Thesis Proposal

Ruud Koot

March 30, 2012

Introduction

This document is a proposal for a thesis on answering the research question:

> Can we use a type and effect system in combination with refinement types to develop a pattern-match analysis for a non-strict higher-order functional language that is both performant and precise enough to be of practical use?

In Chapter 1 we present a number of examples to demonstrate why this is an interesting problem. In Chapter 2 we give a short introduction to the relevant concepts of the research question: higher-order functional languages, type and effect systems and refinement types. In Chapter 3 we present work related to our research question and discuss which aspects of that work are relevant to or different from our proposed system. In Chapter 4, 5 and 6 we will discuss our envisioned approach to answering the research question, the set of deliverables and how we will evaluate whether the research question has been answered successfully or not.

Chapter 1

Motivation

In 1978, Robin Milner [7] famously wrote (and proved) that:

> Well-typed programs cannot "go wrong".

This observation is sometimes optimistically overstated as "if your Haskell program type checks and compiles it will work." Even if we ignore logic errors, such a strong interpretation of Milner's statement does not hold, as anyone who has seen the dreaded:

```
*** Exception: PM.hs:1:1-13: Non-exhaustive patterns in function f
```

can attest to. To correctly interpret Milner we need to distinguish between three kinds of "wrongness":

Getting stuck If a program tries to evaluate a non-sensical expression – such as `3 + true` – it cannot possibly make any further progress and is said to be "stuck". This is the "go wrong" Milner referred to. He proved that a sound type system can statically guarantee such expressions will never occur or need to be evaluated at run-time.

Diverging If we have a function definition `f = f`, the evaluation of `f` $\rightsquigarrow$ `f` $\rightsquigarrow$ ... will fail to terminate. We might be making progress in a technical sense, but it will have no useful observable result.

Undefinedness Another source of "wrongness" are partial functions: functions which are not defined on all the elements of their domain. Prime examples are case-statements with missing constructors and functions defined by pattern matching but which do not cover all possible patterns.

Unlike Milner, who spoke about the first kind of wrongness and the work on termination checking, which concerns itself with the second, we shall focus on the third: a pattern-match analysis which determines if functions are only invoked on values for which they are defined.

1.1 Examples

Partial functions Haskell programmers often work with partial functions, the most common one possibly being `head`:

```
1  main = let xs = if length "foo" > 5 then [1,2,3] else []
2          in head xs
3
4  head (x : xs) = x
```

This program is guaranteed to crash at run-time, so we would like to be warned beforehand by the compiler or a tool:

```
On line 2 you applied the function "head" to the empty list "xs".
The function "head" expects a non-empty list as its first argument.
```

If the guard of the if-statement had read `length "foo" < 5` the program would have run without crashing and we would like the compiler or tool not to warn us spuriously. In case it is not possible to determine statically whether or not a program will crash, a warning should still be raised.

Compiler construction Compilers work with large and complex data types to represent the abstract syntax tree. These data structures must be able to represent all syntactic constructs the parser is able to recognize. This results in an abstract syntax tree that is unnecessarily complex and too cumbersome for the later stages of the compiler – such as the optimizer – to work with. This problem is resolved by *desugaring* the original abstract syntax tree into a simpler – but semantically equivalent – abstract syntax tree than does not use all of the constructors available in the original abstract syntax tree.

The compiler writer now has a choice between two different options: either write a desugaring stage `desugar :: ComplexAST -> SimpleAST` – duplicating most of the data type representing and functions operating on the abstract syntax tree – or take the easy route `desugar :: AST -> AST` and assume certain constructors will no longer be present in the abstract syntax tree at stages of the compiler executed after the desugaring step. The former has all the usual downsides of code duplication – such as having to manually keep the two or more data types synchronized – while that latter forgoes many of the advantages of strong typing and type safety: if the compiler pipeline is restructured and one of the stages that was originally assumed to run only after the desugaring suddenly runs before that point the error might only be detected at run-time by a pattern match failure. A pattern match analysis should be able to detect such errors statically.

Maintaining invariants Many algorithms and data structures maintain invariants that cannot be easily encoded into their type. These invariants often ensure that certain incomplete case-statements are guaranteed not to cause a pattern match failure. An example is the `risers` function from [9], calculating monotonically increasing segments of a list (e.g., `risers [1,3,5,1,2]` $\rightsquigarrow$ `[[1,3,5],[1,2]]`):

```
risers :: Ord a => [a] -> [[a]]
risers []         = []
risers [x]        = [[x]]
risers (x1:x2:xs) = let (s:ss) = risers (x2:xs)
                     in if x <= y then (x:s):ss else [x]:(s:ss)
```

The let-binding in the third alternative of `risers` expects the recursive call to return a non-empty list. A naive analysis might raise a warning here. If we think a bit longer, however, we see that we also pass the recursive call to `risers` a non-empty list. This means we will end up in either the second or third alternative in the recursive call. Both the second alternative and both branches of the if-statement in the third alternative result in a non-empty list, satisfying the assumption we made earlier.

Another example might be a collection of mathematical operations working on bitstrings (integers encoded as lists of binary digits without leading zeroes):

```
type Bitstring = [Int]

add :: Bitstring -> Bitstring -> Bitstring
add [] y = y
add x [] = x
add (0:x) (0:y) = 0 : add x y
add (0:x) (1:y) = 1 : add x y
add (1:x) (0:y) = 1 : add x y
add (1:x) (1:y) = 0 : add (add [1] x) y
```

The patterns in `add` are far from complete, but maintain the invariant if passed arguments that satisfy the invariant. So if we are careful to only pass valid bitstrings into a complex mathematical expression of bitstring-operations it will result in a valid bitstring without crashing due to a pattern match failure.

Chapter 2

Context

2.1 Haskell

Haskell is statically-typed functional programming language with non-strict evaluation semantics [14]. As a functional language it has first-class and higher-order functions and features a rich type system supporting parametric polymorphism and type classes. Programmers can define custom types in the form of algebraic data types and write functions over them using pattern matching.

Like most functional languages, Haskell can be easily translated into a typed λ-calculus (System F_C). From the point-of-view of the programmer it offers a wealth of syntactic sugar over a plain λ-calculus – such as guards and list comprehensions – allowing programs to be expressed concisely and in a readable fashion.

The following example demonstrates the the syntax of Haskell and some features mentioned previously:

```
1  data Tree a = Branch (Tree a) (Tree a)
2              | Leaf a
3
4  mapTree :: (a -> b) -> Tree a -> Tree b
5  mapTree f (Branch t1 t2) = Branch (mapTree f t1) (mapTree f t2)
6  mapTree f (Leaf a)       = Leaf (f a)
7
8  instance Functor Tree where
9      fmap = mapTree
```

Lines 1–2 define an algebraic data type (ADT) representing a tree. A tree can be constructed by either a `Branch` containing a left and a right subtree or by a `Leaf` containing a value. The ADT is parameterized by a type `a` giving the type of the value stored in the `Leaf`.

Lines 4–5 define a higher-order function `mapTree` which applies a given function `f :: a -> b` to all the values stored in all the leaves of a given `Tree` resulting in a new `Tree`. As the domain and the range of `f` need not coincided the type parameter of the tree changes as well. The body of the function is defined by pattern matching over the constructors of `Tree` and performs recursion in the case of a `Branch` and applies the argument `f` to the field of type `a` in `Leaf` otherwise.

Lines 8–9 declare the type constructor `Tree` to be an instance of the type class `Functor`, expressing it to be an endofunctor in the category of data types with its mapping on morphisms given by `mapTree`.

2.1.1 Pattern matching

Particularly relevant to our analysis are Haskell's pattern matching abilities. We demonstrate some of them in the following function:

```
rotate (Branch b@(Branch (Leaf x) (Leaf y)) (Leaf z))
           | z < x     = Branch (Leaf z) b
           | z < y     = Branch (Branch (Leaf x) (Leaf z)) (Leaf y)
           | otherwise = undefined
```

The function will rotate a tree `(Branch (Branch (Leaf 2) (Leaf 3)) (Leaf 1))` into `Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))`. It accomplishes this through the use of *nested pattern matching*, *as-patterns* and *guards.*

This function only works on trees of a very specific shape. Feeding the function a tree of a different shape will cause a pattern match failure. Similarly, an error will be raised if the function reaches the `otherwise` branch, containing an `undefined` value. This would also happen if the branch had called `error` or the branch and its guard were left out entirely.

2.2 Type and effect systems

Type and effect systems are approaches to program analysis suitable for typed languages [10]. In this formalism we take the the *underlying type system* of the language and (conservatively) extend it by adding *annotations* to the types (base, function and other) of the system.

2.2.1 Example

As an example we give an analysis which determines, in the form of an annotation, which values an expression can evaluate to. The language is the simply-typed λ-calculus with let-bindings and an if-statement. The underlying type systems is standard and given in Figure 2.1.

The annotated type system is given in Figure . It does little more than collecting all constants in the annotations. The only interesting rule is for the if-statement. We need to take the union over the values collected in both branches of the if-statement. We could have chosen to make the rule depend on the annotation of the guard expression, passing on only the values collected in the then-branch along if we knew it could only be `True` or only the values collected in the else-branch if we knew it could only be `False` from its annotation. This would make the analysis more precise.

We have also added an inference rule for *subeffecting.* This allows us to make the annotation less precise. This is sometimes necessary when applying a value about which we have very accurate information to a function which requires a more general argument.

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} c : \tau_c}\ [\mathrm{con}] \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathrm{UL}} x : \tau}\ [\mathrm{var}]$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash_{\mathrm{UL}} e : \tau_2}{\Gamma \vdash_{\mathrm{UL}} \lambda x.e : \tau_1 \to \tau_2}\ [\mathrm{abs}]$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash_{\mathrm{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} e_1\ e_2 : \tau_1}\ [\mathrm{app}]$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash_{\mathrm{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \tau_2}\ [\mathrm{let}]$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_1 : \mathrm{bool} \quad \Gamma \vdash_{\mathrm{UL}} e_2 : \tau \quad \Gamma \vdash_{\mathrm{UL}} e_2 : \tau}{\Gamma \vdash \mathrm{if}\ e_0\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2 : \tau}\ [\mathrm{if}]$$

Figure 2.1: Underlying type system

$$\frac{}{\Gamma \vdash_{\mathrm{AN}} n : \mathrm{int}^{\{n\}}}\ [\mathrm{con}] \qquad \frac{\Gamma(x) = \widehat{\tau}}{\Gamma \vdash_{\mathrm{AN}} x : \widehat{\tau}}\ [\mathrm{var}]$$

$$\frac{\Gamma[x \mapsto \widehat{\tau}_1] \vdash_{\mathrm{AN}} e : \widehat{\tau}_2}{\Gamma \vdash_{\mathrm{AN}} \lambda x.e : \widehat{\tau}_1 \to \widehat{\tau}_2}\ [\mathrm{abs}]$$

$$\frac{\Gamma \vdash_{\mathrm{AN}} e_1 : \widehat{\tau}_2 \to \widehat{\tau}_1 \quad \Gamma \vdash_{\mathrm{AN}} e_2 : \widehat{\tau}_2}{\Gamma \vdash_{\mathrm{AN}} e_1\ e_2 : \widehat{\tau}_1}\ [\mathrm{app}]$$

$$\frac{\Gamma \vdash_{\mathrm{AN}} e_1 : \widehat{\tau}_1 \quad \Gamma[x \mapsto \widehat{\tau}_1] \vdash_{\mathrm{AN}} e_2 : \widehat{\tau}_2}{\Gamma \vdash_{\mathrm{AN}} \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : \widehat{\tau}_2}\ [\mathrm{let}]$$

$$\frac{\Gamma \vdash_{\mathrm{AN}} e_1 : \mathrm{bool} \quad \Gamma \vdash_{\mathrm{AN}} e_2 : \tau^{\varphi_2} \quad \Gamma \vdash_{\mathrm{AN}} e_2 : \tau^{\varphi_3}}{\Gamma \vdash_{\mathrm{AN}} \mathrm{if}\ e_0\ \mathrm{then}\ e_1\ \mathrm{else}\ e_2 : \tau^{\varphi_2 \cup \varphi_3}}\ [\mathrm{if}]$$

$$\frac{\Gamma \vdash_{\mathrm{AN}} e : \tau^{\varphi}}{\Gamma \vdash_{\mathrm{AN}} e : \tau^{\varphi \cup \varphi'}}\ [\mathrm{sub}]$$

Figure 2.2: Annotated type system

2.2.2 Type inference

An analysis specified as an inference system does not allow us to compute the result of the analysis directly, as the premises of several of the inference rules usually require us to guess the correct type annotations for the type and annotation variables (e.g., in the presence of subeffecting). To overcome this problem we need a *type reconstruction algorithm.*

If the underlying type system is Hindley–Milner, then Algorithm W can be used to infer the types. Algorithm W works bottom-up and can compute types in a single pass by unifying type variables. This may not always be possible for the annotations in an annotated type system, however. Here we can use a variant of Algorithm W to gather a set of constraints on the annotation variables and solve these constraints with a worklist algorithm in a second phase.

2.2.3 Polyvariance

Consider the following program:

```
f = let id x = x
     in (id 1, id 2)
```

When performing the analysis given above we would assign the type $\texttt{int} \rightarrow \texttt{int}^{\{1,2\}}$ to `id` and as a result the type $\texttt{int}^{\{1,2\}} \times \texttt{int}^{\{1,2\}}$ to `f`. Clearly, this result is not optimal. This loss in precision has been caused by the two separate calls to `id` *poisoning* each other.

A similar situation occurs at the type-level in the function:

```
f = let id x = x
     in (id 1, id true)
```

Hindley–Milner manages to avoid problems in this situation by a mechanism called *let-generalization.* The binding `id` has the type $\alpha \rightarrow \alpha$. Instead of trying – and failing – to unify α with both `int` and `bool` in the body of the let-binding, the type of `id` is generalized to the polymorphic type $\forall \alpha . \alpha \rightarrow \alpha$. In the body of the let-binding, `id` can be instantiated twice: once to $\texttt{int} \rightarrow \texttt{int}$ and once to $\texttt{bool} \rightarrow \texttt{bool}$.

A similar trick can be used for the annotated type system, except that instead of quantifying over type variables, we quantify over annotation variables.

2.3 Refinement types

Refinement types are a form of subtyping that allow us to state more accurately what values a variable of a particular type can contain. For example, a refinement type { `a :: Int | a < 0` } states that `a` is a variable of type `Int` than can only contain negative numbers.

Freeman and Pfenning [3, 2] give a formal development of refinement types for the ML programming language that can support recursive types.

Their key contribution is allowing the programmer to define `rectypes`. Besides defining a recursive type for lists:

```
data List a = Nil | Cons a (List a)
```

$$\begin{tikzcd}
& \texttt{List a} \arrow[d, dash] & \\
& \texttt{Singleton a} \vee \texttt{Nil a} \arrow[dl, dash] \arrow[dr, dash] & \\
\texttt{Singleton a} \arrow[dr, dash] & & \texttt{Nil a} \arrow[dl, dash] \\
& \bot &
\end{tikzcd}$$

Figure 2.3: A lattice of `List` and its `rectype`s.

The programmer can also define a `rectype` describing singleton lists:

```
rectype Singleton a = Cons a Nil
```

Together with an automatically derived `rectype` for the non-recurisive constructor `Nil` (and a union type contructor) the compiler can construct – using known algorithms on regular tree grammars [4] – a finite lattice of refinement types for `List a` given in Figure 2.3.

2.3.1 Union and Intersection Types

Refinement types are constructed from regular types, including function types, as well as union and intersection types.

A union type such as `Int ∨ Bool`, `List a ∨ Singleton a` or `Int -> Int ∨ Bool -> Bool` state a value can be either of the type on the left or of the type on the right, but we do not know which. In the case of `List a ∨ Singleton a` we are able to simplify this type to `List a`, as `Singleton a` is a subtype of `List a`.

An intersection type such as `Int ∧ Bool`, `List a ∧ Singleton a` or `Int -> Int ∧ Bool -> Bool` state the a value has both the type on the left and the type on the right. Compared to union types these are more interesting. The type `List a ∧ Singleton a` can still be simplified, but now to `Singleton a`. A value cannot be of both type `Int` and `Bool`, so we can simply the type of such a value to the bottom or empty type. There do exist functions that are of both type `Int -> Int` and `Bool -> Bool`, for example `id`. In fact, intersection types can in the limit be viewed as a form of parametric polymorphism.

2.3.2 Constructors and Pattern Matching

To infer refinement types more accurate types need to be given to constructors. This is done by using a restricted[1] form of intersection types. For example, the `Cons` constructor is given the type:

```
Cons :: a -> Nil       a -> Singleton a
     /\ a -> Singleton a -> List      a
     /\ a -> List      a -> List      a
```

[1]We only take intersections of subtypes of the same data type.

This type can be derived automatically from the `rectype` declarations. Types of functions can also be inferred automatically, although there may be a loss of precision when higher-order functions or polymorphism are involved.

Type inference for intersection types is undecidable in general [12, 13]. Because the lattice of types is finite the algorithm is effectively able to do an exhaustive search over all possible types, however. Higher-order functions can cause the size of the type to blow up exponentially, each pairing of their range and domain needs to be included in the intersection type.

A case-statement

```
case G of
  Nil       -> E1
  Cons a as -> E2
```

can be seen as a call to a higher-order function

```
case_List G E1 (\a as -> E2)
```

where `case_list` has the refinement type

```
case_List :: forall a. forallR (r1 :: a). forallR (r2 :: a).
       Nil a       -> r1 -> (List a -> r2) -> r1
    /\ Singleton a -> r1 -> (List a -> r2) -> r2
    /\ List a      -> r1 -> (List a -> r2) -> (r1 \/ r2)
```

Chapter 3

Related Work

3.1 Neil Mitchell's *Catch*

Neil Mitchel's *Catch* ("CAse Totality CHecker") [8] is a tool specialized in finding potential pattern match failures in Haskell programs.

3.1.1 Overview

Catch works by calculates preconditions on functions. The preconditions are given in a constraint language. By varying the constraint language trade-offs between performance and precision can be made. If the calculated precondition for `main` is True, the program does not crash.

Preconditions are calculated iteratively. Precondition start out as `True`, except for `error`, whose precondition is `False`.[1]

Example Following the example given by Mitchell, given the function:

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

The computed precondition will be:

$$\text{Precondition}(\texttt{null xs}) \wedge (\texttt{null xs} \in \{\texttt{True}\} \vee \text{Precondition}(\texttt{tail xs})),$$

stating the necessary precondition for `null xs` not to crash mush be fulfilled and either `null xs` must evaluate to `True` or the necessary precondition for `tail xs` not to crash must be fulfilled.

As `null xs` cannot crash we find its precondition to be simply True and we can deduce the precondition for `tail xs` to be $\texttt{xs} \in \{(:)\}$. Substituting these into our precondition we find

$$\text{True} \wedge (\texttt{null xs} \in \{\texttt{True}\} \vee \texttt{xs} \in \{(:)\}),$$

which simplifies to

$$\texttt{null xs} \in \{\texttt{True}\} \vee \texttt{xs} \in \{(:)\}.$$

[1]Note that `undefined` is defined in terms of `error`.

The subexpression $\texttt{null xs} \in \{\texttt{True}\}$ forms a postcondition on `null xs`. From this postcondition Catch computes the required precondition that satisfies it, which in this particular case should turn out to be $\texttt{xs} \in \{[]\}$. Substituting, we find

$$\texttt{xs} \in \{[]\} \vee \texttt{xs} \in \{(:)\},$$

which, using our knowledge of the list data type, turns out the be a tautology, i.e. the (trivial) precondition for `safeTail` is True.

3.1.2 Constraint systems

The example in the previous section used a simple constraint language, specifying to which set of head constructors an expression should evaluate. Mitchell developed two more expressive constraint systems: regular expression constraints and multipattern constraints.

Regular expression constraints Constraints are formed by a regular expression over an alphabet of a number of ad-hoc *selectors* (e.g. `hd` and `tl` for lists.) A precondition for `map head xs` using regular expression constraints reads $\texttt{xs} \in (\texttt{tl}^* \cdot \texttt{hd} \rightsquigarrow \{(:)\})$. It should be interpreted as "after applying zero or more `tl`s to `xs` and then applying a `hd` we should find a `(:)` constructor."

Accoriding to Mitchell regular expression constraints tend to scale badly as the program increase in size, although he could not identify more specific condition under which this problem manifests.[2]

Multipattern constraints Multipatterns are of the form $\alpha \star \rho$, where α gives the set of constructors that are valid at the head of the value, while ρ gives the set of constructors that can appear at the recursive positions (if any) of the constructor at the head of the value. The elements of these sets can again be multipatterns. To specify that `xs` in `map head xs` should be a list of non-empty lists, we use the multipattern:

$$\begin{array}{c} \{[], (:)(\{(:)\ \texttt{Any}\} \star \{[], (:)\ \texttt{Any}\})\} \\ \star \\ \{[], (:)(\{(:)\ \texttt{Any}\} \star \{[], (:)\ \texttt{Any}\})\} \end{array}$$

`Any` is a wildcard that matches any constructor.

3.1.3 Discussion

The analysis of Catch works on a first-order language. The input program needs to be defunctionalized before it can be analyzed. The defunctionalization algorithm employed by Catch is not complete.

Calculated preconditions are unnecessarily restrictive in the presence of laziness (Mitchell, p. 142).

[2]Personal communication

$$\begin{array}{lcll} \text{t} & ::= & \{x|p\} & \text{Predicate} \\ & | & x : t_1 \to t_2 & \text{Dependent function} \\ & | & (t_1, t_2) & \text{Constructor} \\ & | & \texttt{Any} & \text{Polymorphic “Any”} \end{array}$$

Figure 3.1: Syntax of Xu’s contracts

3.2 Static Contract Checking

3.2.1 Overview

Dana Xu’s work on static contract checking for Haskell [17] (including the ESC/Haskell system [16]) allows checking of arbitrary programmer supplied pre- and postconditions on functions and is able to detect pattern match failures in the process.

Programmers define pre- and postconditions in the form of a contract (a refinement type). An appropriate contract for the `head` function defined above might be:

```
{-# CONTRACT head :: {s | not (null s)) -> {x | True} #-}
```

This contract states that if head is given a list `s` for which the predicate `not (null s)` holds, i.e. it is a non-empty list, the function will not crash[3], indicated by the trivial postcondition `True` on the return value.

Contracts can be constructed from predicates, a dependent function space constructor, arbitrary constructors and a polymorphic `Any` contract that is always satisfied, including by functions which crash (Fig. 3.1). Any `Bool`-valued Haskell expression can be used as a predicate.

Dependent functions are helpful when declaring a contract, e.g. for the `reverse` function:

```
{-# CONTRACT reverse :: {xs | True} -> {rs | length xs == length rs} #-}
```

The predicate on the return value depends on the the input. Constructors and the `Any` contract are useful when declaring a contract for `fst`:

```
{-# CONTRACT fst :: (Ok, Any) -> Ok #-}
```

Here `Ok` is used as a shorthand for $\{x$ `| True`$\}$. As `fst` discards the value on the right in the tuple we should not care if it is an expression that crashes, so we can not use `Ok`.

3.2.2 Contract checking

Haskell’s syntax is extended with two *exception values* – `BAD` and `UNR` – which are only used internally by the checker. `BAD` signifies an expression which crashes and `UNR` an expression which is either unreachable or diverges.

In the program that is going to be verified all crashing expressions (including `error` and `undefined`), as well as missing patterns in case-expressions are replaced by `BAD` exceptions.

[3]The system only guarantees partial correctness, so the term might still diverge.

The verification of the contracts is based on a translation of the contracts into Findler–Felleisen wrappers [1], which will cause a function to crash (using `BAD`) if it disobeys its contract or diverge if it called in a way that is not permitted. While technically interesting it is not directly relevant to the detection of pattern match failures so we shall not discuss it in more depth.

The actual verification process continues by *symbolic evaluation*, basically applying various simplifications to the resulting program including β-reductions. Any code deemed to be unreachable is pruned from the program.

The presence of recursive functions in the contracts might cause the symbolic evaluation to diverge if care is not taken to limit the number of evaluation steps. By setting an upper bound on the number of simplification steps we lose accuracy, but gain decidability of the verification process.

Arithmetical expressions in case-expressions (e.g. `case x*x >= 0 of` { `True -> ...; False -> BAD` }) cannot be handled directly by the symbolic evaluator. These are collected in a table and send to an external constraint or SMT solver. If the solver determines these expressions are inconsistent the code is unreachable and can be pruned.

After the symbolic evaluation has terminated the checker will tell the programmer if the program is "definitely satisfies the contract" (if no `BAD` exceptions remain anywhere in the program), "definitely does not satisfy the contract" or "don't know."

3.2.3 Discussion

Unlike our envisioned system, Xu's static contract checking requires programmer supplied contracts on nearly all functions (contracts on trivial functions can be omitted and are handled by inlining their definition when called.) Compared to Mitchell's Catch it can handle higher-order functions natively and has a more (too?) expressive contract language.

3.3 Dependent ML

DML($\mathcal{L}$) is an extension of the ML programming language which enriches ML's type system with a limited form of dependent types over a (decidable) constraint domain (or index language) $\mathcal{L}$ [15].

Xi's initial example – recast in a more Haskell-like syntax – declares a `List` data type dependent on an integer corresponding to the length of the list and a function to concatenate two such lists:

```
type Nat = {a :: Int | a >= 0}

data List<Int> a = Nil<0>
                 | {n :: Nat} Cons<n+1> a (List<n> a)

(++) :: {m :: Nat} {n :: Nat} List<m> a -> List<n> a -> List<m+n> a
(++) Nil         ys = ys
(++) (Cons x xs) ys = Cons x (append xs ys)
```

For many functions producing a list, the exact length might not be derived by such a trivial calculation (`m+n`) on the lengths of the input lists. A prominent example is the `filter` function:

```
filter :: (a -> Bool) -> {m :: Nat} List<m> a -> [n :: Nat | n <= m] List<n> a
filter p Nil         = Nil
filter p (Cons x xs) | p x       = Cons x (filter p xs)
                     | otherwise =         filter p xs
```

here we only know that the resulting list is equal or shorter in length than the input list (`n <= m`). This is expressed as the dependent sum `[n :: Nat | n <= m]`. In a full-fledged dependently-typed language we would also return a proof object stating the predicate `p` holds for all the elements in the output list, but this is beyond the expressive power of DML.

What we gain is a relief from the need to provide proofs for trivial arithmetical (in)equalities. Imagine a `zip` function which requires the two list to be zipped together to be of equal length. When calling `zip (xs ++ ys) (ys ++ xs)` this preconditions seems to be intuitively satisfied. From the point-of-view of the compiler the first list has length $m + n$ and the second lists $n + m$, however. In most dependently-typed languages we will now have to invoke a lemma or tactic proving the commutativity of addition. DML can simply send the constraint $m + n = n + n$ to an ILP solver and conclude the constraint is satisfiable.

3.3.1 Discussion

Compared to Xu's static contract checking, Xi's DML constrains the constraint system to a decidable theory. The constraints and indices – such as `m+n` and `n <= m` – may superficially look like ordinary Haskell expression, but in fact belong to a much smaller index language. While type checking is decidable, type inference is not and as a result DML still requires type annotations.

To be applicable to a pattern match analysis we must try to infer a type like:

```
head :: {n :: Int | n >= 1} [a]<n> -> a
```

for the `head` function. We also have to implicitly extend the list data type with an index representing its length.

This seems significantly more challenging than inferring the Catch-like type

```
head :: {xs :: [a] | xs ∈ (:) } -> a
```

The correspondence between the the type and the pattern matching happening in the definition of `head` is much more direct.

3.4 Compiling Pattern Matching

Case-statements in toy languages often have a very simple *decision tree* semantics. Case-statements in Haskell have a more complex *backtracking automaton* semantics. There is a body of work on compiling the latter into the former. Maranget [6] gives an algorithm for determining whether a case-statement is exhaustive and whether all patterns are useful. As the analysis does not consider any dataflow information it is much too imprecise for our purpose. It does turn out that the analysis closely follows, but can at places be simplified with respect to, the manner in which pattern matching is compiled. This indicates might also be able to follow such an approach when analyzing case-statements in our pattern-match analysis.

Chapter 4

Approach

4.1 Incremental development

We will incrementally develop our analysis for the various constructs in the Haskell language.

Variables, constants, abstraction, application, if-statements and simple let-bindings We will start out by implementing the basic constructs of the simply typed λ-calculus. This includes simple let-bindings, i.e. non-recursive let-bindings in which no pattern matching or generalization occurs.

At this stage we will only allow booleans and integers as constants. Due to their primitive and infinity nature integers will need to be treated specially by the analysis in any case. Booleans and if-statements can be handles in a more general fashion during later stages.

Recursion The next elementary construct is recursion. This will require us to implement a form of *widening* for the integers and lists to ensure termination of the analysis. As the exact form of widening used can have a big impact on the precision of the analysis we do not want to commit to a specific lattice here. Various widening strategies should be easily pluggable into the source code and perhaps eventually be able to be specified at a higher-level by the user of the tool.

Simple case-statements The most critical part of the analysis will be how it handles case-statements. In this state we will add simple case-statements (i.e., non-nested, non-backtracking, guardless case-statements) and primitive list constants, our first recursive data type.

Complex case-statements and guards To be able to handle the full Haskell 98 language our analysis must be able to deal with the complex semantics of case-statements. This includes nested-patterns, guards and backtracking. We would like to be able to analyze any boolean-valued expression that is used as a guard.

Algebraic data types Next we extend the analysis to work on arbitrary programmer-defined data types instead of only on our primitive list data type. This will mostly involve generalizing out analysis to work on with a representation of arbitrary algebraic data types. Data types containing fields with function types may also prove interesting.

Polymorphism Polymorphism can generally be added to an analysis as a relatively orthogonal feature. However, there might be some interesting interactions with our refinement and intersection types here.

Full let-bindings We extend the analysis for let-bindings to handle (mutual) recursion and pattern matching.

Type classes An interesting feature of Haskell's type system are type classes. We currently do not foresee any specific difficulties with respect to be able to handle them in out analysis.

Syntactic sugar As the final stage we would add constructs which are only syntactic sugar and therefore would not pose any significant problem to our analysis. This includes for example list-comprehensions and do-notation.

4.2 Refinement types

Compared to Freeman and Pfenning's approach to refinement types we will initially not rely on programmer supplied, but instead build our lattice of types from finite unrollings of data types. We hope this makes calculating the lattice easier.

Freeman's inference algorithm for refinement type assign a single, albeit complex, type to each binding. This makes it a monovariant analysis. We will develop a polyvariant analysis, making it sensitive to the sites where a binding is used. We hope this will improve the performance of the analysis.

4.3 Parsing

We will develop our tool in the Haskell programming language. A number of libraries for parsing input programs are available. The most obvious choice seem to be the `haskell-src-exts` package, which is also used by the HLint tool. If this proves to be a problematic choice, other possibilities are the parser from GHC or UHC.

Chapter 5

Deliverables

There will be two deliverables of this project:

1. A *tool* implementing the pattern match analysis for Haskell 98. This tool is intended to be practically usable and not a mere research prototype.
2. A *thesis* describing the theory underlying the tool and documenting the implementation choices made in the tool.

Chapter 6

Evaluation

The evaluate whether we were able to answer research question we first need to evaluate whether the developed tool implements a practically usable pattern-match analysis. We can do so by trying it out on the examples sketched in Section 1.1, the literature and for example the `nofib` testsuite.

If we can answer this question with a “yes”, and assuming the tool indeed implements the analysis using a type and effect system and refinement types, we can also answer out research question with a “yes”.

In case of a “no” we will need to argument whether this is due to a failure of our implementation or because there are good theoretical reasons why such an analysis cannot be implemented.

Chapter 7

Planning

Implement framework (April) Implement a constraint-based version of Hindley–Milney for Haskell (the underlying the system of the analysis). This is known work [11, 5] and should not prove to be difficult. This implementation will provide a guiding framework for the analysis, highlight possible issues we might run into when implementing a constraint-based analysis for full Haskell and allow us to decide which parsing library and abstract syntax representation to use.

Implement analysis (April–July) The largest portion of time is assigned to incrementally developing and building the pattern-match analysis. The likely order will be:

1. Variables, constants, abstraction, application, if-statements and simple let-bindings (1.5 weeks)
2. Recursion (1.5 weeks)
3. Simple case-statements (2 weeks)
4. Complex case-statements and guards (2 weeks)
5. Algebraic data types (1.5 weeks)
6. Polymorphism (1.5 weeks)
7. Full let-bindings (1 week)
8. Type classes (1.5 weeks)
9. Syntactic sugar (1.5 weeks)

Evaluation and thesis writing (July–August) The final phase of this project will include an evaluation of the tool and finishing of the thesis. A significant potion of the thesis is expected to be written in parallel with the implementation, after a particular subtask has been implemented successfully.

Chapter 8

Future Work

Non-goals of the thesis project are:

1. A formal soundness proof of the analysis, and
2. Supporting Haskell extensions beyond Haskell 98.

Both of these would be candidates for future work.

Bibliography

[1] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[2] Tim Freeman. Refinement types for ML, 1994.

[3] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[4] Ferenc Gécseg and Magnus Steinby. *Tree Automata.* Akadéniai Kiadó, Budapest, Hungary, 1984.

[5] Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electr. Notes Theor. Comput. Sci.*, 236:163–183, 2009.

[6] Luc Maranget. Warnings for pattern matching. *J. Funct. Program.*, 17(3):387–421, 2007.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[8] Neil Mitchell. Transformation and analysis of functional programs.

[9] Neil Mitchell and Colin Runciman. A static checker for safe pattern matching in haskell. In Marko C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 15–30. Intellect, 2005.

[10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[11] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, January 1999.

[12] Benjamin C. Pierce. Programming with intersection types and bounded polymorphism. Technical report, 1991.

[13] John C. Reynolds. Design of the programming language forsythe. Technical report, 1996.

[14] Simon, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. The Haskell 98 Report, 1999.

[15] Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

[16] Dana N. Xu. Extended static checking for haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 48–59, New York, NY, USA, 2006. ACM.

[17] Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *In Proceedings of the 36 th Annual ACM Symposium on the Principles of Programming Languages*, pages 41–52. ACM, 2009.