Higher-Order Pattern Match Analysis

Ruud Koot

MSc Thesis

August 31, 2012

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Supervisor:
prof. dr. S.D. Swierstra
Daily Supervisor:
dr. J. Hage

Abstract

In this thesis we attempt to answer 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 and type and effect systems. In Chapter 3 we give an overview of the pattern match analysis we developed and give a detailed description of the constraint generation and constraint solving phases in respectively Chapter 4 and Chapter 5. In Chapter 7 we discussion the implementation of the analysis we have built. We evaluate the effectiveness and limitations of our analysis in Chapter 6. In Chapter 8 we present work related to our research question and discuss which aspects of that work are relevant to or different from our proposed system. Finally, in Chapter 9, we propose a number of directions for further research and future implementation work to improve the precision and applicability of the analysis.

Acknowledgments

I would like to acknowledge my daily supervisor, Jurriaan Hage, and the members of the Friday Pieday Club, including—but not limited to—Paul, Sjoerd and Ruben.

Contents

1 Introduction **11**
1.1 Examples 12

2 Context **15**
2.1 Haskell 15
2.1.1 Pattern Matching 16
2.2 Type and Effect Systems 16
2.2.1 Example 16
2.2.2 Type Inference 18
2.2.3 Polyvariance 18

3 Pattern Match Analysis **19**
3.1 Overview 19
3.2 Higher-Order Functions 19
3.3 Analysis 21

4 Constraint Generation **23**
4.1 Overview 23
4.2 Types and Annotations 23
4.3 Refinements and Lattices 26
4.3.1 Booleans 26
4.3.2 Integers 26
4.3.3 Lists 27
4.4 Type System 28
4.5 Instantiation and Generalization 28
4.6 Unification 32

5 Constraint Solving **33**
5.1 Overview 33
5.2 Dependency Analysis 34
5.3 Worklist Algorithm 34
5.3.1 Generic Worklist Algorithm 34
5.3.2 Resolving Individual Constraints 34

6 Evaluation **37**
6.1 Examples 37
6.1.1 Higher-order functions 37
6.1.2 Detecting pattern match failures 37
6.1.3 *filter* 37
6.2 Limitations 38
6.2.1 Context-insensitivity 38
6.2.2 Structurally recursive functions 38

7 Implementation **39**

8 Related Work **41**
8.1 Neil Mitchell's *Catch* 41
8.1.1 Overview 41
8.1.2 Constraint Systems 42
8.1.3 Discussion 42
8.2 Static Contract Checking 43
8.2.1 Overview 43
8.2.2 Contract Checking 43
8.2.3 Discussion 44
8.3 Dependent ML 44
8.3.1 Discussion 45
8.4 Refinement Types 46
8.4.1 Union and Intersection Types 46
8.4.2 Constructors and Pattern Matching 47
8.5 Compiling Pattern Matching 47

9 Conclusions and Further Research **49**
9.1 Conclusions 49
9.2 Improving Precision 49
9.2.1 Intersection Types 49
9.2.2 Implication Constraints 50
9.2.3 Set-based Constraints 51
9.3 Improving Applicability 51
9.3.1 Handling $undefined$ 51

A Demonstration of Implementation **53**
A.1 Input 53
A.2 Output 53
A.3 Legend 57

List of Figures

2.1 Underlying type system 17
2.2 Annotated type system 17

4.1 Naming conventions 23
4.2 Annotated types 24
4.3 Annotated type schemes 25
4.4 Annotated type environments 25
4.5 Annotations/refinements 25
4.6 Equality constraints 25
4.7 Subset constraints 25
4.8 Trivial abstraction for **Bool** 26
4.9 Sign abstraction for **Int** 26
4.10 Parity abstraction for **Int** 26
4.11 Operations on refinements 27
4.12 Shape abstraction for lists 27
4.13 Instantiation 29
4.14 Generalization 29
4.15 Type system (part 1) 30
4.16 Type system (part 2) 31

7.1 Module overview 39

8.1 Syntax of Xu's contracts 43
8.2 A lattice of *List* and its *rectype*s. 46

Chapter 1

Introduction

In 1978, Robin Milner [8] 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: 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 $3 + \mathit{true}$—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 $f \leadsto f \leadsto ...$ 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 $head$:

$$\begin{aligned}&main = \textbf{let}\ xs = \textbf{if}\ length\ \texttt{"foo"} > 5\ \textbf{then}\ [1,2,3]\ \textbf{else}\ [\,]\\&\qquad\quad \textbf{in}\ head\ xs\\&head\ (x : xs) = x\end{aligned}$$

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 $desugar :: ComplexAST \rightarrow SimpleAST$—duplicating most of the data type representing and functions operating on the abstract syntax tree—or take the easy route $desugar :: AST \rightarrow AST$ 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 the 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 easily be 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 [10], calculating monotonically increasing segments of a list (e.g., $risers\ [1,3,5,1,2] \leadsto [[1,3,5],[1,2]]$):

$$
\begin{aligned}
&\mathit{risers} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [[a]] \\
&\mathit{risers}\ [\,] \qquad\qquad\quad = [\,] \\
&\mathit{risers}\ [x] \qquad\qquad\ \ = [[x]] \\
&\mathit{risers}\ (x_1 : x_2 : \mathit{xs}) = \mathbf{let}\ (s : \mathit{ss}) = \mathit{risers}\ (x_2 : \mathit{xs}) \\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{in\ if}\ x_1 \leqslant x_2\ \mathbf{then}\ (x_1 : s) : \mathit{ss}\ \mathbf{else}\ [x_1] : (s : \mathit{ss})
\end{aligned}
$$

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):

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Bitstring} = [\mathit{Int}] \\
&\mathit{add} :: \mathit{Bitstring} \rightarrow \mathit{Bitstring} \rightarrow \mathit{Bitstring} \\
&\mathit{add}\ [\,] \qquad y \qquad\ = y \\
&\mathit{add}\ x \qquad [\,] \qquad = x \\
&\mathit{add}\ (0 : x)\ (0 : y) = 0 : \mathit{add}\ x\ y \\
&\mathit{add}\ (0 : x)\ (1 : y) = 1 : \mathit{add}\ x\ y \\
&\mathit{add}\ (1 : x)\ (0 : y) = 1 : \mathit{add}\ x\ y \\
&\mathit{add}\ (1 : x)\ (1 : y) = 0 : \mathit{add}\ (\mathit{add}\ [1]\ x)\ y
\end{aligned}
$$

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 [12]. 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 syntax of Haskell and some features mentioned previously:

```haskell
data Tree a = Branch (Tree a) (Tree a)
            | Leaf a
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Branch t1 t2) = Branch (mapTree f t1) (mapTree f t2)
mapTree f (Leaf a)       = Leaf (f a)
instance Functor Tree where
  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 $f :: a \to b$ 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 coincide, 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 recurses 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:

$$\begin{aligned}
&rotate\ (Branch\ b@(Branch\ (Leaf\ x)\ (Leaf\ y))\ (Leaf\ z))\\
&\quad \mid z < x \qquad = Branch\ (Leaf\ z)\ b\\
&\quad \mid z < y \qquad = Branch\ (Branch\ (Leaf\ x)\ (Leaf\ z))\ (Leaf\ y)\\
&\quad \mid otherwise = undefined
\end{aligned}$$

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 [11]. In this formalism we take 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 2.2. 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.

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} c : \tau_c}\ [\text{T-Con}] \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathrm{UL}} x : \tau}\ [\text{T-Var}]$$

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

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash_{\mathrm{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} e_1\ e_2 : \tau_1}\ [\text{T-App}]$$

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

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

Figure 2.1: Underlying type system

$$\frac{}{\Gamma \vdash_{\mathrm{AN}} n : \text{Int}^{\{n\}}}\ [\text{T-Con}] \qquad \frac{\Gamma(x) = \widehat{\tau}}{\Gamma \vdash_{\mathrm{AN}} x : \widehat{\tau}}\ [\text{T-Var}]$$

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

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

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

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

$$\frac{\Gamma \vdash_{\mathrm{AN}} e : \tau^{\varphi}}{\Gamma \vdash_{\mathrm{AN}} e : \tau^{\varphi \cup \varphi'}}\ [\text{T-Sub}]$$

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:

$$\begin{aligned} main = {} & \mathbf{let}\ id\ x = x \\ & \mathbf{in}\ (id\ 1, id\ 2) \end{aligned}$$

When performing the analysis given above we would assign the type $Int \to Int^{\{1,2\}}$ to id and as a result the type $Int^{\{1,2\}} \times Int^{\{1,2\}}$ 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:

$$\begin{aligned} main = {} & \mathbf{let}\ id\ x = x \\ & \mathbf{in}\ (id\ 1, id\ true) \end{aligned}$$

Hindley–Milner manages to avoid problems in this situation by a mechanism called *let-generalization*. The binding id has the type $\alpha \to \alpha$. 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 $\forall \alpha. \alpha \to \alpha$. In the body of the let-binding, id can be instantiated twice: once to $Int \to Int$ and once to $Bool \to Bool$.

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

Chapter 3

Pattern Match Analysis

3.1 Overview

The key idea behind the analysis is to keep track of the possible values each variable and expression can have. When pattern matching with an if-then-else or case-expression, we verify that the values the scrutinee can have are covered by the patterns in each of the alternatives.

The set of possible values associated with each variable and expression is called a *refinement* and placed as an annotation on the type of that variable or expression. We call the type without such an annotation the *underlying type* of an expression and the type with such an annotation the *annotated type* or *refinement type*.

Examples

$$\mathbf{True} \quad : \quad \mathbf{Bool}^{\{\mathbf{True}\}} \tag{3.1}$$

$$42 \quad : \quad \mathbf{Int}^{\{42\}} \tag{3.2}$$

$$(7, \mathbf{False}) \quad : \quad (\mathbf{Int}^{\{7\}}, \mathbf{Bool}^{\{\mathbf{False}\}})^{\top} \tag{3.3}$$

$$[3, 2, 1] \quad : \quad [\mathbf{Int}^{\{1,2,3\}}]^{\{(_:_:_:[])\}} \tag{3.4}$$

$$\lambda x.\, x + 1 \quad : \quad \mathbf{Int}^{\top} \xrightarrow{\top} \mathbf{Int}^{\top} \tag{3.5}$$

Similarly, by analyzing the patterns of case-statements we infer the maximal set of values each variable can have without causing a pattern-match failure. Thus, if the set of values a variable can have is not a subset of the values it is allowed to have, we have statically detected a potential source of pattern-match failures.

3.2 Higher-Order Functions

The analysis should be able to handle higher-order functions. Let us take a moment to review what the refinements type should look like for higher-order functions and how they should be interpreted.

Consider the function:

$$
\begin{aligned}
main\ b\ f = \mathbf{if}\ b\ \mathbf{then} & \\
& \mathbf{if}\ f\ 42\ \mathbf{then}\ 100\ \mathbf{else}\ 200 \\
\mathbf{else} & \\
& \mathbf{if}\ f\ 43\ \mathbf{then}\ 300\ \mathbf{else}\ 400
\end{aligned}
$$

The function $main$ takes a boolean and a function as its arguments. The argument *f* is applied to an integer constant in the body of main and—because it is used as the scrutinee of an if-then-else expression—should return a boolean as its result. Depending on the values of b and f, the latter after having been applied to either 42 or 43, the function $main$ will return one of four different integer values.

From this we can infer the underlying type of $main$ should be:

$$\mathbf{Bool} \to (\mathbf{Int} \to \mathbf{Bool}) \to \mathbf{Int}$$

Under what conditions is this function guaranteed not to crash due to a pattern match failure? The argument b and function f, after having been applied to an integer, can evaluate to arbitrary boolean values. The function f, however, might also cause a pattern match failure while being evaluated. We need to be sure it does not do so if passed one of the values 42 or 43. An appropriate choice for the refined type would thus be:

$$\mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}} \to (\mathbf{Int}^{\{42,43\}} \to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\{100,200,300,400\}}$$

Note the difference in meaning of the refinement depending on whether it appears on an underlying type in a *covariant* or *contravariant* position. For clarity, we give the type once more with variance annotations (where + means covariant and - means contravariant):

$$\mathbf{Bool}_-^{\{\mathbf{True},\mathbf{False}\}} \to (\mathbf{Int}_+^{\{42,43\}} \to \mathbf{Bool}_-^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}_+^{\{100,200,300,400\}}$$

Refinements appearing on types in a covariant position give either the minimal set of values that a function can return, or—in the case of arguments passed to a functional parameter—a minimal set of values that function should be able to accept without failing due to a pattern match. In all cases the set of values appearing in the refinement can safely be enlarged.

Refinements appearing on types in a contravariant position give either the maximal set of values that can be passed to an argument, or—in the case of a return value of a functional parameter—a maximal set of values which that function is allowed to return without causing a pattern match failure in the caller. In all cases the set of values appearing in the refinement can safely be reduced.

Another valid refinement type for $main$ would be:

$$\mathbf{Bool}^{\{\mathbf{True}\}} \to (\mathbf{Int}^{\{41,42,43\}} \to \mathbf{Bool}^{\{\mathbf{False}\}}) \to \mathbf{Int}^{\{100,200,300,400,500\}}$$

This type is less desirable than the previous refinement type: it accepts fewer values for the parameter b, the functions satisfying the type $\mathbf{Int}^{\{41,42,43\}} \to$

$\mathbf{Bool}^{\{\mathbf{False}\}}$ form a subset of the functions satisfying $\mathbf{Int}^{\{42,43\}} \rightarrow \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}$ and the larger refinement on the result type may prevent $main$ from being passed as an argument to another higher-order function.

The analysis should strive to infer refinement types with the smallest possible refinements on types in covariant positions and the largest possible refinement on types in contravariant positions.

3.3 Analysis

The analysis is *constraint-based* and *type-directed*, meaning it is formulated as a two part process.

The first is a type system that generates a set of constraints from the abstract syntax tree. It generates two distinct set of constraints: one set allows us to infer the underlying type of expressions, the other allows us to construct the refinements belonging those types.

The second part consists of solving the generated constraints. The first set of constraints—representing the underlying types—can be solved using unification. The second set requires a more complex solver based on a worklist algorithm.

The generation of constraints and the unification algorithm are described in Chapter 4. The constraint solver for the second constraint set is described in Chapter 5.

Chapter 4

Constraint Generation

4.1 Overview

The constraint generation phase of the analysis is built on top of a constraint-based version of the Hindley–Milner type system [8, 14] extended to handle annotations and generate constraints on and between those annotations.

As mentioned in Section 3.3 the type system generating the constraints produces two distinct sets of constraints. The first is a set of equality constraints between *simply annotated types* as given in Figure 4.2. Simply annotated types are annotated, except that all annotations are restricted to be plain annotation variables. The reason for this is restriction is to make annotated types into a free algebra, so the constraint set can be solved by a unification algorithm presented in Section 4.6. The result of the unification algorithm is a substitution from type variables to simply annotated types, that is applied to both the inferred type and the second constraint set. The second constraint set consists of subtype constraints (Figure 4.7) between annotations (Figure 4.5).

4.2 Types and Annotations

Throughout this and the next chapter we will assume the following naming conventions for variables:

$x, f \in \mathbf{Var}$	variables
$\widehat{\tau} \in \widehat{\mathbf{Type}}$	annotated types
$\alpha \in \mathbf{TVar}$	type variables
$\widehat{\Gamma} \in \widehat{\mathbf{TEnv}}$	annotated type environments
$\varphi \in \mathbf{Ann}$	annotations/refinements
$\beta \in \mathbf{AVar}$	annotation variables

Figure 4.1: Naming conventions

An *annotated type* (Figure 4.2) can be a plain type variable α, a type constant such as **Bool** or **Int** with an annotation φ, a tuple of two annotated types, a list with elements of an annotated type, or a function with the domain and range being given by annotated types.

$$\begin{array}{rcl}
\widehat{\tau} & ::= & \alpha \\
& | & \mathbf{Bool}^{\varphi} \\
& | & \mathbf{Int}^{\varphi} \\
& | & \mathbf{(}\widehat{\tau}, \widehat{\tau}\mathbf{)}^{\varphi} \\
& | & \mathbf{[}\widehat{\tau}\mathbf{]}^{\varphi} \\
& | & \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2
\end{array}$$

Figure 4.2: Annotated types

There is no annotation on the type variable as it gains one when it is substituted by a type constant or constructor. In any other case, we are dealing with a polymorphic argument of a polymorphic function. As we are not able to pattern match on such values, it is not useful to keep track of the values such a variable can have.

The annotations on tuples and functions are not genuinely used, in the sense that in the current implementation of the analysis their elements are always picked from the trivial one element lattice. We have included them here for uniformity and to make the framework more general for future work.

Lists indirectly have an annotation both on the type of its elements and on the list as a whole. This allows for more flexible ways of specifying the possible values a list may have. Recall the add example from Section 1.1. While we could attempt to precisely represent all possible lists that might occur—remembering both the shape of the list and which values each element at each each position in the list can have—this approach is unlikely to scale in practice. The only information necessary for determining that the add function does not fail due to a pattern match is that its inputs should be lists of integers where each elements in the lists is an integer in the set $\{0, 1\}$. We can also infer it will always produce such a list. An appropriate type for add would thus be

$$\mathbf{[Int}^{\{0,1\}}\mathbf{]}^{\{\star\}} \rightarrow \mathbf{[Int}^{\{0,1\}}\mathbf{]}^{\{\star\}} \rightarrow \mathbf{[Int}^{\{0,1\}}\mathbf{]}^{\{\star\}}$$

where the refinement $\star$ represents a list of any shape.

When analyzing polymorphic functions working on lists of any type of elements, we are likely only interested in the shape of the list and not in the values of its elements. For example, an appropriate type for $head$ would be:

$$\mathbf{[}\alpha\mathbf{]}^{\{\mathbf{(}_:\star\mathbf{)}\}} \rightarrow \alpha$$

Like in the Hindley–Milner type system, we represent polymorphic types by a *type scheme* (Figure 4.3). The difference here is that we are also able to

$$\widehat{\sigma} \quad ::= \quad \forall \overline{\alpha}\overline{\beta}.\, R \Rightarrow \widehat{\tau}$$

Figure 4.3: Annotated type schemes

quantify over, not only type variables, but also over annotation variables. Furthermore, we associate a set R of subtype constraints (Figure 4.7) with each type scheme. These are constraints the constraint solver was unable to discharge when the type was generalized and should be verified to hold when the type is later instantiated.

Type schemes can be associated with variables using a type environment (Figure 4.4).

$$\widehat{\Gamma} \quad ::= \quad \epsilon \mid \widehat{\Gamma},\, x : \widehat{\sigma}$$

Figure 4.4: Annotated type environments

Annotations hold the refinements belonging to the underlying type. Refinements form a set of annotation variables and abstract values (Section 4.3).

$$\begin{array}{rcl} \varphi & ::= & \beta \\ & \mid & \{\pi_\tau\} \\ & \mid & \varphi_1 \cup \varphi_2 \end{array}$$

Figure 4.5: Annotations/refinements

The type system generates a set of *equality constraints* between annotated types and a set of *subset constraints* between refinements:

$$\begin{array}{rcl} C & ::= & \{\widehat{\tau}_1 = \widehat{\tau}_2\} \\ & \mid & C_1 \cup C_2 \end{array}$$

Figure 4.6: Equality constraints

$$\begin{array}{rcl} R & ::= & \{\varphi_1 \subseteq \varphi_2\} \\ & \mid & R_1 \cup R_2 \end{array}$$

Figure 4.7: Subset constraints

4.3 Refinements and Lattices

In the examples we have seen so far, we have used the actual values of a type as refinements. This approach will at best fail to scale up when analyzing larger programs, but worse, will likely cause the constraint solver to diverge: while the refinements naturally form a lattice under subset inclusion with the meet and join operations given by intersection and union, it will not satisfy the ascending chain condition for infinite types, such as integers and list. As a result the refinements can grow infinitely large.

To overcome this problem we abstract from the concrete values of a type in its refinement and apply widening to deal with recursive data types.

4.3.1 Booleans

Booleans are a finite type and thus a set of booleans forms finite lattice under subset inclusion. Finite lattices always satisfy the ascending chain condition, thus there is little need to abstract the values of a boolean in a refinement.

$$\begin{array}{rcl} \pi_{\mathbf{Bool}} & ::= & \mathbf{True} \\ & | & \mathbf{False} \end{array}$$

Figure 4.8: Trivial abstraction for **Bool**

4.3.2 Integers

Integers form an infinite set, so will need to abstract from them. Possible choices include taking only the sign of an integer into account:

$$\begin{array}{rcl} \pi_{\mathbf{Int}} & ::= & \mathbf{+} \\ & | & \mathbf{0} \\ & | & \mathbf{-} \end{array}$$

Figure 4.9: Sign abstraction for **Int**

or only its parity:

$$\begin{array}{rcl} \pi_{\mathbf{Int}} & ::= & \mathbf{Odd} \\ & | & \mathbf{Even} \end{array}$$

Figure 4.10: Parity abstraction for **Int**

More complicated alternatives are possible. We could take the product of the sign and parity lattices or partition the integers into a finite number of intervals with the cuts determined by concrete integers appearing in the program text.

The type system producing the constraints and constraint solver are agnostic towards the actual choice. They only require a number of operations to be defined: abstracting a concrete value (4.2), computing the greatest lower and least upper bound of two refinements (4.3 and 4.4) and determining whether one refinement is contained in (smaller than) another (4.4).

$$\iota \;:\; \tau \to \pi_\tau \tag{4.1}$$
$$\sqcup \;:\; \varphi \to \varphi \to \varphi \tag{4.2}$$
$$\sqcap \;:\; \varphi \to \varphi \to \varphi \tag{4.3}$$
$$\sqsubseteq \;:\; \varphi \to \varphi \to \mathbf{Bool} \tag{4.4}$$

Figure 4.11: Operations on refinements

4.3.3 Lists

Lists, being a recursive data type, require a more complicated abstraction. We might choose to abstract to only the shape of the list, remembering the nesting of nil and cons constructors, but forgetting the elements of the list stored in the cons constructors:

$$\begin{array}{lcl} \pi_{\mathbf{List}} & ::= & \beta \\ & | & [] \\ & | & (_ : \pi_{\mathbf{List}}) \\ & | & \star \end{array}$$

Figure 4.12: Shape abstraction for lists

In order to represent constraints generated by pattern-matching and applying a cons constructor to another expression, we need to be able to store annotation variables (β) in the recursive positions of the cons constructor.

As lists can be of infinite length, this is not yet sufficient to guarantee that the lattice formed by sets of abstracted lists satisfies the ascending chain condition. The constraint solver will thus use widening operators $\nabla : \varphi \to \varphi \to \varphi$ and $\Delta : \varphi \to \varphi \to \varphi$ to calculate the greatest lower and least upper bounds after instantiating annotation variables in an abstracted lists (see Section 5.3.2), that will limit the depth of an abstract list by replacing the recursive position of the cons constructor at a user-specified depth by the wildcard $\star$.

4.4 Type System

The analysis is defined by the *constraint typing relation*

$$\widehat{\Gamma} \vdash e : \widehat{\tau} \leadsto C \ \& \ R.$$

The relations says that expression e has the annotated type $\widehat{\tau}$ in the typing environment $\widehat{\Gamma}$ if the type constraints C and refinements constraints R are satisfied.

The relation is defined by the typing rules given in Figure 4.15 and 4.16.

Implementation notes The implementation of this analysis is also capable of handling simultaneous and mutually recursive functions and bindings. For each binding group, we build a graph with a vertex for each binding and edges from each usage of a particular binding to its definition. We determine the strongly connected components of this graph—each component being a group of mutually recursive bindings—and topologically sort the condensed graph to determine the order in which the let-bindings should be nested. Each mutually recursive group

$$\begin{aligned}
&\mathbf{let}\ x_1 = \ \dots\ x_i\ \dots\ x_j\ \dots \\
&\quad\ \ x_2 = \ \dots\ x_i\ \dots\ x_j\ \dots \\
&\quad\ \ \dots \\
&\mathbf{in}\ \ \dots
\end{aligned}$$

is then treated as a single recursive binding

$$\begin{aligned}
&\mathbf{letrec}\ x_{\mathrm{r}} = (\dots\ \pi_i\ x_{\mathrm{r}}\ \dots\ \pi_j\ x_{\mathrm{r}}\ \dots, \dots\ \pi_i\ x_{\mathrm{r}}\ \dots\ \pi_j\ x_{\mathrm{r}}\ \dots, \dots) \\
&\mathbf{in}\ \dots
\end{aligned}$$

where π_k projects the kth element from an n-tuple.

The implementation also supports tuples of arbitrary arity and list literals. Their type rules subsume T-Tuple and T-Nil.

4.5 Instantiation and Generalization

As in the Hindley–Milner type system, we need to instantiate type schemes in the T-Var type rule and generalize types to a type scheme in the T-Let and T-LetRec type rules. This last process is called *let-generalization* and is what makes our analysis polyvariant. The addition of constraints—and necessity to quantify over them—makes the instantiation and generalization slight more complicated than it is in a plain Hindley–Milner type system.

When instantiating a type scheme we replace all type variables and annotation variables in a type scheme and which are bound by a quantifier with fresh variables and drop the quantifiers from the type scheme, resulting in a tuple containing an annotated type and its associated constraint set.

During generalization we will unify the constraint set C and apply the resulting substitution θ_{C} to both $\widehat{\tau}$ and R (as some of the annotation variables present in constraints in R may have been unified as well.) Next, we invoke

$$\frac{\theta = [\overline{\alpha \mapsto \alpha'}][\overline{\beta \mapsto \beta'}] \quad \overline{\alpha'}, \overline{\beta'} \text{ fresh}}{inst(\forall \overline{\alpha}\overline{\beta}.\ R \Rightarrow \widehat{\tau}) = (\theta\widehat{\tau}, \theta R)}$$

Figure 4.13: Instantiation

the constraint solver to solve or simplify a part of the constraint set $\theta_{\mathrm{C}}R$. We quantify over all annotation variables and type variables that are free in the environment and store the remaining constraints from $\theta_{\mathrm{C}}R$—those which were not solved—in the type scheme.

The essential detail here is choosing which constraints in $\theta_{\mathrm{C}}R$ to solve. We would like our analysis to be as polyvariant as possible and thus not prematurely fix the values of any annotation variables about which we can learn more later—where "later" should be taken to mean "at the call site of a let-bound function," as we gain more information by applying arguments to it.

We thus exclude any constraints that directly or indirectly—through any transitive constraints, as determined by the dependency analysis (Section 5.2)—depend on annotation variables appearing in $\theta_{\mathrm{C}}\widehat{\tau}$ from being solved and store them in the type scheme instead, to be solved only at their call sites after having been instantiated again. We will call such constraints *input-dependent constraints*.

$$\frac{\begin{array}{c}\theta_{\mathrm{C}} = \widehat{\mathcal{U}}(C) \\ R' = \mathit{input\text{-}dependent}(\theta_{\mathrm{C}}R, \theta_{\mathrm{C}}\widehat{\tau}) \quad \theta_{\mathrm{R}} = solve(\theta_{\mathrm{C}}R - R') \\ \overline{\alpha} = ftv(\theta_{\mathrm{R}}\theta_{\mathrm{C}}\widehat{\tau}) - ftv(\widehat{\Gamma}) \quad \overline{\beta} = fav(\theta_{\mathrm{R}}\theta_{\mathrm{C}}\widehat{\tau}) \cup fav(\theta_{\mathrm{R}}R')\end{array}}{gen(\widehat{\Gamma}, \widehat{\tau}, C, R) = \forall \overline{\alpha}\overline{\beta}.\ \theta_{\mathrm{R}}R' \Rightarrow \theta_{\mathrm{R}}\theta_{\mathrm{C}}\widehat{\tau}}$$

Figure 4.14: Generalization

$$\frac{\widehat{\Gamma}(x) = \widehat{\sigma} \quad inst(\widehat{\sigma}) = (\widehat{\tau}, R)}{\widehat{\Gamma} \vdash x : \widehat{\tau} \rightsquigarrow \emptyset \;\&\; R} \text{ [T-Var]}$$

$$\frac{\widehat{\Gamma}, x : \alpha \vdash e : \widehat{\tau} \rightsquigarrow C \;\&\; R \quad \alpha, \beta \text{ fresh}}{\widehat{\Gamma} \vdash \lambda x.\, e : \alpha \xrightarrow{\beta} \widehat{\tau} \rightsquigarrow C \;\&\; R \cup \{\top \subseteq \beta\}} \text{ [T-Abs]}$$

$$\frac{\widehat{\Gamma} \vdash f : \widehat{\tau}_{\mathrm{f}} \rightsquigarrow C_{\mathrm{f}} \;\&\; R_{\mathrm{f}} \quad \widehat{\Gamma} \vdash x : \widehat{\tau}_{\mathrm{x}} \rightsquigarrow C_{\mathrm{x}} \;\&\; R_{\mathrm{x}} \quad \alpha_{\mathrm{r}}, \beta \text{ fresh}}{\widehat{\Gamma} \vdash f\; x : \alpha_{\mathrm{r}} \rightsquigarrow C_{\mathrm{f}} \cup C_{\mathrm{x}} \cup \{\widehat{\tau}_{\mathrm{f}} = \widehat{\tau}_{\mathrm{x}} \xrightarrow{\beta} \alpha_{\mathrm{r}}\} \;\&\; R_{\mathrm{f}} \cup R_{\mathrm{x}} \cup \{\beta \subseteq \top\}} \text{ [T-App]}$$

$$\frac{\beta \text{ fresh}}{\widehat{\Gamma} \vdash n : \mathbf{Int}^{\beta} \rightsquigarrow \emptyset \;\&\; \{\{\iota(n)\} \subseteq \beta\}} \text{ [T-Int]}$$

$$\frac{\beta \text{ fresh}}{\widehat{\Gamma} \vdash \mathbf{True} : \mathbf{Bool}^{\beta} \rightsquigarrow \emptyset \;\&\; \{\{\iota(\mathbf{True})\} \subseteq \beta\}} \text{ [T-True]}$$

$$\frac{\beta \text{ fresh}}{\widehat{\Gamma} \vdash \mathbf{False} : \mathbf{Bool}^{\beta} \rightsquigarrow \emptyset \;\&\; \{\{\iota(\mathbf{False})\} \subseteq \beta\}} \text{ [T-False]}$$

$$\frac{\begin{array}{c} \widehat{\Gamma} \vdash g : \widehat{\tau}_{\mathrm{g}} \rightsquigarrow C_{\mathrm{g}} \;\&\; R_{\mathrm{g}} \quad \beta \text{ fresh} \\ \widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \rightsquigarrow C_1 \;\&\; R_1 \quad \widehat{\Gamma} \vdash e_2 : \widehat{\tau}_2 \rightsquigarrow C_2 \;\&\; R_2 \\ C = C_{\mathrm{g}} \cup C_1 \cup C_2 \cup \{\widehat{\tau}_{\mathrm{g}} = \mathbf{Bool}^{\beta}, \widehat{\tau}_1 = \widehat{\tau}_2\} \\ R = R_{\mathrm{g}} \cup R_1 \cup R_2 \cup \{\beta \subseteq \{\mathbf{True}, \mathbf{False}\}\} \end{array}}{\widehat{\Gamma} \vdash \mathbf{if}\; g\; \mathbf{then}\; e_1\; \mathbf{else}\; e_2 : \widehat{\tau}_1 \rightsquigarrow C \;\&\; R} \text{ [T-If]}$$

$$\frac{\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \rightsquigarrow C_1 \;\&\; R_1 \quad \widehat{\Gamma} \vdash e_2 : \widehat{\tau}_2 \rightsquigarrow C_2 \;\&\; R_2 \quad \beta \text{ fresh}}{\widehat{\Gamma} \vdash \mathbf{(}e_1, e_2\mathbf{)} : \mathbf{(}\widehat{\tau}_1, \widehat{\tau}_2\mathbf{)}^{\beta} \rightsquigarrow C_1 \cup C_2 \;\&\; R_1 \cup R_2 \cup \{\top \subseteq \beta\}} \text{ [T-Tuple]}$$

$$\frac{\widehat{\Gamma} \vdash e : \mathbf{(}\widehat{\tau}_1, \widehat{\tau}_2\mathbf{)}^{\beta} \rightsquigarrow C \;\&\; R}{\widehat{\Gamma} \vdash \mathbf{fst}\; e : \widehat{\tau}_1 \rightsquigarrow C \;\&\; R \cup \{\beta \subseteq \top\}} \text{ [T-Fst]}$$

$$\frac{\widehat{\Gamma} \vdash e : \mathbf{(}\widehat{\tau}_1, \widehat{\tau}_2\mathbf{)}^{\beta} \rightsquigarrow C \;\&\; R}{\widehat{\Gamma} \vdash \mathbf{snd}\; e : \widehat{\tau}_2 \rightsquigarrow C \;\&\; R \cup \{\beta \subseteq \top\}} \text{ [T-Snd]}$$

Figure 4.15: Type system (part 1)

$$\frac{\alpha \text{ fresh} \quad \beta \text{ fresh}}{\widehat{\Gamma} \vdash \textbf{[]} : \textbf{[}\alpha\textbf{]}^{\beta} \rightsquigarrow \emptyset \;\&\; \{\{\textbf{[]}\} \subseteq \beta\}} \text{[T-Nil]}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \vdash e_{\mathrm{h}} : \widehat{\tau}_{\mathrm{h}} \rightsquigarrow C_{\mathrm{h}} \;\&\; R_{\mathrm{h}} \quad \widehat{\Gamma} \vdash e_{\mathrm{t}} : \widehat{\tau}_{\mathrm{t}} \rightsquigarrow C_{\mathrm{t}} \;\&\; R_{\mathrm{t}} \quad \beta_1, \beta_2 \text{ fresh} \\ C = C_{\mathrm{h}} \cup C_{\mathrm{t}} \cup \{\textbf{[}\widehat{\tau}_{\mathrm{h}}\textbf{]}^{\beta_2} = \widehat{\tau}_{\mathrm{t}}\} \quad R = R_{\mathrm{h}} \cup R_{\mathrm{t}} \cup \{\{\textbf{(}_\textbf{:}\beta_2\textbf{)}\} \subseteq \beta_1\}\end{array}}{\widehat{\Gamma} \vdash e_{\mathrm{h}} \textbf{:} e_{\mathrm{t}} : \textbf{[}\widehat{\tau}_{\mathrm{h}}\textbf{]}^{\beta_1} \rightsquigarrow C \;\&\; R} \text{[T-Cons]}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \vdash g : \widehat{\tau}_{\mathrm{g}} \rightsquigarrow C_{\mathrm{g}} \;\&\; R_{\mathrm{g}} \quad \widehat{\Gamma} \vdash e_{\mathrm{n}} : \widehat{\tau}_{\mathrm{n}} \rightsquigarrow C_{\mathrm{n}} \;\&\; R_{\mathrm{n}} \quad \alpha, \beta_1 \text{ fresh} \\ C = C_{\mathrm{g}} \cup C_{\mathrm{n}} \cup \{\widehat{\tau}_{\mathrm{g}} = \textbf{[}\alpha\textbf{]}^{\beta_1}\} \quad R = R_{\mathrm{g}} \cup R_{\mathrm{n}} \cup \{\beta_1 \subseteq \{\textbf{[]}\}\}\end{array}}{\widehat{\Gamma} \vdash \textbf{case } g \textbf{ of } \{\textbf{[]} \to e_{\mathrm{n}}\} : \widehat{\tau}_{\mathrm{n}} \rightsquigarrow C \;\&\; R} \text{[T-Case-N]}$$

$$\frac{\begin{array}{c}\alpha, \beta_1, \beta_2 \text{ fresh} \\ \widehat{\Gamma} \vdash g : \widehat{\tau}_{\mathrm{g}} \rightsquigarrow C_{\mathrm{g}} \;\&\; R_{\mathrm{g}} \quad \widehat{\Gamma}, h : \alpha,\, t : \textbf{[}\alpha\textbf{]}^{\beta_2} \vdash e_{\mathrm{c}} : \widehat{\tau}_{\mathrm{c}} \rightsquigarrow C_{\mathrm{c}} \;\&\; R_{\mathrm{c}} \\ C = C_{\mathrm{g}} \cup C_{\mathrm{c}} \cup \{\widehat{\tau}_{\mathrm{g}} = \textbf{[}\alpha\textbf{]}^{\beta_1}\} \quad R = R_{\mathrm{g}} \cup R_{\mathrm{c}} \cup \{\beta_1 \subseteq \{\textbf{(}_\textbf{:}\beta_2\textbf{)}\}\}\end{array}}{\widehat{\Gamma} \vdash \textbf{case } g \textbf{ of } \{\textbf{(}h\textbf{:}t\textbf{)} \to e_{\mathrm{c}}\} : \widehat{\tau}_{\mathrm{c}} \rightsquigarrow C \;\&\; R} \text{[T-Case-C]}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \vdash g : \widehat{\tau}_{\mathrm{g}} \rightsquigarrow C_{\mathrm{g}} \;\&\; R_{\mathrm{g}} \quad \alpha, \beta_1, \beta_2 \text{ fresh} \\ \widehat{\Gamma} \vdash e_{\mathrm{n}} : \widehat{\tau}_{\mathrm{n}} \rightsquigarrow C_{\mathrm{n}} \;\&\; R_{\mathrm{n}} \quad \widehat{\Gamma}, h : \alpha,\, t : \textbf{[}\alpha\textbf{]}^{\beta_2} \vdash e_{\mathrm{c}} : \widehat{\tau}_{\mathrm{c}} \rightsquigarrow C_{\mathrm{c}} \;\&\; R_{\mathrm{c}} \\ C = C_{\mathrm{g}} \cup C_{\mathrm{n}} \cup C_{\mathrm{c}} \cup \{\widehat{\tau}_{\mathrm{g}} = \textbf{[}\alpha\textbf{]}^{\beta_1}, \widehat{\tau}_{\mathrm{n}} = \widehat{\tau}_{\mathrm{c}}\} \\ R = R_{\mathrm{g}} \cup R_{\mathrm{n}} \cup R_{\mathrm{c}} \cup \{\beta_1 \subseteq \{\textbf{[]}, \textbf{(}_\textbf{:}\beta_2\textbf{)}\}\}\end{array}}{\widehat{\Gamma} \vdash \textbf{case } g \textbf{ of } \{\textbf{[]} \to e_{\mathrm{n}}; \textbf{(}h\textbf{:}t\textbf{)} \to e_{\mathrm{c}}\} : \widehat{\tau}_{\mathrm{n}} \rightsquigarrow C \;\&\; R} \text{[T-Case-NC]}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \vdash e_1 : \widehat{\tau}_1 \rightsquigarrow C_1 \;\&\; R_1 \quad \mathit{gen}(\widehat{\Gamma}, \widehat{\tau}_1, C_1, R_1) = \widehat{\sigma} \\ \widehat{\Gamma}, x : \widehat{\sigma} \vdash e_2 : \widehat{\tau}_2 \rightsquigarrow C_2 \;\&\; R_2\end{array}}{\widehat{\Gamma} \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \widehat{\tau}_2 \rightsquigarrow C_2 \;\&\; R_2} \text{[T-Let]}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \vdash \lambda f.\, e_1 : \widehat{\tau}_{\mathrm{r}} \rightsquigarrow C_1 \;\&\; R_1 \quad \mathit{gen}(\widehat{\Gamma}, \widehat{\tau}_1, C_1 \cup \{\widehat{\tau}_1 \to \widehat{\tau}_1 = \widehat{\tau}_{\mathrm{r}}\}, R_1) = \widehat{\sigma} \\ \widehat{\Gamma}, x : \widehat{\sigma} \vdash e_2 : \widehat{\tau}_2 \rightsquigarrow C_2 \;\&\; R_2\end{array}}{\widehat{\Gamma} \vdash \textbf{let } f = e_1 \textbf{ in } e_2 : \widehat{\tau}_2 \rightsquigarrow C_2 \;\&\; R_2} \text{[T-LetRec]}$$

Figure 4.16: Type system (part 2)

4.6 Unification

To solve the constraint set C we use a unification algorithm $\widehat{\mathcal{U}}$ that either computes a substitution θ, such that θC contains only trivial equalities of the form $\widehat{\tau} = \widehat{\tau}$ or fails if the constraint set is unsolvable and the program is type incorrect.

$$
\begin{array}{rcl}
\widehat{\mathcal{U}} & : & \mathcal{P}(\textbf{Constr}) \to \textbf{Subst} \\
\widehat{\mathcal{U}}(\emptyset) & = & \mathrm{id} \\
\widehat{\mathcal{U}}(\{\alpha = \widehat{\tau}\} \cup C) & = & \textbf{let } \theta = \begin{cases} [\alpha \mapsto \widehat{\tau}] & \text{if } \alpha \notin \mathrm{ftv}(\widehat{\tau}) \\ \textbf{fail} & \text{otherwise} \end{cases} \\
 & & \textbf{in } \widehat{\mathcal{U}}(\theta C) \circ \theta \\
\widehat{\mathcal{U}}(\{\widehat{\tau} = \alpha\} \cup C) & = & \textbf{let } \theta = \begin{cases} [\alpha \mapsto \widehat{\tau}] & \text{if } \alpha \notin \mathrm{ftv}(\widehat{\tau}) \\ \textbf{fail} & \text{otherwise} \end{cases} \\
 & & \textbf{in } \widehat{\mathcal{U}}(\theta C) \circ \theta \\
\widehat{\mathcal{U}}(\{\widehat{\tau}_1 \xrightarrow{\varphi_1} \widehat{\tau}_2 = \widehat{\tau}'_1 \xrightarrow{\varphi_2} \widehat{\tau}'_2\} \cup C) & = & \textbf{let } \theta = \mathcal{V}(\varphi_1 = \varphi_2) \\
 & & \textbf{in } \widehat{\mathcal{U}}(\theta\{\widehat{\tau}_1 = \widehat{\tau}'_1, \widehat{\tau}_2 = \widehat{\tau}'_2\} \cup \theta C) \\
\widehat{\mathcal{U}}(\{\textbf{Bool}^{\varphi_1} = \textbf{Bool}^{\varphi_2}\} \cup C) & = & \textbf{let } \theta = \mathcal{V}(\varphi_1 = \varphi_2) \\
 & & \textbf{in } \widehat{\mathcal{U}}(\theta C) \circ \theta \\
\widehat{\mathcal{U}}(\{\textbf{Int}^{\varphi_1} = \textbf{Int}^{\varphi_2}\} \cup C) & = & \textbf{let } \theta = \mathcal{V}(\varphi_1 = \varphi_2) \\
 & & \textbf{in } \widehat{\mathcal{U}}(\theta C) \circ \theta \\
\widehat{\mathcal{U}}(\{\textbf{(}\widehat{\tau}_1, \widehat{\tau}_2\textbf{)}^{\varphi_1} = \textbf{(}\widehat{\tau}'_1, \widehat{\tau}'_2\textbf{)}^{\varphi_2}\} \cup C) & = & \textbf{let } \theta = \mathcal{V}(\varphi_1 = \varphi_2) \\
 & & \textbf{in } \widehat{\mathcal{U}}(\theta\{\widehat{\tau}_1 = \widehat{\tau}'_1, \widehat{\tau}_2 = \widehat{\tau}'_2\} \cup \theta C) \\
\widehat{\mathcal{U}}(\{\textbf{[}\widehat{\tau}\textbf{]}^{\varphi_1} = \textbf{[}\widehat{\tau}'\textbf{]}^{\varphi_2}\} \cup C) & = & \textbf{let } \theta = \mathcal{V}(\varphi_1 = \varphi_2) \\
 & & \textbf{in } \widehat{\mathcal{U}}(\theta\{\widehat{\tau} = \widehat{\tau}'\} \cup \theta C) \\
\text{otherwise} & = & \textbf{fail}
\end{array}
$$

The unification algorithms $\widehat{\mathcal{U}}$ relies on a unification algorithm $\mathcal{V}$ that unifies annotations.

$$
\mathcal{V}(\varphi_1 = \varphi_2) = \begin{cases} \mathrm{id} & \text{if } \varphi_1 = \varphi_2 \\ [\varphi_1 \mapsto \varphi_2] & \text{otherwise} \end{cases}
$$

This unification algorithm $\mathcal{V}$ is straightforward, because the constraints in C only concern simply annotated types—remember that these are annotated types where the annotations can only be annotation variables and nothing else.

The resulting substitution can contain both mappings $[\alpha \mapsto \widehat{\tau}]$ from type variables to annotated types and mappings $[\beta_1 \mapsto \beta_2]$ from annotation variables to annotation variables.

Chapter 5

Constraint Solving

5.1 Overview

The constraint generation phase of the analysis generates two constraint sets: a constraint set C containing equality constraints between simply annotated types and a constraint set R containing subtype constraints between the annotations. As we have seen the constraint set C can be solved by a slightly modified, but fairly traditional unification algorithm, resulting in a substitution θ_C from type variables to annotated types and from annotation variables to annotation variables.

The first step in solving the constraints set R is to apply the substitution θ_C to it, making sure any annotation variables that have been unified while solving C are also unified in R.

The constraints in R can have various forms, for example: producing a **True** will introduce a constraint

$$\{\textbf{True}\} \subseteq \beta$$

for a fresh annotation variable β. Pattern matching on an empty list (and nothing else) will generate a constraint

$$\beta \subseteq \{\textbf{[]}\}$$

Recursive data types, such as lists, give rise to a more complicated form of *transitive constraints*. For example, pattern matching on a list will generate a constraint

$$\beta_1 \subseteq \{\textbf{[]}, \textbf{(}_\textbf{:}\beta_2\textbf{)}\}$$

where β_1 is the annotation variable on the scrutinee of the case-expression and β_2 the annotation on the tail of the list. Intuitively, if we learn something about β_1 this might influence what we know about β_2 and vice versa.

To solve the constraint set R we need to find for each annotation variable β on a type τ an (as large as possible) interval $I = (l, u) \in \mathcal{L}^2$ of the lattice $\mathcal{L}$ abstracting the underlying type of the annotation variable that is consistent with the constraint set. As the starting interval we take the whole of the lattice $(\bot_{\mathcal{L}}, \top_{\mathcal{L}})$. We then apply a worklist algorithm that will have constraints of

the form $\{\mathbf{True}\} \subseteq \beta_1$ push the lowerbound l of the interval I_1 belonging to β_1 upwards and constraints of the form $\beta_2 \subseteq \{\mathbf{[]}\}$ the upperbound u of the interval I_2 belonging to β_2 downwards. Transitive constraints such as $\beta_1 \subseteq \{\mathbf{[]}, \mathbf{(}_\mathbf{:}\beta_2\mathbf{)}\}$ will affect the upper- and lowerbounds of both β_1 and β_2.

5.2 Dependency Analysis

Transitive constraints—such as $\beta_1 \subseteq \beta_2$—which contain annotation variables on both the left- and right-hand side of the constraint introduce dependencies between constraints.

The dependency analysis phase of the analysis determines the dependencies between subset constraints and annotation variables by building two dependency graphs. One between annotation variables and one from annotation variables to subset constraints. These two graphs are used when determining the input-independent constraints and by the worklist algorithm described next.

5.3 Worklist Algorithm

5.3.1 Generic Worklist Algorithm

The constraints solver for the constraint set R is built on top of a generic worklist algorithm:

$$\begin{array}{l} \mathit{worklist} :: (a \to b \to ([\,a\,], b)) \to b \to [\,a\,] \to b \\ \mathit{worklist}\ _\ r\ [\,] \qquad\quad = r \\ \mathit{worklist}\ f\ r\ (x : xs) = \mathbf{let}\ (ys, r') = f\ x\ r\ \mathbf{in}\ \mathit{worklist}\ f\ r'\ (xs +\!\!+\ ys) \end{array}$$

The function argument f takes a takes a constraint $x :: a$ and intermediate result $r :: b$ and returns a new intermediate result $r' :: b$ and an additional set of constraints $ys :: [\,a\,]$ the worklist algorithm has to add to its current working set.

In our application of this generic worklist algorithm these additional constraints will be constraints that have been previously processed, but for which the dependency checker has determined they may need to be looked at again, because one or more of their dependent variables have been updated.

While the generic worklist algorithm does not necessarily terminate, it will in our application. Abstract values are modelled by lattices satisfying the ascending chain condition, meaning the dependency analysis will eventually stop feeding constraints into the worklist when all dependent variables have reached their fixed points.

5.3.2 Resolving Individual Constraints

The function we pass as the argument f to $\mathit{worklist}$ will—after having assisted with dependency checking and updating—invoke the function $\mathit{resolveConstr}$,

which updates the intermediate result by making it consistent with an individual constraint.

In its most generic shape such a constraint will be of the form

$$\beta_1 \cup \beta_2 \cup ... \cup \beta_m \cup \{K_1, K_2, ..., K_n\} \subseteq \beta'_1 \cup \beta'_2 \cup ... \cup \beta'_{m'} \cup \{K'_1, K'_2, ..., K'_{n'}\}$$

where $\beta_i, \beta_{i'}$ are annotation variables and $K_j, K_{j'}$ are elements of some lattice π_τ—they can be thought of as constructors—possibly containing more annotation variables within them.

In order to make the intermediate result consistent with this constraint we may need to do two things:

1. Raise the lowerbound of refinement variables present—either directly as one of the $\beta_{i'}$s or inside one of the constructors $K'_{j'}$—in the right-hand side of the constraint;
2. Lower the upperbound of refinement variables present in the left-hand side of the constraint.

In order to accomplish the first $resolveConstr$ will:

1. Instantiate all refinement variables on the left-hand side of the constraint—including those inside constructors—with the current lowerbound recorded in the intermediate result. Call this L. (For constructors we assume a widening operator is applied during instantiation if this is required to maintain the ascending chain condition of their corresponding lattice.)
2. For each refinement variable $\beta'_{i'}$ on the right-hand side $resolveConstr$ will update $\theta_{\mathrm{L}}(\beta'_{i'})$, the lowerbound for that variable as recorded in the intermediate result, to $\theta_{\mathrm{L}}(\beta'_{i'}) \sqcup L$.
3. For each constructor $K'_{j'}$ on the right-hand side $resolveConstr$ will make new constraints
$$\Pi^p_{K_{j'}}(L) \subseteq \Pi^p_{K_{j'}}(\{K'_{j'}\})$$
where Π^p_K projects out the pth field of a K-constructor or gives an empty result if its input is not a K-constructor. This whole procedure will then be recursively applied to the newly generated constraints.

In order to accomplish the second we can apply the same procedure *mutatis mutandis*; in particular the updated value of the intermediate result in the second step should now become $\theta_{\mathrm{U}}(\beta_i) \sqcap R$.

Example Assume $\theta(\beta_1) = (\{\mathbf{[]}, \mathbf{(_:[])}, \mathbf{(_:(_:[]))}\}, \top)$ and $\theta(\beta_2) = (\bot, \top)$. Given the constraint

$$\{\beta_1\} \subseteq \{\mathbf{(_:}\beta_2\mathbf{)}\} \tag{5.1}$$

$resolveConstr$ will instantiate the left-hand side to

$$\{\mathbf{[]}, \mathbf{(_:[])}, \mathbf{(_:(_:[]))}\} \subseteq \{\mathbf{(_:}\beta_2\mathbf{)}\} \tag{5.2}$$

As there is both a **[]**-constructor and a **(** $_$ **:** $\cdot$ **)**-constructor on the right-hand side, *resolveConstr* will partition the constraint into two constraints

$$\{[]\} \subseteq \emptyset \tag{5.3}$$
$$\{(_:[]), (_:(_:[]))\} \subseteq \{(_:\beta_2)\} \tag{5.4}$$

As the right-hand side of (5.3) is empty this constraint is discarded. Both left and right-hand side of (5.4) have top-level **(** $_$ **:** $\cdot$ **)**-constructors. We project out their fields, resulting in the constraint

$$\{[], (_:[])\} \subseteq \{\beta_2\} \tag{5.5}$$

To solve this constraint *resolveConstr* will be recursively called on it. As there is only a refinement variable on the right-hand side of (5.5) *resolveConstr* will update $\theta_{\mathrm{L}}(\beta_2)$ to

$$\bot \sqcup \{(_:[]), (_:(_:[]))\} = \{(_:[]), (_:(_:[]))\}$$

We have now "pushed up" the lowerbound of β_2. Next, we "push down" the upperbound of β_1. We begin by instantiating the right-hand side of (5.1) to

$$\{\beta_1\} \subseteq \{(_:[]), (_:(_:[])), ..., (_:...(_:[])...), (_:...(_:\star)...)\} \tag{5.6}$$

We immediately end up with a constraint of the form $\beta \subseteq \{...\}$, so we do not need to recurse and can update $\theta_{\mathrm{U}}(\beta_1)$ to

$$\begin{aligned} &\top \sqcap \{(_:[]), (_:(_:[])), ..., (_:...(_:[])...), (_:...(_:\star)...)\} \\ &\qquad = \{(_:[]), (_:(_:[])), ..., (_:...(_:[])...), (_:...(_:\star)...)\} \end{aligned}$$

The resulting intervals are now

$$\begin{aligned} \theta(\beta_1) &= (\{[], (_:[]), (_:(_:[]))\}, \{(_:[]), (_:(_:[])), ..., (_:...(_:\star)...)\}) \\ \theta(\beta_2) &= (\{(_:[]), (_:(_:[]))\}, \top) \end{aligned}$$

As the lowerbound of β_1 is no longer contained in its upperbound—the lowerbound includes a **[]**-constructor, while the upperbound does not—we have detected a possible pattern match failure.

Chapter 6

Evaluation

6.1 Examples

6.1.1 Higher-order functions

Given the program

$$\begin{aligned}
main = \lambda b \to \lambda phi \to\ & \mathbf{if}\ b \\
& \mathbf{then\ if}\ phi\ 42\ \mathbf{then}\ 100\ \mathbf{else}\ 200 \\
& \mathbf{else}\ \ \mathbf{if}\ phi\ 43\ \mathbf{then}\ 300\ \mathbf{else}\ 400
\end{aligned}$$

the analysis will infer the type $\forall\alpha\beta.\ \{\alpha \subseteq \{\mathbf{False}, \mathbf{True}\}, \beta \subseteq \{\mathbf{False}, \mathbf{True}\}\} \Rightarrow (\mathbf{Bool}^{\alpha} \xrightarrow{\top} ((\mathbf{Int}^{\{+\}} \xrightarrow{\top} \mathbf{Bool}^{\beta}) \xrightarrow{\top} \mathbf{Int}^{\{+\}}))$.

6.1.2 Detecting pattern match failures

Given the program

$$\begin{aligned}
main = \mathbf{let}\ & f = \lambda b \to \lambda phi \to \mathbf{if}\ b \\
& \qquad\qquad \mathbf{then\ if}\ phi\ [42]\ \mathbf{then}\ 100\ \mathbf{else}\ 200 \\
& \qquad\qquad \mathbf{else}\ \ \mathbf{if}\ phi\ [43]\ \mathbf{then}\ 300\ \mathbf{else}\ 400 \\
& g = \lambda x \to \mathbf{case}\ x\ \mathbf{of} \\
& \qquad\qquad [\,] \to \mathit{True} \\
& h = \lambda x \to \mathbf{case}\ x\ \mathbf{of} \\
& \qquad\qquad (a : as) \to \mathit{True} \\
\mathbf{in}\ & f\ \mathit{True}\ g
\end{aligned}$$

the analysis will report that a pattern match failure occurs. Changing $f\ \mathit{True}\ g$ into $f\ \mathit{True}\ h$ will make the pattern match failure—and the reporting of it by the analysis—go away.

6.1.3 *filter*

We given the program

$$
\begin{array}{l}
main = \textbf{let}\ filter = \lambda p \rightarrow \lambda ys \rightarrow \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \qquad \rightarrow [\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad (x : xs) \rightarrow \textbf{let}\ g = p\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if}\ g\ \textbf{then}\ x : filter\ p\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else} \quad filter\ p\ xs \\
\quad \textbf{in}\ filter\ (\lambda x \rightarrow x)\ [\,True, False\,]
\end{array}
$$

the analysis will infer the type $\mathbf{[Bool}^{\{\mathbf{False},\mathbf{True}\}}\mathbf{]}^{\top}$.

6.2 Limitations

6.2.1 Context-insensitivity

The analysis is *context-insensitive*. Given the program:

$$
\begin{array}{l}
main = \textbf{let}\ f = \lambda b \rightarrow \textbf{let}\ tail = \lambda xss \rightarrow \textbf{case}\ xss\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (x : xs) \rightarrow xs \\
\qquad\qquad\qquad\qquad \textbf{in}\ tail\ (\textbf{if}\ b\ \textbf{then}\ [\,]\ \textbf{else}\ [1, 2, 3]) \\
\qquad\quad \textbf{in}\ f\ False
\end{array}
$$

a pattern match failure will be reported. One does not occur practice, as the value **False** passed to b will prevent an empty list being returned from the if-then-else expression to $tail$. Section 9.2.2 discusses how to alleviate this limitation.

6.2.2 Structurally recursive functions

Our analysis suffer from another limitation: the inferred type for a number of structurally recursive function such as map and $foldr$ is not precise enough.

While the inferred type for $tail$

$$\forall \alpha, (\beta \subseteq \mathbf{(}\,_\mathbf{:}\gamma\mathbf{)}).\ \mathbf{[}\alpha\mathbf{]}^{\beta} \rightarrow \mathbf{[}\alpha\mathbf{]}^{\gamma}$$

expresses everything we can say about the function—the values of the elements in the list are preserved and the lenght is output list is one element shorten than the input—this is not the case for map:

$$\forall \alpha, \beta, \gamma.\ (\alpha \rightarrow \beta) \rightarrow \mathbf{[}\alpha\mathbf{]}^{\gamma} \rightarrow \mathbf{[}\beta\mathbf{]}^{\top}$$

Our constraint solver is not able to infer from the constraints generated by the pattern matching on the input list and subsequent cons of the modified head and tail that map preserves the length of the list. I.e., we would have expected the type

$$\forall \alpha, \beta, \gamma.\ (\alpha \rightarrow \beta) \rightarrow \mathbf{[}\alpha\mathbf{]}^{\gamma} \rightarrow \mathbf{[}\beta\mathbf{]}^{\gamma}$$

This will be a severe limitation in practice, as even innocent functions such as $head\ (map\ id\ [1\mathinner{..}9])$ will result in a pattern match warning.

Chapter 7

Implementation

An implementation of the analysis has been implemented in Haskell. It has been built on top of the `haskell-src-exts` library, which is used for both parsing of the input and representation of the abstract syntax tree.

Refinements and other extensions on Haskell's type system that are required for the analysis are represented using the annotation fields available in the abstract syntax of `Language.Haskell.Exts.Annotated`.

An overview of the modules that comprise the implementation can be found in Figure 7.1.

Additionally, we have build a collection of example functions that where used for regression testing purposes during the development of the analysis and its implementation.

A demonstration of the implementation can be found in Appendix A.

Module	Lines	Description
`Main`	134	Main
`TypeInference`	391	Constraint generation
`PatternTyping`	153	Typing of patterns
`Solver`	263	Constraint solving
`Common`	342	Substitution and unification
`Refinements`	152	Refinements (representation and operations)
`Abstract.*`	355	Abstract values (unit, booleans, integers, lists)
`Util.*`	48	Miscellaneous

Figure 7.1: Module overview

Chapter 8

Related Work

8.1 Neil Mitchell's *Catch*

Neil Mitchel's *Catch* ("CAse Totality CHecker") [9] is a tool specialized in finding potential pattern match failures in Haskell programs.

8.1.1 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:

$$
\begin{aligned}
safeTail\ xs = \textbf{case}\ null\ xs\ \textbf{of} \\
True \rightarrow [\,] \\
False \rightarrow tail\ xs
\end{aligned}
$$

the computed precondition will be:

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

stating the necessary precondition for $null\ xs$ not to crash must 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 $xs \in \{(:)\}$. Substituting these into our precondition we find

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

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

which simplifies to

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

The subexpression $null\ xs \in \{True\}$ 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 $xs \in \{[]\}$. Substituting, we find

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

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

8.1.2 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 $xs \in (tl^{*} \cdot hd \rightsquigarrow \{(:)\})$. It should be interpreted as "after applying zero or more tls to xs and then applying a hd we should find a $(:)$ constructor."

According 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:

$$\begin{array}{l} \{[], (:)(\{(:)\ Any\} \star \{[], (:)\ Any\})\} \\ \quad \star \\ \{[], (:)(\{(:)\ Any\} \star \{[], (:)\ Any\})\} \end{array}$$

Any is a wildcard that matches any constructor.

8.1.3 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

t	::=	$\{x \vert p\}$	Predicate
	$\vert$	$x : t_1 \to t_2$	Dependent function
	$\vert$	(t_1, t_2)	Constructor
	$\vert$	Any	Polymorphic "Any"

Figure 8.1: Syntax of Xu's contracts

8.2 Static Contract Checking

8.2.1 Overview

Dana Xu's work on static contract checking for Haskell [19] (including the ESC/Haskell system [18]) 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 $\neg\ (null\ s)$ 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. 8.1). 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 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 $\{\, x \mid True \,\}$. 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.

8.2.2 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.

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

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.

The verification of the contracts is based on a translation of the contracts into Findler–Felleisen wrappers [2], 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** $x{*}x \geqslant 0$ **of** $\{$ *True* $\rightarrow$ ...; *False* $\rightarrow$ 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."

8.2.3 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.

8.3 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) $\mathcal{L}$ [17].

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 ($n \leqslant m$). This is expressed as the dependent sum $[n :: Nat \mid n \leqslant m]$. 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 precondition 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 $m+n = n+m$ to an ILP solver and conclude the constraint is satisfiable.

8.3.1 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 $n \leqslant m$—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 type and the pattern matching happening in the definition of *head* is much more direct.

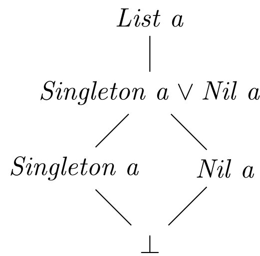

Figure 8.2: A lattice of *List* and its *rectype*s.

8.4 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* that can only contain negative numbers.

Freeman and Pfenning [4, 3] 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 *rectype*s. Besides defining a recursive type for lists:

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

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 [5]—a finite lattice of refinement types for *List a* given in Figure 8.2.

8.4.1 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 that 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.

8.4.2 Constructors and Pattern Matching

To infer refinement types more accurate types need to be given to constructors. This is done by using a restricted[4] 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
```

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 [13, 15]. 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)
```

8.5 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 [7] 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

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

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 we might also be able to follow such an approach when analyzing case-statements in our pattern-match analysis.

Chapter 9

Conclusions and Further Research

9.1 Conclusions

Using our analysis we are able to statically detect pattern match failures—or the absence of them, where a more naive totality checker would have given us spurious warnings. Using a type and effect system to do pattern match analysis certainly appears to be a viable approach.

As we demonstrated in Chapter 6, there are a number of limitations to our current analysis that may result in reduced accuracy, that is, an increase in the number of false positives. We will present a number of suggestion for further research that may help overcome these limitations.

9.2 Improving Precision

9.2.1 Intersection Types

The analysis sketched so far falls short if we want to analyze functions using common operators such as $(+)$ or $(\leqslant)$. To see the problem here we should ask ourself what type we would like to assign to such expressions. The operator $(\leqslant)$ can take any integer arguments—which does not cause a pattern match failure when evaluated—and returns either **True** or **False** without causing a pattern match failure itself. The most reasonable annotated type for this operator thus seems to be:

$$(\leqslant) : \mathbf{Int}^{\top} \rightarrow \mathbf{Int}^{\top} \rightarrow \mathbf{Bool}^{\top}$$

Using the operator in the following situation:

$$f\ x = \mathbf{if}\ 0 \leqslant 1\ \mathbf{then}\ x\ \mathbf{else}\ crash$$

will produce a spurious warning about a pattern-match error occurring. The guard of the if-then-else expression will always evaluate to **True**, but be-

cause of the type we have assigned to $(\leqslant)$ the analyzer will believe it might evaluate to **False** as well.

The root of the problem is that we can assign several different valid refinement types to $(\leqslant)$:

$$\begin{array}{lcl}(\leqslant) & : & \mathbf{Int}^{\{-\}} \to \mathbf{Int}^{\top} \to \mathbf{Bool}^{\{\mathbf{True}\}} \\ (\leqslant) & : & \mathbf{Int}^{\{0,+\}} \to \mathbf{Int}^{\{-\}} \to \mathbf{Bool}^{\{\mathbf{False}\}} \\ (\leqslant) & : & ... \end{array}$$

neither of which is a *principal type*—that is having as its refinements a strictly minimal set of values on types in a covariant positions and a stricly maximal set of values on types in a contravariant position.

A solution would be to assign this expression a more accurate *intersection type* (see Section 8.4.1). For example, when using a sign abstraction for integers:

$$(\leqslant) : \mathbf{Int}^{\{-\}} \to \mathbf{Int}^{\top} \to \mathbf{Bool}^{\{\mathbf{True}\}} \wedge \mathbf{Int}^{\{\mathbf{0},+\}} \to \mathbf{Int}^{\{-\}} \to \mathbf{Bool}^{\{\mathbf{False}\}} \wedge ...$$

Note that—for our purposes at least—if we assign some expression e the intersection type $\widehat{\tau}_1 \wedge \widehat{\tau}_2 \wedge ... \wedge \widehat{\tau}_n$ then any two annotated types $\widehat{\tau}_i$ and $\widehat{\tau}_j$ will have the same underlying type ($\lfloor\widehat{\tau}_i\rfloor = \lfloor\widehat{\tau}_j\rfloor$) and the two types will thus only differ in their annotations.

To represent intersection types we therefore only have to keep track of the various possible assignments of the annotation variables. We can integrate this concept neatly into our type system by modifying the typing relation $\widehat{\Gamma} \vdash e : \widehat{\tau} \rightsquigarrow C \;\&\; R$ to a typing relation

$$\widehat{\Gamma} \vdash e : \widehat{\tau} \rightsquigarrow C \;\&\; \bar{R}$$

where $\bar{R}$ is a *set of constraint sets* over the refinements, expressing the various possible assignments of annotations; one constraint set for each components of the intersection type.

All typing judgments will need to be modified to properly propagate the sets of constraint sets. During simplification any constraint sets that are inconsistent can be removed.

9.2.2 Implication Constraints

In k-CFA data-flow is used to improve the precision of the control-flow analysis [16]. Conversely, we can try to improve the precision of our data-flow anlysis by considering the control-flow.

Consider again the program we saw in Section 3.2:

$$\begin{array}{l} main\ b\ f = \mathbf{if}\ b\ \mathbf{then} \\ \qquad\qquad\quad \mathbf{if}\ f\ 42\ \mathbf{then}\ 100\ \mathbf{else}\ 200 \\ \qquad\qquad \mathbf{else} \\ \qquad\qquad\quad \mathbf{if}\ f\ 43\ \mathbf{then}\ 300\ \mathbf{else}\ 400 \end{array}$$

We assigned this the type:

$$\mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}} \to (\mathbf{Int}^{\{42,43\}} \to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\{100,200,300,400\}} \qquad (9.1)$$

If we have more knowledge about the exact value of b, e.g. b being always **True** or **False**, we can infer the more precise types

$$\mathbf{Bool}^{\{\mathbf{True}\}} \to (\mathbf{Int}^{\{42\}} \to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\{100,200\}} \tag{9.2}$$

and

$$\mathbf{Bool}^{\{\mathbf{False}\}} \to (\mathbf{Int}^{\{43\}} \to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\{300,400\}} \tag{9.3}$$

This knowledge can be expressed using *implication constraints*

$$\rho_1 \subseteq \rho_2 \models \rho_3 \subseteq \rho_4$$

where the constraint $\rho_3 \subseteq \rho_4$ on the right-hand side only has to hold if the constraint $\rho_1 \subseteq \rho_2$ on the left-hand side holds.

The original type (9.1) and additional types (9.2) and (9.3) can together be represented as a single type using implication constraints:

$$\begin{aligned} \forall \alpha, \beta, \gamma.\ \{&\{\mathbf{True}\} \subseteq \alpha \models \{42\} \subseteq \beta, \{\mathbf{True}\} \subseteq \alpha \models \{100, 200\} \subseteq \gamma, \\ &\{\mathbf{False}\} \subseteq \alpha \models \{43\} \subseteq \beta, \{\mathbf{False}\} \subseteq \alpha \models \{300, 400\} \subseteq \gamma, \\ &\alpha \subseteq \{\mathbf{True}, \mathbf{False}\}\} \Rightarrow \mathbf{Bool}^{\alpha} \to (\mathbf{Int}^{\beta} \to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\gamma} \end{aligned}$$

It is not yet clear if implications constraints would be orthogonal to intersection types. We could represent the type equivalently as:

$$\begin{aligned} &\mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}} \to (\mathbf{Int}^{\{42,43\}} &&\to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\{100,200,300,400\}} \\ \wedge\ &\mathbf{Bool}^{\{\mathbf{True}\}} \to (\mathbf{Int}^{\{42\}} &&\to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\{100,200\}} \\ \wedge\ &\mathbf{Bool}^{\{\mathbf{False}\}} \to (\mathbf{Int}^{\{43\}} &&\to \mathbf{Bool}^{\{\mathbf{True},\mathbf{False}\}}) \to \mathbf{Int}^{\{300,400\}} \end{aligned}$$

9.2.3 Set-based Constraints

It is worth investigating if the generated constraints can be formulated in terms of *set-based constraints* [1, 6]. More elaborate solvers (such as BANE) have been developed for this type of constraints and may give more accurate results than our current solver.

9.3 Improving Applicability

9.3.1 Handling $undefined$

While our analysis can detect pattern match failures due to missing alternatives in case-expressions, we would also like to detect failures occurring due to evaluating $undefined$ or $error$.

A straightforward approach would seem to add an additional value $\bot$—not to be confused with the least element of a lattice—to the language of refinements φ, representing a value the diverges if evaluated. The type rules for constructs that perform pattern matching already, such as T-If and T-App, already ensure that such values would not be allowed as the scrutinee of a pattern match.

There are still two hurdles that need to be overcome, however:

1. While the type constructors for lists, tuples and functions already come prepared with an annotation in which the diverging value $\perp$ can be stored, $undefined$ itself has the polymorphic type $\forall \alpha.\ \alpha$. Type variables, however, do not come equipped with an annotation, so we cannot assign it the obvious type $\forall \alpha.\ \alpha^{\{\perp\}}$.

2. Without context-sensitivity in the type rules T-If and T-Case, they will simply propagate the diverging value forward if it occurs in any one of the branches of an if-then-else or case-expression. Adding context-sensitivity as described Section 9.2.2 would be a prerequisite.

Appendix A

Demonstration of Implementation

A.1 Input

```
main = let f = \b -> \phi -> if b
                              then if phi [42] then 100 else 200
                              else if phi [43] then 300 else 400
           g = \x -> case x of
                          []     -> True
           h = \x -> case x of
                          (a:as) -> True
        in f True g
```

A.2 Output

```
"== [1] REFINEMENT TYPES ==============================="
?148
-- [2] --------------------
[?0 = [?15]@?27 | ?
,?1 = [?46]@?82 | ?
,?4 = Bool@?193 | ?
,Int@?247 = Int@?274 | ?
,?2299 = Bool@?463 | ?
,Int@?247 = Int@?355 | ?
,?58 = (?1003 -> ?2299)@?3595 | app1
,?1003 = [?571]@?3163 | app2
,Int@?4459 = ?571 | ?
,?2326 = Bool@?490 | ?
,Int@?274 = Int@?382 | ?
,?58 = (?1030 -> ?2326)@?3622 | app1
,?1030 = [?598]@?3190 | app2
```

```
,Int@?4486 = ?598 | ?
,?418 = (?67 -> ?148)@?229 | app1
,?67 = (?1 -> Bool@?28)@?10 | app2
,(?4 -> (?58 -> Int@?247)@?85)@?31 = (?175 -> ?418)@?661 | app1
,?175 = Bool@?94 | app2
]
-- [3] --------------------
λ ⊆ ?3
?27 ⊆ {_:?39}
{True} ⊆ ?12
λ ⊆ ?10
?82 ⊆ {[]}
{True} ⊆ ?28
λ ⊆ ?31
λ ⊆ ?85
?193 ⊆ {False,True}
?463 ⊆ {False,True}
λ ⊆ ?3595
{_:[]} ⊆ ?3163
{+} ⊆ ?4459
{+} ⊆ ?247
{+} ⊆ ?355
?490 ⊆ {False,True}
λ ⊆ ?3622
{_:[]} ⊆ ?3190
{+} ⊆ ?4486
{+} ⊆ ?274
{+} ⊆ ?382
λ ⊆ ?229
λ ⊆ ?661
{True} ⊆ ?94
-- [4] --------------------
?0 ↦ [?15]@?27
?1 ↦ [Int@?4459]@?82
?1003 ↦ [Int@?4459]@?82
?1030 ↦ [Int@?4459]@?82
?148 ↦ Int@?382
?175 ↦ Bool@?94
?2299 ↦ Bool@?28
?2326 ↦ Bool@?28
?4 ↦ Bool@?94
?418 ↦ (([Int@?4459]@?82 -> Bool@?28)@?10 -> Int@?382)@?229
?46 ↦ Int@?4459
?571 ↦ Int@?4459
?58 ↦ ([Int@?4459]@?82 -> Bool@?28)@?10
?598 ↦ Int@?4459
?67 ↦ ([Int@?4459]@?82 -> Bool@?28)@?10
```

```
. .[5]. . . . . . . . . . .
?193 ↦ ?94
?247 ↦ ?382
?274 ↦ ?382
?31 ↦ ?661
?3163 ↦ ?82
?3190 ↦ ?82
?355 ↦ ?382
?3595 ↦ ?10
?3622 ↦ ?10
?4486 ↦ ?4459
?463 ↦ ?28
?490 ↦ ?28
?85 ↦ ?229
--------------------------
Sanity check: PASSED
-= [6] =-=-=-=-=-=-=-=-=-=-=-=-=-=-
Int@?382
-- [7] --------------------
λ ⊆ ?3
?27 ⊆ {_:?39}
{True} ⊆ ?12
λ ⊆ ?10
?82 ⊆ {[]}
{True} ⊆ ?28
λ ⊆ ?661
λ ⊆ ?229
?94 ⊆ {False,True}
?28 ⊆ {False,True}
λ ⊆ ?10
{_:[]} ⊆ ?82
{+} ⊆ ?4459
{+} ⊆ ?382
{+} ⊆ ?382
?28 ⊆ {False,True}
λ ⊆ ?10
{_:[]} ⊆ ?82
{+} ⊆ ?4459
{+} ⊆ ?382
{+} ⊆ ?382
λ ⊆ ?229
λ ⊆ ?661
{True} ⊆ ?94
-= [8] =-=-=-=-=-=-=-=-=-=-=-=-=-=-
Lower: {[?27 ~> {?27 ⊆ {_:?39}}]}
Upper: {[?39 ~> {?27 ⊆ {_:?39}}]}
. .[9]. . . . . . . . . . .
```

```
"?10 ~> {}"
"?12 ~> {}"
"?229 ~> {}"
"?27 ~> {\"?39\"}"
"?28 ~> {}"
"?3 ~> {}"
"?382 ~> {}"
"?39 ~> {\"?27\"}"
"?4459 ~> {}"
"?661 ~> {}"
"?82 ~> {}"
"?94 ~> {}"
-- [10] ------------------
Input vars          : fromList []
Input-dependent vars: fromList []
-- [11] ------------------
. .[12] . . . . . . . . . .
λ ⊆ ?3
?27 ⊆ {_:?39}
{True} ⊆ ?12
λ ⊆ ?10
?82 ⊆ {[]}
{True} ⊆ ?28
λ ⊆ ?661
λ ⊆ ?229
?94 ⊆ {False,True}
?28 ⊆ {False,True}
λ ⊆ ?10
{_:[]} ⊆ ?82
{+} ⊆ ?4459
{+} ⊆ ?382
{+} ⊆ ?382
?28 ⊆ {False,True}
λ ⊆ ?10
{_:[]} ⊆ ?82
{+} ⊆ ?4459
{+} ⊆ ?382
{+} ⊆ ?382
λ ⊆ ?229
λ ⊆ ?661
{True} ⊆ ?94
-- [13] ------------------
?10 ↦ (λ,λ)
?12 ↦ ({True},{False,True})
?229 ↦ (λ,λ)
?27 ↦ ({},{_:[],_:_:[],_:_:_:[],_:_:_:_:[],_:_:_:_:_:*})
?28 ↦ ({True},{False,True})
```

```
?3 ↦ (λ,λ)
?382 ↦ ({+},{-,0,+})
?39 ↦ ({},T_List)
?4459 ↦ ({+},{-,0,+})
?661 ↦ (λ,λ)
?82 ↦ ({_:[]},{[]})
?94 ↦ ({True},{False,True})
---------------------------
Sanity check: FAILED
Sanity check: FAILED
-- [14] -------------------
>> PATTERN-MATCH FAILURE DETECTED!!! <<
---[15] -------------------
forall. [] => Int@{+}
```

A.3 Legend

The individual sections of the output generated by our implementation of the analysis are:

1. The inferred type $\widehat{\tau}$ (without any substitutions having been applied to it.)
2. The inferred equality constraint set C (without any substitutions having been applied to it.)
3. The inferred subset constraint set R (without any substitutions having been applied to it.)
4. The substitution θ_C of type variables found by unifying C.
5. The substitution θ_C of annotation variables found by unifying C.
6. The inferred type $\widehat{\tau}$ with the substitution θ_C applied to it.
7. The inferred subset constraint set R with the substitution θ_C applied to it.
8. Dependency analysis of the constraints.
9. Dependency analysis of the annotation variables.
10. List of input and input-dependent variables.
11. Input-dependent constraints (R').
12. Input-independent constraints ($R - R'$).
13. Solution of $R - R'$ found by constraint solver.
14. Result of the analysis ("pattern match failure detected" or "no pattern match failure can occur at run-time.")
15. Inferred type after generalization.

The intermediate steps of the analysis are shown for the top-level binding only.

Bibliography

[1] A. Aiken. Introduction to set constraint-based program analysis. In *Science of Computer Programming*, pages 79–111, 1999.

[2] 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.

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

[4] 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.

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

[6] Nevin Heintze and Joxan Jaffar. Set constraints and set-based analysis. In *In Proceedings of the Workshop on Principles and Practice of Constraint Programming, LNCS 874*, pages 281–298. Springer-Verlag, 1994.

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

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

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

[10] 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.

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

[12] Simon Peyton Jones, 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.

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

[14] Benjamin C. Pierce. *Types and programming languages*, chapter 22.3: Constraint-Based Typing. MIT Press, Cambridge, MA, USA, 2002.

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

[16] Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.

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

[18] 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.

[19] 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.

Index

annotated type, 19, 24
annotation, 16, 25
as-pattern, 16

constraint
  equality, 25
  subset, 25
  transitive, 34
constraint typing relation, 28
constraint-based, 21
contravariant, 20
covariant, 20

generalization
  let-generalization, 18, 28
guard, 16

pattern matching
  nested, 16
poisoning, 18

refinement, 19
refinement type, 19

simply annotated type, 23
subeffecting, 16

type reconstruction
  algorithm, 18
type scheme, 24
type-directed, 21

underlying type, 19
  underlying type system, 16