LULEÅ UNIVERSITY OF TECHNOLOGY

2003:323 CIV

MASTER'S THESIS

Propagation of Location Information in Constrained Type Inference

PETER JONSSON
VIKTOR LEIJON

MASTER OF SCIENCE PROGRAMME

Department of Computer Science and Electrical Engineering
Division of Computer Science and Networking

2003:323 CIV • ISSN: 1402 - 1617 • ISRN: LTU - EX - - 03/323 - - SE

Propagation of location information in constrained type inference

Peter A. Jonsson Viktor Leijon

December 16, 2003

Abstract

The Timber type system is an extension of the classical Hindley-Milner type system, incorporating both qualified types and first class polymorphism. Since we also have subtyping, type errors in this system manifest themselves as unsatisfiable constraints rather than as non-unifiable types as in the Hindley-Milner system.

In this thesis, we present a theoretical model for propagating the source of a constraint so that it can be presented to the user in case of an error. We also present examples from an implementation of our system for the Timber compiler. We propagate the location information by adding annotations to the syntax tree and then preserve that information as we build the predicates and types needed for type checking.

We conclude that the approach to annotation taken here is one possible solution for the propagation of location information in constrained type inference.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

In a strongly typed language we are guaranteed that any type errors are found at compile time. We are assured that once the compiler/interpreter has accepted our program, it will not produce any type errors. In contrast, languages like C or Java make no such assurances. In these languages, the programmer can often accidentally produce type errors that will result either in unpredictable results or an exception.

In a strongly typed language with type inference, we are not only guaranteed that there will be no type errors, but we are also relieved of the responsibility for declaring the types for variables and functions ourselves. In a system with type inference, the compiler will deduce the type of variables through their use. The inference systems that we will consider in this thesis will be capable of producing the most general type possible for a program.

These mechanisms with type inference and lack of run-time type errors remove part of the burden from the programmer, but there are still compile-time type errors to deal with. A programmer must still be able to analyze and understand the error messages from the compiler. It is an unfortunate side effect that the more powerful the type system becomes and the more complex types it is capable of inferring, the more complicated the type errors become.

One such strongly typed language with type inference is Timber. Timber [BCJ+02] is a language created by Johan Nordlander among others, our advisor, and it is in the context of the Timber type system and the Timber compiler this thesis was created.

The topic of this thesis is "Propagation of location information in constrained type inference", but it might as well have been "Making sure the error location is known". We show, first formally and then in practice, how the elements of the type system can be annotated so that at the time of an error, the source of the error can be shown to the programmer. We propose a system of annotations for the type system, as well as give examples of how type errors look in a prototype system incorporating these annotations.

1.2 Type theory

1.2.1 History

In 1936 Alonzo Church invented a formal system named lambda calculus (λ calculus) [Bar88] and defined a computable function via this system. Independently in the same year Alan Turing invented a class of machines (later called Turing machines) and defined a computable function via this class. Turing also proved that both models are equally strong and define the same set of computable functions.

Both Church and Turing wanted to solve the Entscheidungsproblem [Wei03] (German for "decision problem"). Does there exist an algorithm for deciding whether or not a specific mathematical assertion has a proof? They both showed the answer to be negative.

Today's computers are conceptually Turing machines with random access registers. Imperative languages are based on the way a Turing machine operates by executing a sequence of statements.

Functional programming languages on the other hand are based on lambda calculus. Lambda calculus is a form of mathematical logic dealing primarily with functions and the application of functions to their arguments.

1.2.2 Hindley-Milner

Polymorphic types were already known as type schemes in combinatory logic [CFC58]. Hindley working with Curry extending the previous work introduced the idea of a principal type scheme [Hin69] which is the most general polymorphic type of an expression. They also showed that if a combinatorial term has a type, then it has a principal type. This was done by using a result of Robinson about the existence of most general unifiers in his unification algorithm [Rob65].

Independently from Hindley, Milner [Mil78] discovered the same ideas for the metalanguage ML [MTHM97] in the Edinburgh LCF system. Milner implemented the first polymorphic type checker and proved that the system was sound.

The existence of principal types means that a type inference algorithm will always compute a unique 'best' type for a program. An inference algorithm, utilizing unification, called W was introduced by Milner [Mil78, DM82] for use with ML. A practical implementation, in Miranda, of a type checker with type inference is described in [Han87]. Cardelli [Car87] gives a good overview of type checking.

The language Exp

Let x range over identifiers, $x \in Id$.

The language Exp of expressions e is then generated by the grammar in 1.1, extending implicitly typed lambda calculus by introducing the let construction. The language Exp is the language for the expressions which are given types by the type inference algorithm.

$$M, N ::= x \mid M\ N \mid \lambda x.M \mid let\ x = M\ in\ N. \tag{1.1}$$

$$\text{(Var)}\quad \frac{A(x)=\sigma}{P \mid A \vdash x : \sigma}$$

$$\text{(App)}\quad \frac{P \mid A \vdash M : \tau' \to \tau \qquad P \mid A \vdash N : \tau'}{P \mid A \vdash MN : \tau}$$

$$\text{(Lam)}\quad \frac{P \mid A, x : \tau \vdash M : \tau'}{P \mid A \vdash \lambda x{:}\tau.M : \tau \to \tau'}$$

$$\text{(Let)}\quad \frac{P \mid A \vdash M : \sigma \quad Q \mid A_x, x{:}\sigma \vdash N : \tau}{P \cup Q \mid A \vdash (\mathbf{let} = M \ \mathbf{in}\ N) : \tau}$$

$$\text{(Inst)}\quad \frac{P \mid A \vdash M : \forall t.\sigma}{P \mid A \vdash M : [\tau/t]\sigma}$$

$$\text{(Gen)}\quad \frac{P \mid A \vdash M : \sigma \qquad \alpha \notin fv(P, A)}{P \mid A \vdash M : \forall \alpha.\sigma}$$

Figure 1.1: The basic type inference rules for the Hindley-Milner system

Types and type schemes

The distinguishing feature of the Hindley-Milner system is the separation of types τ and type schemes σ. A type scheme is a type with some universally quantified variables.

There are types in the language Exp but they are only used as judgments about terms, never occurring inside terms. Assuming a set of type variables α and primitive type constants $T\,\tau_1 \ldots \tau_n$, the syntax of types τ and of type schemes σ is given by

$$\tau \quad ::= \quad \alpha \mid T\,\tau_1 \ldots \tau_n \mid \tau \to \tau \tag{1.2}$$

$$\sigma \quad ::= \quad \tau \mid \forall \alpha.\sigma \tag{1.3}$$

The typing rules for this type system are given in figure 1.1, and they give a system that can be used to find the type for a given expression in Exp.

1.2.3 Qualified types

In the Hindley-Milner system, we allow only universally quantified types. The statement $\forall\alpha$ is interpreted as 'for all types α in the language'. Qualified types, developed by Mark P. Jones [Jon92] with inspiration from work on type classes, for instance by [WB89, Kae92], allow an extension of this system so that you can qualify a type α with a predicate π. This allows types such as $\forall\alpha.\pi(\alpha)$, meaning 'for all types α in the language, such that the predicate π holds for α ($\pi(\alpha)$ holds)'.

This is done by an extension of the typed lambda calculus to allow for types with predicates as well as an extension to the type inference algorithm and extensions to the concrete language to allow you to express the qualified types. There is also a need to introduce something called 'evidence' so that every use of a qualified type is accompanied by some *evidence* that the use is proper.

Qualified types allow for many kinds of predicates. Two interesting types of predicates are type classes and subtyping which we will consider in greater detail.

```
remove            :: Eq a => [a] → a → [a]
remove []         _ = []
remove (x : xs) el = if (x == el) then
                       remove xs el
                     else
                       x : remove xs el
```

Figure 1.2: A Haskell function removing elements from a list

Type classes

The introduction of type classes to deal with *ad-hoc* polymorphism is due to Wadler and Blott [WB89] and originates in the work of the Haskell [Pey03] committee. Formally, the predicates for type classes take the form C τ, where C is the type class that τ belongs to.

Type classes are part of the Haskell language and allows for overloading functions and operators for different types, a typical example being the `Eq` class which allows equality checks, in effect overloading the `==` operator.

Type classes in Haskell are on the form `Eq a`, asserting that the type `a` is an instance of the type class `Eq`. This makes it possible to write function declarations such as the function `remove`, figure 1.2, which removes all occurrences of an element from a list, and which will work for all types for which equality is defined.

Subtyping

Subtyping corresponds to the usual subtyping relation in object orientation. The predicate[1] means that $\tau \leq \tau'$, τ is a subtype of τ' and thus that the type τ is the more general one. That τ is more general means that whenever we need something of type τ' we can instead use something of type τ.

This also accommodates for concepts such as `Int` $\leq$ `Float`. This relationship is motivated by the fact that whenever we want something of type `Float` we can instead use something of type `Int`. This assumes that there is a way to (automatically) perform the conversion from `Int` to `Float`. This conversion function is the *evidence* that the predicate holds.

The evidence for subtyping is referred to as the *coercion function*.

Entailment

The qualified type system relies on the entailment relationship $\Vdash$ to determine 'entailment' between sets of predicates, the statement $A \Vdash B$ means that if the set of predicates A holds, then the set of predicates B will also hold. This relationship will vary depending on the specific type system.

[1] Jones uses the notation $\tau \subseteq \tau'$, we adopt the notation $\tau \leq \tau'$ used in Timber throughout.

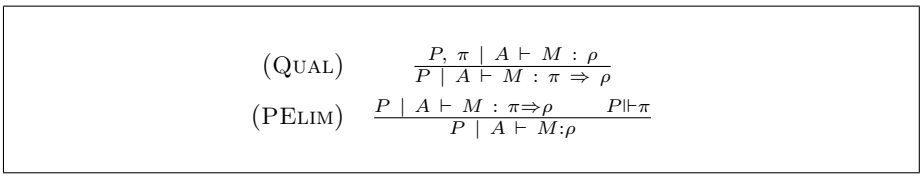

Figure 1.3: Extended typing rules for qualified types

Extensions of typed lambda calculus

The syntax for type schemes in equation 1.3 must be extended so that we can express qualified types:

$$\sigma ::= \tau \mid \forall\alpha.\sigma \mid \pi \Rightarrow \sigma, \tag{1.4}$$

where the $\Rightarrow$ notation is used to indicate which set predicates π the type scheme σ must fulfill. This allows us to express qualified types without further modifying the syntax of our lambda calculus.

We also need to extend the typing rules in figure 1.1 with the new rules, the rules needed for typing a system with qualified types are presented in figure 1.3. These rules allow us to deduce qualified types for our expressions, they are presented as closely as possible to the form they take in the Timber system.

Evidence

Evidence or *witnesses* are evidence that a certain predicate holds. It can be an instance of the overloaded function or a coercion function in the case of subtyping. Evidence is not strictly needed to infer the type of a statement, but is instead needed to evaluate the statement providing the information needed to deal with the overloading. The appropriate evidence functions will be used whenever overloading occurs, so that the comparison `1 == 3` in Haskell will use the function `(==)` that is supplied for integers.

To assert that the type τ belongs to the type class π we need as *witnesses* all the overloaded functions/operators that belong to the type class π. In Haskell the witnesses for `Eq a` is a function that is able to check equality for values of type `a` as well as a function that is able to check inequality for the type `a`, that is two functions: `(==) :: a->a->Bool` and `(/=) :: a->a->Bool`.

The coercion function for the relationship $\tau \leq \tau'$ is a function of type $\tau \rightarrow \tau'$ that converts an instance of type τ to an instance of type τ'. This corresponds to an 'upcast' in C++ terminology, an example of this is in figure 1.4.

Type inference system

Jones [Jon92] shows that there is an ordering for type assignments and sets of predicates, and therefore a most general type assignment (the principal type) such that no other type assignment is more general than the most general one. In [Jon99] he describes a new, simpler, implementation of a type checker that can handle qualified types.

This results in a sound and complete extension to the algorithm W from Milner [Mil78] that is capable of calculating principal types in systems with type classes [WB89].

```
class Shape { ... };
class Square : public Shape { ... };

Shape * upcaster(Square *sq) {
  Shape *shape = (Shape *) sq;
  return shape;
}
```

Figure 1.4: An upcast in C++

1.2.4 First-class polymorphism

A value or a range of values represented by a datatype is a 'first-class' citizen if the language does not in any way discriminate against the value, letting it appear in any syntactic construct in which other values may occur.

The Hindley-Milner type system is limited though, polymorphic values are not first-class. Formally this is captured by making the distinction between monomorphic types and polymorphic type schemes.

Extensions to the Hindley-Milner system which allows first-class polymorphism exist in [Jon97, OL96]. The task of defining typing rules that allow type schemes as arguments to functions is not impossible, however, the user is then required to define which type scheme to be used in the lambda term. Odersky and Läufer extends the syntax of type schemes with $\sigma \rightarrow \sigma'$:

$$\sigma ::= \tau \mid \forall\alpha.\sigma \mid \sigma \rightarrow \sigma', \tag{1.5}$$

and the term language with annotated terms:

$$\begin{array}{rcll} e & ::= & \lambda x :: \sigma.M \textit{ annotated terms} \\ & \mid & \ldots \end{array} \tag{1.6}$$

This makes it possible to require things from callers. Formally this is done by moving the quantifier from the outermost level to an inner level, `f (g:` $\forall c$`[c]` $\rightarrow$ `Int) = g [``hello''] + g [1, 2]`.

The types and typing rules for FCP is a conservative extension to the Hindley-Milner system shown in figure 1.5. All programs that were previously typable in the Hindley-Milner system continue to be so.

1.2.5 Timber type system

The Timber type system is an extension of the Hindley-Milner system that incorporates both the Qualified Types discussed in section 1.2.3 and First Class Polymorphism as discussed in section 1.2.4. We do not intend to explain the exact mechanisms of the type system, but a detailed explanation can be found in a forthcoming paper [Nor04].

Since Timber has both of these extensions we need the type scheme syntax to be the union of the syntaxes in equations 1.4 and 1.5:

Predicate rules:

$$\frac{P \vdash \forall\alpha.(\sigma \leq \sigma') \quad \alpha \notin fv(\sigma)}{P \vdash \sigma \leq \forall\alpha.\sigma'}\ \textsc{AllSup} \qquad \frac{P \vdash [\tau/\alpha]\sigma \leq \sigma'}{P \vdash \forall\alpha.\sigma \leq \sigma'}\ \textsc{AllSub}$$

$$\frac{}{P \vdash id : \tau \leq \tau}\ \textsc{Refl} \qquad \frac{P \vdash \sigma_1' \leq \sigma_1 \quad P \vdash \sigma_2 \leq \sigma_2'}{P \vdash \sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2'}\ \textsc{Fun}$$

Typing rules:

$$\frac{P \mid A \vdash M : \sigma' \to \sigma \quad P \mid A \vdash N : \sigma'}{P \mid A \vdash M\,N : \sigma}\ \textsc{App}$$

$$\frac{P \mid A, x{:}\sigma \vdash M : \sigma'}{P \mid A \vdash \lambda x{::}\sigma.M : \sigma \to \sigma'}\ \textsc{Lam}$$

$$\frac{P \mid A \vdash M : \sigma \quad P \vdash \sigma \leq \sigma'}{P \mid A \vdash M : \sigma'}\ \textsc{Sub}$$

Figure 1.5: Extensions to Hindley-Milner for first class polymorphism.

$$\sigma ::= \tau \mid \forall\alpha.\sigma \mid \pi \Rightarrow \sigma \mid \sigma \to \sigma'. \tag{1.7}$$

The syntaxes for the predicates and predicate schemes are:

$$c ::= T\,\tau_1 \ldots \tau_n \mid \tau_1 \leq \tau_2, \tag{1.8}$$

$$\pi ::= c \mid \pi_1 \Rightarrow \pi_2 \mid \forall\alpha.\pi \mid \sigma_1 \leq \sigma_2. \tag{1.9}$$

Note that the Timber system allows subtyping predicates for typeschemes.

The type rules for the Timber type system will be the union of the rules presented in the previous sections (figures 1.3 and 1.5), this expresses the fact that everything that can be expressed in either of those two extensions can be expressed in the Timber type system. The full typing rules and the predicate rules are summarized in Appendix A. The predicate rules for Timber show how predicates relate to each other and contain all of the rules from First Class Polymorphism together with Timber specific ones.

This type system is powerful enough to type check the Timber language. The work in this thesis is based on the Timber type system and the Timber language.

It is not necessary to understand the workings of the type system to understand the contents of this thesis. It is enough to keep in mind that Timber has a type system that is an aggregate of the extensions discussed above. The relevant details of the type inference algorithm will be discussed in connection with the concrete implementation.

1.3 The Timber Compiler

The Timber type system and the Timber language are implemented in the Timber compiler. The compiler is written in Haskell. All of our modifications to the compiler has been made in a pre-release version of the compiler.

The compiler works in multiple stages. The relevant stages in the context of this thesis are:

`parser` parses the input and and outputs a syntax tree.

`desugar` creates a less complex form of the syntax tree. For instance, all object syntax is removed in this step together with nested pattern matchings.

`syntax2core` transforms the syntax tree from the `Syntax` to the `Core` form, reducing it to a few basic constructs. It also supplies type annotations for all identifiers and replaces wild-cards(_) with new type variables. Wild-cards represent type variables not specified by the programmer.

`kindcheck` does kind checking, making sure that all elements have compatible kinds. The interested reader is referred to [Jon93] for an in-depth treatment of kinds.

`typecheck` is the actual type inference/type checking. It is divided into the type checker and the context reducer.

The Timber language has its origins in Haskell, so all examples of Timber programs should be accessible to someone who knows a bit of Haskell. We will not be using any Timber specific constructs, except for how we display types. The Timber type

```
a1 -> a1 -> Bool \\ Ord a1, a1 :: *
```

has the same meaning as the Haskell type

```
Ord a => a -> a -> Bool.
```

1.4 Previous work on type error reporting

Most of the existing work has been on the languages ML and Haskell and has been restricted to the Hindley-Milner system, sometimes extended with type classes.

One method used for reporting type errors has been to identify the usage of a term and suggest that the minority use of the term is the erroneous one. In figure 1.6 (adapted from [HJSAA02]) we show an instance of this type of error. In this case the error is caused because the first case has type `Bool` and the other cases have the type `String` and all branches of a case expression has to have the same type. Many current compilers will report the type of the second case (`String`) as being in error, but the reasoning has been that the majority is probably right.

```
f x =case x of
        0 →   False
        1 →   "One"
        2 →   "Two"
        3 →   "Three"
        4 →   "Four"
```

Figure 1.6: An example of a broken case expression

Beaven and Stansifer [BS93] suggest a model for allowing the user to explore the 'how' and 'why' of type assignment, and why a type error has occurred in a program. This gives the user an interface to explore the 'explanation space' of a type assignment.

The Helium compiler [HLI03] has been developed especially for learning Haskell and has a strong emphasis on giving good error messages for type errors. This compiler is however limited in scope to traditional Haskell and the current implementation implements a subset of Haskell which is most notably missing type classes.

Work has been done on finding the minimal subset of constraints that cause an error [BSW03, CH95] to limit the size of the error report presented to the user. Haack and Wells [HW04] discuss slicing the code into minimal subsets of code, and provide a prototype system for ML that implements their slicing rules.

Helium has lists of common errors and a hint to each one of them of how to solve the problems. The common errors are collected with a logging compiler in a beginners course in functional programming at Utrecht University.

Chapter 2

Analysis

2.1 Problem overview

In the classical Hindley-Milner system, the failure of type inference, a type error, occurs only at unification and results in the conclusion that two types are not unifiable. This means that all type errors can be expressed as a failure of an equality to hold. This also holds for many extensions to the Hindley-Milner system and is a basic premise for the approaches to error reporting discussed in section 1.4. An example of a type error in Hugs [JR+02] can be found in figure 2.1, where two types are not unifiable, and thus cannot be equal.

In the Timber type system the equalities are replaced by inequalities, and type errors are found in the context reduction phase as an inconsistency among type constraints. One example of such unsatisfiable type constraints are the constraints: `Int < a, a < [Int]`. The presence of subtyping is what sets our task fundamentally apart from that of previous work on error reporting.

The basic problem is to display this conflict to the user in a succinct way. It has been observed that the size of the set of constraints grow proportionally to the number of statements in the code, potentially leading to unwieldy error reports.

`error2.hs`:

$f\ [] = True$
$f\ a = a$

```
Type checking
ERROR "error2.hs":2 - Type error in function binding
*** Term           : f
*** Type           : Bool -> Bool
*** Does not match : [a] -> Bool
```

Figure 2.1: An erroneous Haskell program with error message

We divide the problem into two parts: First we decide what information is needed for error presentation, and modify the compiler to annotate the internal syntax tree with that information and preserve it through the internal transformations in the compiler. The second part is providing an error reporting routine that will use the annotations to present a type error to the user in an understandable way.

2.2 Theory

2.2.1 Annotation of the abstract syntax tree

Each leaf in the abstract syntax tree (AST) is annotated with its origin in the source code. Each annotation contains information about which portion of the source code it annotates and it also contains a reason for the annotation.

This annotation is assumed to exist on all leaves in the AST when the type inference algorithm is applied. All user supplied type signatures are also annotated.

2.2.2 Annotation of predicates

All predicates are annotated with an explanation of their origin. Possible origins of predicates are type checking of applications, declarations, the initial environment or type checking of user supplied signatures.

2.2.3 Experimental annotation of types

As an attempt to solve certain problems with the error reporting, namely the fact that the substitutions change the types in the predicates, we introduce an experimental annotation of types in the type tree. The source of the problem is that two occurrences of the same variable are bound by reusing the same type variable, not by introducing two type variables and a constraint relating these variables. This, in combination with the substitutions which are applied to the predicates during context reduction, leads to predicates where the annotation of the predicate points to the correct location in the program but the contents of the predicate do not directly relate to that location predicate, but instead is the result of substitutions applied to the predicate.

So while every part of the expression by itself might be solvable the whole set of predicates is unsolvable. In an attempt to provide error reporting in this case, we annotate every type and type scheme with its origin which can be either a specific location in the code (for explicit type annotations), an expression (for type variables connected to expressions in the code) or a substitution that made two previous types equal. For a good example of when this is needed see `Test5` in 3.2.

It should be noted that this annotation is experimental, and that we are not certain that this is the best way to get this information. Other alternatives could be explored. Also, our current implementation does not annotate the $\textsc{Mu}^W$ or $\textsc{Gen}^W$ rules.

$$\frac{A(x) = \forall \overline{\alpha}.\overline{\pi} \Rightarrow \rho \quad \overline{\alpha}, \overline{v} \text{ new}}{\overline{v:\pi} \mid A \vdash^W x : \rho_{B_0} \rightsquigarrow x\,\overline{v}} \text{ VAR}^W$$

$$\frac{P \mid A \vdash^W M : (\sigma \to \rho)_{B_0} \rightsquigarrow M' \quad P' \mid A \vdash^G N : \sigma'_{B_1} \rightsquigarrow N' \quad B_2 = MN, v \text{ new}}{P, P', v:\sigma'_{B_1} \leq \sigma_{B_0} \mid A \vdash^W M\,N : \rho_{B_2} \rightsquigarrow M'\,(v\,N')} \text{ APP1}^W$$

$$\frac{P \mid A \vdash^W M : \alpha_{B_0} \rightsquigarrow M' \quad P' \mid A \vdash^W N : \rho_{B_1} \rightsquigarrow N' \quad B_2 = MN, v, \beta \text{ new}}{P, P', v:\alpha_{B_0} \leq (\rho \to \beta)_{B_1} \mid A \vdash^W M\,N : \beta_{B_2} \rightsquigarrow v\,M'\,N'} \text{ APP2}^W$$

$$\frac{P \mid A \vdash^G M : \sigma \rightsquigarrow M' \quad P' \mid A, x:\sigma \vdash^W N : \rho_{B_0} \rightsquigarrow N'}{P, P' \mid A \vdash^W \mathbf{let}\, x = M\, \mathbf{in}\, N : \rho_{B_0} \rightsquigarrow \mathbf{let}\, x = M'\, \mathbf{in}\, N'} \text{ LET}^W$$

$$\frac{P \mid A, x:\sigma_{B_0} \vdash^W M : \rho_{B_1} \rightsquigarrow M' \quad \sigma = [\overline{\beta}/\_]\hat{\sigma} \quad B_1 = x, \overline{\beta} \text{ new}}{P \mid A \vdash^W \lambda x :: \hat{\sigma}.M : \sigma_{B_0} \to \rho_{B_1} \rightsquigarrow \lambda x :: \sigma.M'} \text{ LAM}^W$$

$$\frac{P \mid A, x:\sigma \vdash^G M : \sigma' \rightsquigarrow M' \quad \sigma = [\overline{\beta}/\_]\hat{\sigma} = \forall \overline{\alpha}.\overline{\pi} \Rightarrow \rho \quad \overline{\alpha}, \overline{\beta}, \overline{v}, v \text{ new}}{P, \overline{v:\pi}, v:\sigma' \leq \sigma \mid A \vdash^W \mu x :: \hat{\sigma}.M : \rho \rightsquigarrow (\mu x :: \sigma.v\,M')\,\overline{v}} \text{ MU}^W$$

$$\frac{\overline{v:\pi}, \overline{\varepsilon} \mid A \vdash^W M : \rho \rightsquigarrow M' \quad \theta \mid \overline{v}':\overline{\pi}' \vDash^R \overline{e} : \overline{\pi}, \overline{\varepsilon}}{\theta \mid A \vdash^G M : gen(\theta A, \theta(\overline{\pi}' \Rightarrow \rho)) \rightsquigarrow \lambda \overline{v}'.(\lambda \overline{v}.M')\,\overline{e}} \text{ GEN}^W$$

Figure 2.2: Modified type inference rules

2.2.4 Type inference algorithm

The Timber type inference algorithm consists of two parts: the type inference algorithm itself and a context reducer. The type inference algorithm outputs a set of predicates that the context reducer reduces.

The type inference algorithm itself will never fail, instead it will produce predicates that cannot be reduced, leaving the type error detection to the context reduction algorithm.

The type inference algorithm creates new subtyping predicates on the form $\sigma' \leq \sigma$ in the rules APP1^W,APP2^W and MU^W. These predicates are collected and annotated with the expression they belong to.

The type checker also outputs new type expressions with the suggested experimental annotations. These new types will be annotated with a subscripted B. The annotated type inference rules are presented in figure 2.2. All of these annotations are of the type relating to an expression in the code. The squiggly arrow in the type inference rules represent transformations were witness variables are introduced.

Full details of the type inference algorithm can be found in [Nor04].

2.2.5 Context reduction algorithm

The context reduction algorithm proceeds by successive "small-step reductions" that reduce the set of predicates, reducing complex predicates to simpler ones and removing predicates that can be satisfied. Since two predicates are never

reduced to an aggregate predicate the annotation can be preserved through the reduction.

We propose that annotations of predicates be denoted by a subscripted letter on the implication arrow($\Rightarrow_A$) meaning that the predicates to the right of the arrow has the annotation A.

All rules in the small-step reduction are modified to incorporate the transfer of annotations. All predicates in the initial set of predicates have been annotated by the type inference algorithm. The modified rules are shown in figure 2.3. The original rules are in [Nor04].

With the annotation of types we also modify the substitution function so that when we substitute a type scheme σ_B for a type variable $\alpha_{B'}$ the resulting type gets the new annotation $\sigma_{B,B'}$, indicating both the reason for the original type variable and the reason for the new type expression.

2.2.6 Error reporting

All type errors manifest themselves as irreducible predicates in the context reduction. Since all the initial predicates produced by the type inference algorithm are annotated and the small-step reductions preserve the annotations we know that all predicates will contain an annotation of their origin. This gives a theoretical basis for including the location information in an error report.

Note that we focus on what information the user will need to fix the type error, the actual presentation of the error message to the user is a HCI problem, and should be studied separately.

2.3 An example of deduction

2.3.1 Premises

We want to type check the simple addition: ((+)1)1.0. The elements of the expression has known types and locations: the constant 1 has the type Int and the location l_1, 1.0 has the type Float and location l_2 and the variable (+) has the type $Num\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ and location l_3.

We start with the following set of predicates that are known to hold: $\{Int \leq Int, Float \leq Float, Num\ Int, Num\ Float\}$.

2.3.2 Type inference

Following the type inference rules we first use the rule VAR^W to get fresh type variables for our instance of (+). We choose the fresh type variable as v_1, giving it the type $v_1 \rightarrow v_1 \rightarrow v_1$. We also add the predicate $Num\ v_1$, annotated with $A_1 = l_1$, to the set of predicates.

We then use the first application rule APP1^W on the inner parenthesis, using (+) for M and 1 for N. This adds the new subtyping predicate $Int \leq v_1$ with the location annotation A_2 = "an application from l_1 to l_2". The type of the expression will then be $v_1 \rightarrow v_1$.

Finally we again use the application rule APP1^W using the result from the first application as M and 1.0 as N. This introduces the subtyping predicate $Float \leq v_1$ annotated with A_3 = "an application from (application from l_1 to l_2) to l_3", and gives the final type of the expression as v_1.

$$
\begin{array}{lll}
1 & e : \forall\theta.P \Rightarrow_A (\forall\alpha.\pi) & \overset{\mathbf{1}}{\rightleftharpoons} \quad e : \forall[T\,\overline{\beta}/\alpha]\theta.P \Rightarrow_A \pi \\
 & & \quad \overline{\beta} = fv(\theta P, \theta(\forall\alpha.\pi)) \\
 & & \quad \alpha, T \text{ new} \\
2 & \lambda v.e : \forall\theta.P \Rightarrow_A (\pi' \Rightarrow \pi) & \overset{\mathbf{1}}{\rightleftharpoons} \quad e : \forall\theta.(P, v{:}\pi') \Rightarrow_A \pi \\
 & & \quad v \text{ new} \\
3 & e : \forall\theta.P \Rightarrow_A (\sigma \leq \forall\alpha.\sigma') & \overset{\mathbf{1}}{\rightleftharpoons} \quad e : \forall\theta.P \Rightarrow_A \forall\alpha.(\sigma \leq \sigma') \\
 & & \quad \alpha \text{ new} \\
4 & \lambda v.\lambda v'.e\,v'\,v : \forall\theta.P \Rightarrow_A (\sigma \leq \pi \Rightarrow \sigma') & \overset{\mathbf{1}}{\rightleftharpoons} \quad e : \forall\theta.P \Rightarrow_A \pi \Rightarrow (\sigma \leq \sigma') \\
 & & \quad v, v' \text{ new} \\
5 & e : \forall\theta.P \Rightarrow_A (\forall\alpha.\sigma \leq \rho) & \overset{\mathbf{1}}{\rightleftharpoons} \quad e : \forall\theta.P \Rightarrow_A (\sigma \leq \rho) \\
 & & \quad \alpha \text{ new} \\
6 & \lambda v.\, e'\,(v\,e) : \forall\theta.P \Rightarrow_A (\pi \Rightarrow \sigma \leq \rho) & \overset{\mathbf{1}}{\rightleftharpoons} \quad e : \forall\theta.P \Rightarrow_A \pi, \\
 & & \quad e' : \forall\theta.P \Rightarrow_A (\sigma \leq \rho) \\
 & & \quad v \text{ new} \\
7 & e'' : \forall\theta.P \Rightarrow_A (\sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2') & \overset{\mathbf{1}}{\rightleftharpoons} \quad e : \forall\theta.P \Rightarrow_A (\sigma_1' \leq \sigma_1), \\
 & & \quad e' : \forall\theta.P \Rightarrow_A (\sigma_2 \leq \sigma_2') \\
 & & \quad e'' = \lambda v.\lambda v'.e'\,(v\,(e\,v')) \\
 & & \quad v, v' \text{ new} \\
8 & e : \forall\theta.P \Rightarrow_A (\alpha \leq \sigma_1 \to \sigma_2) & \overset{\theta_1}{\rightleftharpoons} \quad e : \theta'(\forall\theta.P \Rightarrow_A (\alpha \leq \sigma_1 \to \sigma_2)) \\
 & & \quad \beta_1, \beta_2 \text{ new} \\
 & & \quad (\beta_1 \to \beta_2) \overset{\theta_1}{\sim} \theta\alpha \\
9 & e : \forall\theta.P \Rightarrow_A (\sigma_1 \to \sigma_2 \leq \alpha) & \overset{\theta_1}{\rightleftharpoons} \quad e : \theta'(\forall\theta.P \Rightarrow_A (\sigma_1 \to \sigma_2 \leq \alpha)) \\
 & & \quad \beta_1, \beta_2 \text{ new} \\
 & & \quad (\beta_1 \to \beta_2) \overset{\theta_1}{\sim} \theta\alpha \\
10 & v_i\,\overline{e}_i : \forall\theta.P \Rightarrow_A c & \overset{\theta_i \setminus \overline{\alpha}_i}{\rightleftharpoons} \quad \overline{e}_i : \theta_i(\forall\theta.P \Rightarrow_A \overline{\pi}_i) \\
 & & \quad (v_i : \forall\overline{\alpha}_i.\overline{\pi}_i \Rightarrow c_i) \in P \cup P_0 \\
 & & \quad \overline{\alpha}_i \text{ new} \\
 & & \quad \theta c \overset{\theta_i}{\sim} \theta c_i \\
11 & \lambda v.v : \forall\theta.P \Rightarrow_A \tau \leq \tau' & \overset{\theta_1}{\rightleftharpoons} \quad \emptyset \\
 & & \quad \theta\tau \overset{\theta_1}{\sim} \theta\tau' \\
12 & \tau = \tau' & \overset{\theta}{\rightleftharpoons} \quad \emptyset \\
 & & \quad \tau \overset{\theta}{\sim} \tau'
\end{array}
$$

Figure 2.3: The annotated small step reduction rules

After this the type inference finishes successfully, leaving the final set of predicates $\{Num\ v_1,\ Int \leq v_1,\ Float \leq v_1\}$ for the context reduction, with the annotations $\{A_1,\ A_2,\ A_3\}$ as above.

2.3.3 Context reduction

The context reduction algorithm will start with one of the subtyping predicates, for instance $Float \leq v_1$. Since the only type in our system that is greater than

$Float$ is $Float$ itself, v_1 will have to be $Float$ and we will substitute $Float$ for v_1 using rule 11 from the small-step reduction rules (figure 2.3). This produces the predicate set $\{Num\ Float, Int \leq Float, Float \leq Float\}$, with the same set of annotations as before,

After removing the satisfied predicates $Num\ Float$ and $Float \leq Float$ the single predicate $Int \leq Float$ remains. Since Int is not a subtype of $Float$ in our system, this predicate is not satisfiable. The annotation for the failing predicate will be A_2.

Chapter 3

Implementation and examples

3.1 Implementation

3.1.1 Annotation of the syntax tree

The parser is modified to annotate each leaf on the syntax tree with a `LocInfo` structure indicating the symbol's position and function in the source code. The definition of the annotations can be found in figure 3.2. The `LocWhy` structure looks as follows:

`LocUnknown` Anything that the reason is unknown for. In a complete implementation this should not be needed since everything should be known.

`LocLiteral` A literal in the source code. Examples are 2 and "hello world".

`LocEnvironment` Things from the initial environment. Our implementation had the type of several functions in the initial environment, head and length among them.

`LocVar` Variables in the source code.

`LocCon` Type constructors in the source code.

`LocAp` For applications. This is used by the error reporting routines rather than the lexer and parser. We believe that this could be eliminated.

The leaves in the syntax tree represent the elements in the source code. The simple example program `TestA`, figure 3.1, results in the AST

```
01 Module "TestA" [
02  DEqn (LFun "k" [])
03       (RExp
04        (EAp
05         (EAp
06          (EVar (Id' "+"
07           (LocInfo (3,8) (3,8) LocLiteral)))
```

```
08          (ELit (Lit' (Int 1)
09           (LocInfo (3,4) (3,4) LocLiteral))))
10         (ELit (Lit' (Rat (1 % 1))
11          (LocInfo (3,8) (3,8) LocLiteral)))))
12 ]
```

where each leaf is annotated with the associated source code location. After the parser builds the AST the module Desugar transforms it into a simpler form, removing syntactic sugar. The tree is then transformed from Syntax to Core form during the Syntax2Core transformation. In this step our effort is preserving the location information.

---

module *TestA* **where**
$k = 1 + 1.0$

Figure 3.1: `TestA` - A very simple Timber program

The internal nodes of the AST has as previously stated no location information stored in them. In order to find the location of those nodes the subtrees can be traversed.

An example is the function application on line 5 in example 1. The left subtree is just one leaf, a variable with location information. The right subtree is also a leaf, a literal with location information. By combining these `LocPos`' a region starting at line 3 column 4 and ending at column 8 can be formed. The result is rather good for binary operators since the leafs starting points and endpoints cover the operator. For ordinary functions the region will be from where the function name ends to the end of the arguments. This is a flaw since the region is approximated by the position of the arguments. Other types of internal nodes suffer from the same problem, let expressions being the worst.

A possible way around having to carry around full annotation would be to run the parser again if an error has occurred and parse to a different kind of syntax tree with an extended annotation.

3.1.2 Annotation of the predicates

When the type checker generates predicates they are annotated with their origin in the code using the `PredWhy` data type (figure 3.3). For our example `TestA` (figure 3.1) the type inferrer produces these predicates

```
Float < _2,
Int < _2,
Num _2
```

where _2 represents a type variable that stands for the type of `k`.

```
data LocWhy = LocUnknown
            | LocLiteral
            | LocEnvironment
            | LocVar
            | LocCon
            | LocAp
              deriving (Eq, Show)

type LocPos  = (Int, Int)    — (row,col)

data LocInfo = LocInfo LocPos LocPos LocWhy
              deriving (Eq, Show)
```

Figure 3.2: The Haskell definition of the location information

Example 1. *The annotation for the first of these predicates:*

```
PredAp (EAp
         (EAp
            (EVar (Id' "+"
               (LocInfo (3,8) (3,8) LocLiteral)))
            (ELit (Lit' (Int 1)
               (LocInfo (3,4) (3,4) LocLiteral))))
         (ELit (Lit' (Rat (1 % 1))
            (LocInfo (3,8) (3,8) LocLiteral)))))
```

This means that the predicate exists because of an application of the application of the integer `1` to the variable `+` on the rational number `1.0`, `+` is defined in the environment as a function of type:

```
(+) :: a -> a -> a \\ Num a, a ::*.
```

The generated set of predicates is then given to the context reducing algorithm to reduce. It then works on the predicates one by one, simplifying and breaking them up, with the ultimate goal of removing them altogether. In the process of removing predicates, it also creates substitutions that act on the remaining predicates.

If the reducer finds a predicate that cannot be satisfied, an error condition is triggered and our error reporting routine is called.

Our example produces the unsatisfiable predicate `Int < Float`. Note that the predicate *could* be satisfiable, if we adopted the notion that `Int` is a subtype of `Float`, but it is not in this implementation.

The `PredWhy` structure looks as follows:

`PredAp` This is used if the predicate is derived from an application. A reference to the expression is kept to be used in the error reporting routines.

```
data PredWhy = PredAp Exp
             | PredDecl TScheme TScheme
             | PredEnv
             | PredSig
               deriving(Eq, Show)
```

Figure 3.3: The Haskell definition of the predicate information

```
data TypeWhy = TypeLoc LocInfo                 — Type annotation
             | TypeExp Exp                     — Type deduced from expression
             | TypeSubst TypeWhy TypeWhy       — 'Unification' of two types
               deriving(Eq, Show)
```

Figure 3.4: The Haskell definition of the type information

`PredDecl` If the user has supplied a type for a function, this is the annotation for it. The two type schemes are referenced for the error reports.

`PredEnv` Anything coming from the environment during derivation gets this annotation. No extra parameters.

`PredSig` The predicates given by the user in the function signature has this annotation.

3.1.3 Experimental annotation of types

Since the reduction of predicates can produce substitutions that are applied to other predicates, a predicate can look substantially different at the time of an error compared to the time it was created. This problem becomes apparent when the error is really a conflict between two different uses of the same variable. This can result in predicates that are formally correct but where the user will have a hard time understand where the types come from.

As an experimental feature of our implementation we have added annotations of types (figure 3.4). The explanation `TypeSubst` explains that the type is there because of a substitution, and it contains the reason for both the original type variable and the substituted type.

This annotation will allow us to provide a more detailed error report to the user. It will allow us both to explain why the predicates exist, for instance because of an application, and why the types in the predicates have a certain value.

3.1.4 Error reports

When an error occurs in the context reducer it calls `reportError` from our `ErrorReport` module. This function is responsible for displaying and explaining the error.

Our example `TestA` (figure 3.1) results in this error message:

```
1: ERROR: Int <1> is not a subtype of Float<2>.
2: Subtype constraint inferred from application of:
3:  Int
4: to:
5:  a1 -> a1 -> a1 \\ Num a1, a1 :: *
6: application at line 2, col 4 - 8.
7: <1> because of literal at line 2, col 4
8: <2> because of literal at line 2, col 8
```

Line 1 displays the predicate that could not be reduced. This corresponds to the unsatisfiable predicate `Int < Float`.

Line 2 tries to explain why the predicate exists. It exists because of an application, with lines 3-5 saying that it is an application of something of type `Int` to a function of the type `a1->a1->a1 \\Num a1`. Line 6 shows the location of the application.

Lines 7 and 8 serve as footnotes for the `Int` and `Float` respectively, explaining from where these concrete types come, pointing to actual locations in the code

In addition to using the annotations on the predicates for presenting the error report, our experimental implementation uses the additional `TypeWhy` information to provide a more detailed error message in some situations, most notably in the presence of substitutions.

3.2 Examples

This section presents a long list of programs with type errors and the associated type error report.

```
module Test1 where

k = (>) 0.0 3
```

The function $(>)$ has type $a \rightarrow a \rightarrow Bool \setminus\setminus Ord\ a$ and a Float and an Int are passed as arguments. This is a type error since Int is not a subtype of Float. The error message reported by the compiler as seen below matches this information.

```
ERROR: Int <1> is not a subtype of Float<2>.
Subtype constraint inferred from application of:
 a1 -> a1 -> Bool \\ Ord a1, a1 :: *
to:
 Int
application at line 4, col 5 - 8.
<1> because of literal at line 4, col 8
<2> because of literal at line 4, col 10
```

```
module Test2 where

fib n = iff (3 > n) 1 (fib (n - 1) + fib (n - 2))
k = fib 's'
```

Here a character is passed as argument to the function fib which is of type $Int \rightarrow Int$ since all numbers in the function are of type Int. The compiler reports this as Char is not a subtype of Int as below.

```
ERROR: Char <1> is not a subtype of Int<2>.
Subtype constraint inferred from application of:
 Int -> Int
to:
 Char
application at line 5, col 4 - 8.
<1> because of literal at line 5, col 8
<2> because of literal at line 4, col 32
```

```
module Test3 where

blanks n = iff (1 > n) ("") (" " ++ blanks (n-1))
k = blanks 3.0
```

This is the same category of type error as Test2, an application of a function to a different type. This error is mainly here to demonstrate that it works for lists as well, as shown below.

```
ERROR: Float <1> is not a subtype of Int<2>.
Subtype constraint inferred from application of:
 Int -> [Char]
to:
 Float
application at line 5, col 4 - 11.
<1> because of literal at line 5, col 11
<2> because of literal at line 4, col 46
```

```
module Test4 where

k :: Int->Int
k l = 2.0*l
```

The declaration of k is faulty, the real type of the function is $Float \rightarrow Float$ and not $Int \rightarrow Int$ as declared.

```
ERROR: Int <1> is not a subtype of Float<2>.
Subtype constraint inferred from declaration.
Declared type:
 Int -> Int.
inferred type:
 Float -> Float
<1> because of literal in type constructor at line 4, col 5
<2> because of literal at line 5, col 6
```

```
module Test5 where

f x = (x 7,x+7)
```

This example uses x twice in different ways. The error occurs in the type checker of the application of 7 to x after the type of x has been decided to be an Int because of the application to "+" together with 7, which forces x to be the same type as the constant 7.

The error is that $Int \rightarrow a$ is not a subtype of Int, for any value of a. Our explanation tries to convey the true source of the error: two incompatible uses of x.

In this test our experimental annotation of types (sections 2.2.3 and 3.1.3) is used. The information that a type substitution has taken place is used to emit the text `has to be the same as`, indicating that the true source of the type is somewhere else.

```
ERROR: Int->something(35) <1> is not a subtype of Int<2>.
Subtype constraint inferred from application at line 4, col 11 - 13.
<1> because of it has to be the same as
 Int -> _28 (application at line 4, col 7 - 9)
<2> because of literal at line 4, col 13
```

```
module Test6 where

g = 3.0

f x = (3 > x, x > g)
```

Here x is compared to both an Int and a Float. This is caught and the following error is reported.

```
ERROR: Int <1> is not a subtype of Float<2>.
Subtype constraint inferred from application at line 6, col 14 - 18.
<1> because of literal at line 6, col 7
<2> because of literal at line 4, col 4
```

```
module Test7 where

len' xs = (head xs) + (length xs)
o = len' "GH"
```

A list of characters is applied to the function len' which is of type $[a] \rightarrow a \setminus\setminus Num\ a,\ a :: *$. Since Char is not in the class Num it is a type error. The functions head and length are defined in the initial environment, therefore a message about the constraint being inferred from the environment.

```
ERROR: Type Char<1> not an instance of class Num.
Instance constraint inferred from Environment.
<1> because of literal at line 5, col 9
```

```
module Test8 where

f = head 3
```

A function operating on lists is applied to a single element here. This is shown below.

```
ERROR: Int <1> is not a subtype of [something(2)]<2>.
Subtype constraint inferred from application of:
 [a1] -> a1 \\ a1 :: *
to:
 Int
application at line 4, col 4 - 9.
<1> because of literal at line 4, col 9
<2> because of literal in environment at line 0, col 0
```

```
module Test9 where

f = let x = 3 in x > 0.3
```

In this case the type error is that the let bound variable x first has to be the same type as 3 and then, through the application of $>$, has to be the same type as 0.3. This leads to incompatible constraints for x, presented below.

```
ERROR: Int <1> is not a subtype of Float<2>.
Subtype constraint inferred from application at line 4, col 17 - 21.
<1> because of literal at line 4, col 12
<2> because of literal at line 4, col 21
```

Chapter 4

Conclusions

4.1 Conclusions

In section 2.2 we have proposed an annotation for the Timber type system which will preserve the information about the origin of a constraint throughout the type checking and context reduction. This makes it possible to produce good error reports.

In section 3.1 we describe a working implementation of the proposed system for the Timber compiler. We give a brief overview of the design and in section 3.2, we give examples of error reports for a selection of type errors.

We believe that the experimental evidence indicates that our approach produces a viable alternative for type error reporting in Timber and possibly other languages which have a type system with qualified types.

The main source of problems for us has been the presence of subtyping relationships which leads to a system where unification cannot be used and error detection cannot occur before the context reduction phase. Other efforts have been focused on less powerful type systems without predicates and do not encounter this problem.

A better understanding of the theory from the outset would have led to a better prototype and would have given us the possibility to plan our work in more detail. Unfortunately, time constraints limited the scope of our work; we feel that with better organization, we would have had time to look into some of the issues discussed in the next section, as well as produce a more complete implementation.

4.2 Future work

The current implementation should be rewritten or cleaned up with the current knowledge of the problem kept in mind. Before doing that, the experimental notation of types needs to be analyzed further. Without proper annotation of types it is impossible to report correct errors for some programs.

The error messages should be studied with human computer interaction in mind. With the current notation there is enough information to present an extensive error message to the user. The problem is to extract the relevant

information. A possibility to conceal or reveal different parts of the error tree in a graphical user interface could be helpful to narrow the error down.

The previous work done on finding minimal unsatisfiable subsets and the possibility to combine it with the current implementation in order to minimize the amount of superfluous information should be investigated.

A list of common type errors like in Helium could be collected for Timber as well and assembled with hints about how to correct them.

Appendix A

Rules for Timber

A.1 Typing rules

These are the complete rules for typing expressions in the Timber type system. They can be compared to the rules in figures 1.1 and 1.3. For a more complete overview of the type system, see [Nor04]. The squiggly arrows ($\rightsquigarrow$) represent transformations where witness variables are introduced.

$$\frac{A(x) = \sigma}{P \mid A \vdash x : \sigma \rightsquigarrow x} \textsc{Var}$$

$$\frac{P \mid A \vdash M : \sigma' \to \sigma \rightsquigarrow M' \quad P \mid A \vdash N : \sigma' \rightsquigarrow N'}{P \mid A \vdash M\,N : \sigma \rightsquigarrow M'\,N'} \textsc{App}$$

$$\frac{P \mid A \vdash M : \sigma \rightsquigarrow M' \quad P \mid A, x{:}\sigma \vdash N : \sigma' \rightsquigarrow N'}{P \mid A \vdash \mathbf{let}\, x = M\, \mathbf{in}\, N : \sigma' \rightsquigarrow \mathbf{let}\, x = M'\, \mathbf{in}\, N'} \textsc{Let}$$

$$\frac{P \mid A, x{:}\sigma \vdash M : \sigma' \rightsquigarrow M' \quad \sigma = [\overline{\tau}/\_]\hat{\sigma}}{P \mid A \vdash \lambda x{::}\hat{\sigma}.M : \sigma \to \sigma' \rightsquigarrow \lambda x{::}\sigma.M'} \textsc{Lam}$$

$$\frac{P \mid A, x{:}\sigma \vdash M : \sigma \rightsquigarrow M' \quad \sigma = [\overline{\tau}/\_]\hat{\sigma}}{P \mid A \vdash \mu x{::}\hat{\sigma}.M : \sigma \rightsquigarrow \mu x{::}\sigma.M'} \textsc{Mu}$$

$$\frac{P \mid A \vdash M : \sigma \rightsquigarrow M' \quad P \vdash e : \sigma \leq \sigma'}{P \mid A \vdash M : \sigma' \rightsquigarrow e\,M'} \textsc{Sub}$$

$$\frac{P \mid A \vdash M : \sigma \rightsquigarrow M' \quad \alpha \notin \mathit{fv}(P, A)}{P \mid A \vdash M : \forall\alpha.\sigma \rightsquigarrow M'} \textsc{Gen}$$

$$\frac{P, v{:}\pi \mid A \vdash M : \sigma \rightsquigarrow M'}{P \mid A \vdash M : \pi \Rightarrow \sigma \rightsquigarrow \lambda v.M'} \textsc{Qual}$$

A.2 Predicate rules

These are the complete rules for how predicates relate to each other. Note that these rules include witness terms as discussed in section 1.2.3.

$$\frac{P \vdash e : \pi \quad \alpha \notin fv(P)}{P \vdash e : \forall \alpha.\pi}\ \textsc{AllIntro} \qquad \frac{P \vdash e : \forall \alpha.\pi}{P \vdash e : [\tau/\alpha]\pi}\ \textsc{AllElim}$$

$$\frac{P, v{:}\pi' \vdash e : \pi}{P \vdash \lambda v.e : \pi' \Rightarrow \pi}\ \textsc{ImplyIntro} \qquad \frac{P \vdash e : \pi' \Rightarrow \pi \quad P \vdash e' : \pi'}{P \vdash e\, e' : \pi}\ \textsc{ImplyElim}$$

$$\frac{P \vdash e : \forall \alpha.(\sigma \leq \sigma') \quad \alpha \notin fv(\sigma)}{P \vdash e : \sigma \leq \forall \alpha.\sigma'}\ \textsc{AllSup} \qquad \frac{P \vdash e : [\tau/\alpha]\sigma \leq \sigma'}{P \vdash e : \forall \alpha.\sigma \leq \sigma'}\ \textsc{AllSub}$$

$$\frac{P \vdash e : \pi \Rightarrow (\sigma \leq \sigma')}{P \vdash \lambda v.\lambda v'.e\, v'\, v : \sigma \leq \pi \Rightarrow \sigma'}\ \textsc{ImpSup} \qquad \frac{P \vdash e : \pi \quad P \vdash e' : \sigma \leq \sigma'}{P \vdash \lambda v.e'\,(v\, e) : \pi \Rightarrow \sigma \leq q\sigma'}\ \textsc{ImpSub}$$

$$\frac{}{P, v{:}\pi \vdash v : \pi}\ \textsc{Hyp} \qquad \frac{P \vdash e : \sigma_1' \leq \sigma_1 \quad P \vdash e' : \sigma_2 \leq \sigma_2'}{P \vdash \lambda v.\lambda v'.e'\,(v\,(e\, v')) : \sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2'}\ \textsc{Fun}$$

$$\frac{}{P \vdash id : \tau \leq \tau}\ \textsc{Refl} \qquad \frac{P \vdash e : \tau \leq \tau' \quad P \vdash e' : \tau' \leq \tau''}{P \vdash \lambda v.e'\,(e\, v) : \tau \leq \tau''}\ \textsc{Trans}$$

Bibliography

[Bar88] H. P. Barendregt. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[BCJ+02] A.P. Black, M. Carlsson, M.P. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon Health & Science University, April 2002.

[BS93] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):17–30, 1993.

[BSW03] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 32–43. ACM Press, 2003.

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[CFC58] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory logic*, volume 1. North-Holland, 1958.

[CH95] V. Choppella and C. Haynes. Diagnosis of ill-typed programs. Technical Report TR426, Indiana University, 1995.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* , pages 207–212. ACM Press, 1982.

[Han87] Peter Hancock. *The Implementation of Functional Programming Languages*, chapter A Type-checker, pages 163–182. Prentice-Hall, 1987. Out of print.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, December 1969.

[HJSAA02] Bastiaan Heeren, Johan Jeuring, S. Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical Report UU-CS-2002-009, Institute of Information and Computing Science, University Utrecht, Netherlands, February 2002. Technical Report.

[HLI03] Bastiaan Heeren, Daan Leijen, and Arjan van Ijzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.

[HW04] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. submitted, 2004.

[Jon92] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., 1992.

[Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., 1993. ACM Press.

[Jon97] Mark P. Jones. First-class polymorphism with type inference. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 15–17 1997.

[Jon99] Mark P Jones. Typing Haskell in Haskell. Haskell Workshop, 1999.

[JR+02] Mark P Jones, Alastair Reid, et al. *The Hugs 98 User Manual*, 2002.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204. ACM Press, 1992.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System sciences (JCSS)*, 17:348–375, 1978.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, Cambridge, Mass, 1997.

[Nor04] Johan Nordlander. Qualified types with type signatures. *expected*, 2004.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersberg Beach, Florida*, pages 54–67, New York, N.Y., 1996.

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM Press, 1989.

[Wei03] Erik W. Weisstein. Mathworld, decision problem, 9 Oct 2003. `http://mathworld.wolfram.com/DecisionProblem.html` .