approximately one month.

After this, I will move on to the termination checker. The first task is to examine the theory in a great level of detail, then writing a termination checker. The remaining five months will be spent on this task.

At the end of nine months, I hope to have pattern match checker and a termination checker, both for a reasonable subset of Haskell.

Bibliography

[Abel 98] Andreas Abel. `foetus` – Termination Checker for Simple Functional Programs. Programming Lab Report, Ludwig-Maximilians, München Universität, July 1998.

[Aiken 91] Alex Aiken & 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.

[Aiken 94] Alexander Aiken, Edward L. Wimmers & T. K. Lakshman. *Soft Typing with Conditional Types*. In POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 163–173. ACM Press, 1994.

[Ame 89] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, Dec 1989.

[Antoy 00] Sergio Antoy & Michael Hanus. *Compiling Multi–Paradigm Declarative Programs into Prolog*. In Frontiers of Combining Systems, pages 171–185. Springer LNCS 1794, 2000.

[Armstrong 93] J. L. Armstrong, Mike Williams, Robert Virding & Claes Wilkström. ERLANG for Concurrent Programming. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[Augustsson 85] Lennart Augustsson. *Compiling Pattern Matching*. In Proc. of A Conference on Functional Programming Languages and Computer Architecture, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Backus 90] John Backus, John H. Williams & Edward L. Wimmers. An Introduction to the Programming Language FL. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Brauburger 98] Jürgen Brauburger & Jürgen Giesl. *Termination Analysis by Inductive Evaluation*. In CADE–15: Proceedings of the 15th International Conference on Automated Deduction, pages 254–269. Springer–Verlag, 1998.

[Bray 04] Tim Bray. Extensible Markup Language (XML) 1.0 (Third Edition). `http://www.w3.org/TR/2004/REC-xml-20040204/`, February 2004.

[Chitil 03] Olaf Chitil, Dan McNeill & Colin Runciman. *Lazy Assertions*. In Draft Proceedings of the 15th International Workshop on Implementation of Functional Languages, IFL 2003, pages 31–46, Edinburgh, Scotland, September 2003.

[Claessen 00] Koen Claessen & John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00), pages 268–279. ACM Press, 2000.

[Clark 99] James Clark. XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`, November 1999.

[Conway 71] John Horton Conway. Regular Algebra and Finite Machines. London Chapman and Hall, 1971.

[D'Agostino 99] Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle & Joachim Posegga, editors. The Handbook of Tableau Methods. Kluwer Academic Publishers, 1999.

[Danielsson 04] Nils Anders Danielsson & Patrik Jansson. *Chasing Bottoms; A Case Study in Program Verification in the Presence of Partial and Infinite Values*. In Proc. 7th International Conference on Mathematics of Program Construction. Springer LNCS 3125, 2004.

[Daumé III 03] Hal Daumé III. *Haskell All-In-One*. `http://www.isi.edu/~hdaume/HAllInOne/`, July 2003.

[Giesl 01] Jürgen Giesl & Thomas Arts. *Verification of Erlang Processes by Dependency Pairs*. Applicable Algebra in Engineering, Communication and Computing, vol. 12, no. 1/2, pages 39–72, 2001.

[Glenstrup 99] Arne John Glenstrup. Terminatinator II: Stopping Partial Evaluation of Fully Recursive Programs. Master's thesis, Department of Computer Science, University of Copenhagen, June 1999.

[Hughes 96a] John Hughes. *Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference*. In Olivier Danvy, Robert Glück & Peter Thiemann, editors, Partial Evaluation, pages 183–215. Springer LNCS 1110, February 1996.

[Hughes 96b] John Hughes, Lars Pareto & Amr Sabry. *Proving the Correctness of Reactive Systems Using Sized Types*. In POPL '96: Proceedings of the 23rd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 410–423. ACM Press, 1996.

[ISO 95] ISO/IEC 13211-1. *Information Technology – Programming Languages – Prolog – Part 1: General Core 1995*, 1995.

[Maranget 05] Luc Maranget. Warnings for Pattern Matching. Under consideration for publication in *Journal of Functional Programming*, March 2005.

[McAllester 96] David A. McAllester & Kostas Arkoudas. *Walther Recursion*. In CADE–13: Proceedings of the 13th International Conference on Automated Deduction, pages 643–657. Springer–Verlag, 1996.

[Mitchell 03] Neil Mitchell. Regular Expression Simplification. Undergraduate project, Computer Science Department, University of York, 2003.

[Mitchell 04] Neil Mitchell. Static Analysis of Pasta. Undergraduate project, Computer Science Department, University of York, 2004.

[Mitchell 05] Neil Mitchell & Colin Runciman. Unfailing Haskell, Part 1: A Static Checker for Pattern Matching. Submitted to the Haskell Workshop 2005, June 2005.

[Naish 96] Lee Naish. *Higher–order logic programming in Prolog*. In Workshop on Multi–Paradigm Logic Programming, 1996.

[Nielson 99] Flemming Nielson, Hanne Riis Nielson & Chris Hankin. Principles of Program Analysis. Springer, 1999.

[Panitz 96] Sven Eric Panitz. *Termination Proofs for a Lazy Functional Language by Abstract Reduction*. Technical report 06, Fachbereich Informatik, J.W. Goethe–Universität Frankfurt am Main, June 1996.

[Panitz 97] Sven Eric Panitz & Manfred Schmidt-Schauß. *TEA: Automatically Proving Termination of Programs in a Non–strict Higher–Order Functional Language*. In SAS '97: Proceedings of the 4th International Symposium on Static Analysis, pages 345–360. Springer–Verlag, 1997.

[Pareto 98] Lars Pareto. *Sized Types*. PhD thesis, Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, Sweden, 1998.

[Partain 92] Will Partain. *The `nofib` Benchmark Suite of Haskell Programs*. In J Launchbury & PM Sansom, editors, Functional Programming, Glasgow 1992, pages 195–202. Springer-Verlag Workshops in Computing, 1992.

[Peyton Jones 03] Simon Peyton Jones. Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, first edition, 2003.

[Pientka 01] Brigitte Pientka. *Termination and Reduction Checking for Higher–Order Logic Programs*. In IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning, pages 401–415. Springer–Verlag, 2001.

[Runciman 89] Colin Runciman. *What About the Natural Numbers?* Computer Languages, vol. 14, no. 3, pages 181–191, 1989.

[Runciman 93] Colin Runciman & David Wakeling. *Heap Profiling of Lazy Functional Programs*. Journal of Functional Programming, vol. 3, no. 2, pages 217–245, 1993.

[Sestoft 96] P. Sestoft. *ML Pattern Match Compilation and Partial Evaluation*. In O. Danvy, R. Glück & P. Thiemann, editors, Partial Evaluation, pages 446–464. Springer LNCS 1110, 1996.

[Spivey 90] M. Spivey. *A Functional Theory of Exceptions*. Science of Computer Programming, vol. 14, no. 1, pages 25–42, 1990.

[Steele 90] Guy L. Steele. Common Lisp the Language. Digital Press, second edition, 1990.

[Telford 97] Alastair Telford & David Turner. *Ensuring the Productivity of Infinite Structures*. Technical report TR 14–97, The Computing Laboratory, University of Kent at Canterbury, September 1997.

[Telford 00] Alastair Telford & David Turner. *A Hierarchy of Languages with Strong Termination Properties*. Technical report TR 2–00, The Computing Laboratory, University of Kent at Canterbury, February 2000.

[The GHC Team 05] 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.

[Thiemann 03] René Thiemann & Jürgen Giesl. *Size–change Termination for Term Rewriting*. In R. Nieuwenhuis, editor, Proc. 14th Int. Conf. Rewriting Techniques and Applications (RTA'2003), LNCS 2706, pages 264–278. Springer–Verlag, June 2003.

[Tolmach 01] Andrew Tolmach. *An External Representation for the GHC Core Language*. `http://www.haskell.org/ghc/documentation.html`, September 2001.

[Tozawa 01] 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.

[Turing 37] Alan Mathinson Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, vol. 42, pages 230–265, 1937.

[Turner 04] David Turner. *Total Functional Programming*. Journal of Universal Computer Science, vol. 10, no. 7, pages 751–768, July 2004.

[Wadler 85] Philip Wadler. *How To Replace Failure By a List of Successes*. In Proc. of A Conference on Functional Programming Languages and Computer Architecture, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York.

[Wadler 86] Philip Wadler. The Implementation of Functional Programming Languages, chapter 6. Prentice-Hall International, May 1986.