Eötvös Loránd University
Faculty of Informatics
Dept. of Programming Languages and Compilers

Compositional Type Checking

for Hindley-Milner Type Systems with *Ad-hoc* Polymorphism

Dr. Gergő Érdi

Supervisor: Péter Diviánszky

Budapest, 2011

Abstract

Statically typed functional programming languages usually employ a version of the Hindley-Milner type system extended with ad-hoc polymorphism. When the type checker detects an error, it has to report it to the programmer, to help in fixing the bug. However, usage of algorithms $\mathcal{W}$ and $\mathcal{M}$, commonly used to type-check languages with Hindley-Milner type systems, can result in cryptic error messages. We argue that the holistic nature of these algorithms is a cause of this.

Next, we describe a type checking algorithm originally presented by Olaf Chitil in 2001, that, by its compositional nature, claims to produce error messages that are more suitable for human processing — a property that type systems for imperative programming languages usually have. The main part of the thesis is extending the compositional algorithm for languages supporting ad-hoc polymorphism. A proof of concept implementation is presented for the Haskell 98 programming language, interfacing the Glasgow Haskell Compiler.

In conclusion, we present this implementation and ideas for future work.

For Her, the supportive, Him, the ever curious, and Her, my partner in crime.

Contents

Introduction

Software keeps growing larger and more complex, making it harder for us humans to understand programs enough to be able to reason about them. On the other hand, the field of software engineering is many decades old by now, and it's not unreasonable to expect of it the kind of maturity necessary to build large systems without it collapsing under its own complexity.

On one end of the validation spectrum, we have formal methods, which use mathematical logic to prove program correctness. For example, the *B method*[1] uses the Hoare-Dijkstra model[2, 3] of predicate transformations to ensure that the resulting program meets its specification. However, writing automatically verifiable proofs is an art in itself, requiring intellectual resources that are simply not available for most software development projects.

To make software development practical in the real world, a division of labour is applied to verification: some program properties are expressed in a machine-readable way and can thus be enforced by a compiler, while others are checked by humans using pen and paper proofs or good old hand-waving. *Static typing* provides an old and tried way of using so-called type checkers for ensuring program properties, and various type systems correspond to different levels of formalness. Milner's slogan "Well-typed programs don't go wrong"[4] is true in general insofar as "rightness" can be expressed in the given type system, as stated formally by the Curry-Howard isomorphism[5].

The other straightforward way to try attacking increasing software complexity is breaking programs up into manageable chunks. For this approach to work, the programming language must be so that these parts can be composed into ever larger systems, while also giving a practical way to combine the results of analysing the individual chunks. Obviously, the success of this approach is greatly determined by the ways possible for a given part to affect other parts.

The recent emergence of statically typed, pure functional programming languages is explained in part because they suit these two approaches so well. Their type systems are usually expressive enough to encode meaningful program properties, and referential transparency[6] assures a lack of non-local effects, thus enabling reasonings about parts of programs to be easily elevated to reasonings about the whole of the program.

But there is a price to pay for expressive type systems, and that price comes in the form of the mental capacity needed to use type checkers. Understanding the typing constraints in a dependently-typed functional language[7] like *Agda*[8], *Coq*[9] or *Epigram*[10] is so complicated that interactive tools have to be used to write well-typed programs. Mainstream functional programming languages like *Haskell*[11] or *Clean*[12] use a somewhat less expressive type system based on several extensions to the Hindley-Milner type system. One great advantage of these type systems is that they admit not just type checking, but also type inference.

Users of these programming languages have all encountered error messages from the type checker, and it is a well-known problem that these error messages can often be cryptic. *Helium*[13] is an implementation of the Haskell language addressing these problems by various heuristics[14].

One explanation of the difficulty of understanding type errors is that these type systems yield typing constraints that are non-compositional in the sense that the inferred type of a given expression is not just a function of its subexpressions, but is also influenced by the super-expression it appears in. In 2001, a paper by Olaf Chitil [15] presented a compositional approach to typing expressions in a Hindley-Milner-like type system.

In this thesis, we present an extension to the system described in Chitil's paper that allows for *ad-hoc* polymorphism, a feature extensively used in Haskell[16]. We have implemented this type system on top of the Glasgow Haskell Compiler[17], and the resulting type checker supports most of Haskell 98. The implementation is available under the BSD license at `http://gergo.erdi.hu/projects/tandoori/`.

Notations

Inference rule systems

We will define and examine a number of inference systems. Here, an inference system $\mathcal{S}$ is understood to be a collection of inference rules that are given in the form

$$\frac{\text{Statement } P}{\text{Statement } Q} \quad (\textsc{Rule})$$

where Rule is the name of the rule that indicates, given the statement P, that the statement Q also holds. A statement R is then inferable in $\mathcal{S}$ if there is a suitable tree of inference rule applications.

Rules can be axioms, meaning the P part is missing. Not all axioms are given in inference rule form: true statements that are outside the scope of the examined system are understood to be implicitly inferable. For example, when discussing type systems, we do not present a formal treatment of ZF set theory but use predicates like $x \in X$ freely.

Rules can also be rule schemas that contain variables. The domain of a variable is determined by its name and typesetting (see next section); for example, the rule schema

$$\frac{}{\Upsilon \vdash \alpha} \quad (\textsc{TyVar})$$

defines inference rules for all type variables α.

Environments

Some inferable statements are about certain properties that follow from some environment or context. In these cases, we will write

$$\Pi \vdash P$$

for these statements, where Π is the context and P is some other statement that follows (i.e. inferable) in that context.

Some statements are more about the generation of environments. If it makes more sense to think of a statement as "the property P holds, and also yields the environment Π", we will instead write

$$P \dashv \Pi.$$

In some cases, there is both a context and a resulting environment for some property P. In these cases, the above notations are combined into

$$\Pi \vdash P \dashv \Sigma.$$

Symbols and letters

Most of the symbols and functions used throughout the thesis should be either straightforward or defined at its first occurrence. In particular,

- $\operatorname{dom} f$ gives the domain of the function or finite mapping f,
- $\operatorname{vars} \tau$ is the set of all type variables occurring in τ.

The following table shows the typographical conventions used:

Roman typeface	Functions	dom
Boldface	Elements of expression syntax	**let** . . . **in** . . .
Slanted typeface	Type systems	*HM*, *C*
	Variables	*x*, *f*
Calligraphic typeface	Algorithms	$\mathcal{W}$
Small capital typeface	Inference rules	$\textsc{Abs}$
	(Type) constructors	$\textsc{true}$
Greek small letters	Elements of type systems	τ, α, θ
Greek capitals	Environments	Υ, Γ, Δ, Θ
Overline	List/vector	$\overline{\tau}$

1. The typed lambda calculus

In this chapter, we present a well-known type system for the lambda calculus. It is described here because several notations and customs are widespread and thus it is important to present the exact formulation that we'll use. The opportunity is also used to review some of the basic properties of the Hindley-Milner type system.

1.1 The formal language Λ

The variant of the lambda calculus that we'll explore is lambda calculus with algebraic datatypes, constructors and pattern matching. Its syntax is presented in figure 1.1. Note that in this and later presentations, we consider the list of datatype definitions as given *a priori*.

Expression:	E	=	v
		|	c
		|	$E\ E$
		|	$\lambda v \mapsto E$
		|	**case** E **of** $\overline{P \mapsto E}$
Pattern:	P	=	v
		|	$c\,\overline{P}$
Variable:	v	=	$f \mid x \mid y \mid \ldots$
Constructor:	c		

Figure 1.1: Syntax of our model language Λ

Since in the present work we are only interested in creating a type system for this language, we omit a formal description of the semantics of Λ and instead refer the reader to the numerous treatments of the material, e.g. [18].

For brevity's sake, we'll exploit the left-associativity of function application and right-associativity of λ-abstraction to omit unnecessary parentheses. Also, $\lambda x \mapsto \lambda y \mapsto E$ will be abbreviated as $\lambda x\ y \mapsto E$.

We will also assume every variable name to be unique. This can be easily ensured by appropriate α-conversions.

1.2 Type systems

What we expect of a type system for our language Λ is to statically, that is, without actually evaluating the expression, catch some class of nonsensical expressions. In contrast, a dynamic type system is one that only finds semantic contradictions while evaluating the expression in question.

To reason about the semantics of a given λ-expression without evaluating it, so-called types are attached to it and its subexpressions. A type system is a set of possible types and a collection of rules governing the way propositions of the kind "this given expression E has type τ" can be decided.

One can devise several type systems for Λ, differing in two important, and not at all orthogonal aspects. The first one is the kind of semantic errors that the type system can catch — the wider the range of detectable errors, the safer we can be that an expression accepted by the type checker is meaningful. The second one is the class of semantically correct expressions that the type system accepts. Ideally, we would want all semantically correct (i.e. meaningful and converging) expressions to be accepted by the type checker.

To see why there is a conflict between these two requirements, consider the notorious class of divergent expressions, like the following canonical example[19]:

$$\Omega := (\lambda x \mapsto x\ x)\ (\lambda x' \mapsto x'\ x')$$

Of course, given that Turing-completeness[20] is a property of Λ we very much intend to keep, we cannot hope to detect errors of this kind in general[21].

1.3 The *HM* type system

The type system we'll devise for Λ in this chapter is the Hindley-Milner type system[4]. For example, given the set of datatypes $\Upsilon := \{\textsc{bool} = \textsc{true} \mid \textsc{false}\}$, this type system can catch the error in the following expression:

$$E := (\lambda f \mapsto f\ \textsc{false})\ \textsc{true},$$

arising from the usage of x as some function with a domain in BOOL in the body of the λ-abstraction, versus supplying a value of type BOOL for x.

We detect this error by realizing that the expression is a function application with left subexpression $\lambda x \mapsto x$ FALSE being a function from "functions mapping BOOLS to something else" to that something else (written as $(\text{BOOL} \to \alpha) \to \alpha$), and the right subexpression being a value of type BOOL.

Figure 1.2 gives the syntax and inference rules for the types that we want the type system *HM* to assign to λ-expressions. Types are either type variables like α in the previous example, function types (with a domain type and a codomain type), or concrete datatypes. Datatypes can also have type parameters. The function arity in figure 1.2 gives the arity (number of type parameters) of type constructor T as a natural number, and is understood to use the information stored in the datatype context Υ that is preserved throughout the whole type checking process.

$$
\begin{array}{lcll}
\text{Type variable:} & \alpha & & \\
\text{Type constructor:} & T & & \\
\text{Simple type:} & \tau & = & \alpha \\
 & & \mid & T\ \overline{\tau} \\
 & & \mid & \tau \to \tau \\
\text{Polymorphic type:} & \sigma & = & \forall \overline{\alpha}.\tau
\end{array}
$$

$$\frac{}{\Upsilon \vdash \alpha} \quad (\textsc{TyVar})$$

$$\frac{\Upsilon \vdash \tau_1, \ldots, \Upsilon \vdash \tau_n \qquad \Upsilon \vdash \mathrm{arity}(T) = n}{\Upsilon \vdash T\ \tau_1 \ldots \tau_n} \quad (\textsc{TyCon})$$

$$\frac{\Upsilon \vdash \tau', \Upsilon \vdash \tau}{\Upsilon \vdash \tau' \to \tau} \quad (\textsc{TyFun})$$

$$\frac{\Upsilon \vdash \tau}{\Upsilon \vdash \forall \overline{\alpha}.\tau} \quad (\forall)$$

Figure 1.2: Syntax of and inference rules for valid types of *HM*

The Hindley-Milner type system, then, is a set of rules for deciding the validity of expressions. A given expressions E is accepted if it is typeable, that is, if the propositions $\Upsilon \vdash \tau$ and $\emptyset \vdash E :: \tau$ are provable for some τ

using the inference rules. We'll sometimes write $\vdash E :: \tau$ as shorthand for $\emptyset \vdash E :: \tau$.

In this chapter, E is an expression of Λ and τ is a valid type as defined in figure 1.2. Later, we will extend both Λ and the set of inference rules to support other features by allowing for somewhat different domains for E and τ.

Figure 1.3 on the following page shows the inference rules of *HM*. It uses a context Γ mapping variables to types to ensure consistent typing of variables.

1.4 Observations about *HM*

There are several things to note about these rules. The first is that a given λ-expression can inhabit many types. For example, given the following expression:

$$id := \lambda x \mapsto x,$$

the statements "*id* inhabits the type $\textsc{bool} \to \textsc{bool}$" and "*id* inhabits $(\alpha \to \beta) \to (\alpha \to \beta)$" and many more are provable. So it is natural to ask: what is the most general type that a given expression E inhabits? We'll say that the polymorphic type $\sigma = \forall\overline{\alpha}.\tau$ is the *principal type* of a given expression E if it fulfils the following requirements:

- It is indeed a correct type of E, i.e. $\vdash E :: \tau$
- It is a generalisation of every other type: for all τ', the statement $\vdash E :: \tau'$ holds if and only if $\tau' \in \mathrm{inst}(\sigma)$ (see figure 1.4 for the definition of inst)

It can be shown[22] that any typeable expression E has a unique principal type (up to renaming type variables and omitting superfluous type variables from $\overline{\alpha}$). For example, the principal type of *id* is $\forall\alpha.\alpha \to \alpha$.

The second important thing to note is that some inference rules making up the system *HM* (notably, Abs and Inst) are not constructive in the following sense: consider, for example, the following derivation of the principal type $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ for the expression $\lambda f\ x \mapsto f\ (f\ x)$:

$$\frac{(c :: \tau) \in \Upsilon \qquad \overline{\alpha} = \mathrm{vars}\ \tau}{\Gamma \vdash c :: \forall \overline{\alpha}.\tau} \quad (\textsc{Con})$$

$$\frac{(x :: \tau) \in \Gamma}{\Gamma \vdash x :: \tau} \quad (\textsc{MonoVar})$$

$$\frac{\Gamma \vdash E :: \sigma \qquad \tau \in \mathrm{inst}(\sigma)}{\Gamma \vdash E :: \tau} \quad (\textsc{Inst})$$

$$\frac{\Gamma \vdash E :: \tau' \to \tau \qquad \Gamma \vdash F :: \tau'}{\Gamma \vdash E\ F :: \tau} \quad (\textsc{App})$$

$$\frac{\Gamma; (x :: \tau') \vdash E :: \tau \qquad \Upsilon \vdash \tau'}{\Gamma \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

$$\frac{\begin{array}{rrl} & \Gamma \vdash & E :: \tau_0 \\ P_1 :: \tau_0 \dashv \Gamma_1 & \Gamma; \Gamma_1 \vdash & E_1 :: \tau \\ & & \vdots \\ P_n :: \tau_0 \dashv \Gamma_n & \Gamma; \Gamma_n \vdash & E_n :: \tau \end{array}}{\Gamma \vdash \textbf{case}\ E\ \textbf{of}\ P_1 \mapsto E_1 \ldots P_n \mapsto E_n :: \tau} \quad (\textsc{Case})$$

$$\frac{\Upsilon \vdash \tau}{x :: \tau \dashv (x :: \tau)} \quad (\textsc{VarPat})$$

$$\frac{\begin{array}{c} (c :: \sigma) \in \Upsilon \quad (\tau_1 \to \cdots \to \tau_n \to T\ \overline{\tau}) \in \mathrm{inst}(\sigma) \\ P_1 :: \tau_1 \dashv \Gamma_1 \quad \cdots \quad P_n :: \tau_n \dashv \Gamma_n \end{array}}{c\ P_1 \ldots P_n :: T\ \overline{\tau} \dashv \Gamma_1; \ldots; \Gamma_n} \quad (\textsc{ConPat})$$

Figure 1.3: *HM* inference rules for Λ

$$\frac{\Psi\tau = \tau' \qquad \mathrm{dom}\,\Psi = \overline{\alpha}}{\tau' \in \mathrm{inst}(\forall\overline{\alpha}.\tau)}$$

Figure 1.4: Inference rule for the inst relation

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_2 \vdash f \,::\, \alpha \to \alpha \qquad \Gamma_2 \vdash x \,::\, \alpha}{\Gamma_2 \vdash f\ x \,::\, \alpha} \qquad \Gamma_2 \vdash f \,::\, \alpha \to \alpha}{\Gamma_2 \vdash f\ (f\ x) \,::\, \alpha}}{\Gamma_1 \vdash \lambda x \mapsto f\ (f\ x) \,::\, \alpha \to \alpha}}{\emptyset \vdash \lambda f\ x \mapsto f\ (f\ x) \,::\, (\alpha \to \alpha) \to (\alpha \to \alpha)}$$

where

$$\begin{aligned}
\Gamma_1 &= (f \,::\, (\alpha \to \alpha)) \\
\Gamma_2 &= \Gamma_1; (x \,::\, \alpha)
\end{aligned}$$

As we can see, the derivation is straightforward only once we know a good choice for Γ_1 and Γ_2, which corresponds to choosing τ' when applying rule Abs. However, coming up with the correct types when entering a λ-abstraction is just the problem of type inference that we want to solve!

Fortunately, the situation is not as bad as this circular-looking reasoning would make one believe, and we will later describe in detail the well-known algorithms $\mathcal{W}$ and $\mathcal{M}$ that implement type inference by solving this problem.

2. Let-polymorphism

In this chapter, an important and widespread extension of the language Λ and its typing rules are presented: recursive, polymorphic let bindings.

2.1 The Λ^{let} language

Figure 2.1 shows syntax extensions of the Λ language presented in figure 1.1, with the changes highlighted.

$$
\begin{array}{llcl}
\text{Expression:} & E & = & v \\
 & & \mid & c \\
 & & \mid & E\ E \\
 & & \mid & \lambda v \mapsto E \\
 & & \mid & \textbf{case}\ E\ \textbf{of}\ \overline{P \mapsto E} \\
 & & \mid & \textbf{let}\ \overline{D}\ \textbf{in}\ E \\
\text{Definition:} & D & = & v\ \overline{P} = E \\
\text{Pattern:} & P & = & v \mid c\ \overline{P} \\
\text{Variable:} & v & = & f \mid x \mid y \mid \ldots \\
\text{Constructor:} & c & &
\end{array}
$$

Figure 2.1: Syntax of Λ^{let}

The new construct **let** allows for local function definitions[1]. This is an important difference in our definition in contrast to usual treatments of the subject, which only allow variables to be defined inside a **let**. Since in this thesis we intend to arrive at practical results for reporting type errors, we choose this definition of **let** that is a lot closer to the facilities offered by real-life functional programming languages.

Functions are defined using pattern matching on the arguments, like *map*

[1]Variables can be defined as functions with no arguments

in the following definition, with $\Upsilon = \{[\alpha] = \textsc{nil} \mid \textsc{cons}\ \alpha\ [\alpha]\}$:

$$\begin{aligned}
\textbf{let}\ \ & \mathit{map}\ \ f\ \ \textsc{nil} && = \textsc{nil} \\
& \mathit{map}\ \ f\ \ (\textsc{cons}\ x\ xs) && = \textsc{cons}\ (f\ x)\ (\mathit{map}\ f\ xs) \\
\textbf{in}\ & \mathit{map}.
\end{aligned}$$

2.2 Recursive definitions

Recursion, as demonstrated by the previous example of *map*, is an explicitly desired property of **let**, in the sense that local definitions should be able to refer to themselves.

The following example demonstrates a technique for transforming mutually recursive definitions into straight recursion by combining the definitions into a single tuple; both expressions yield the infinite list $\langle \textsc{true}, \textsc{false}, \textsc{true}, \textsc{false}, \ldots \rangle$:

$$\Upsilon := \{(\alpha, \beta) = (\alpha, \beta),\ [\alpha] = \textsc{nil} \mid \textsc{cons}\ \alpha\ [\alpha],\ \textsc{bool} = \textsc{true} \mid \textsc{false}\}$$

$$\begin{aligned}
\mathit{mutual} := \textbf{let}\ & \mathit{tick} = \textsc{cons}\ \textsc{true}\ \mathit{tock} \\
& \mathit{tock} = \textsc{cons}\ \textsc{false}\ \mathit{tick} \\
\textbf{in}\ & \mathit{tick}
\end{aligned}$$

$$\begin{aligned}
\mathit{flattened} := \textbf{let}\ & \mathit{proj}_1^2\ \ (x, y) = x \\
& \mathit{proj}_2^2\ \ (x, y) = y \\
\textbf{in}\ \textbf{let}\ & \mathit{ticktock} = \textbf{let}\ \mathit{tick} = \textsc{cons}\ \textsc{true}\ (\mathit{proj}_2^2\ \mathit{ticktock}) \\
& \qquad\qquad\qquad \mathit{tock} = \textsc{cons}\ \textsc{false}\ (\mathit{proj}_1^2\ \mathit{ticktock}) \\
& \qquad\qquad\quad \textbf{in}\ (\mathit{tick}, \mathit{tock}) \\
\textbf{in}\ & \mathit{proj}_1^2\ \mathit{ticktock}
\end{aligned}$$

This transformation can be easily automated by detecting mutually recursive definitions (by searching for all strongly connected components of the variable reference graph). By avoiding mutual recursion, it is also enough to allow only a single function definition (with multiple patterns, of course) in one **let**, because multiple definitions can be transformed into nested **let**s in the order determined by the topological sorting of the aforementioned graph.

We will use these observations in later chapters, when detailing type inference algorithms for Λ^{let} , by assuming singular recursive definitions in **let** without loss of generality, by presuming an appropriate transformation step before type checking.

2.3 *HM* type inference for Λ^{let}

Here we extend the type system *HM* with additional rules to support the new **let** construct. First of all, just as the inference rule CASE needed the rules VARPAT and CONPAT to collect the newly bound variables in a pattern, we need rules to collect the variables bound in a **let**:

$$\frac{P_1 :: \tau_1 \dashv \Gamma_1 \quad \cdots \quad P_n :: \tau_n \dashv \Gamma_n \qquad \Gamma; \Gamma_1; \ldots; \Gamma_n \vdash E :: \tau}{\Gamma \vdash f\ P_1 \ldots P_n = E \dashv (f :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau)} \quad (\textsc{Def})$$

Figure 2.2: Typing rules for collecting **let**-bound variables

Given the requirement for allowing recursion, the most straightforward type inference rule one can give for **let** is to require the newly bound definitions to be typeable in the very context that they define, and type the body of the **let** in the same context:

$$\frac{\begin{array}{l}\Gamma' = \Gamma_1; \ldots; \Gamma_n \\ \Gamma; \Gamma' \vdash D_1 \dashv \Gamma_1 \quad \cdots \quad \Gamma; \Gamma' \vdash D_n \dashv \Gamma_n \\ \Gamma; \Gamma' \vdash E :: \tau\end{array}}{\Gamma \vdash \textbf{let}\ D_1 \ldots D_n\ \textbf{in}\ E :: \tau} \quad (\textsc{MonoLet})$$

The actual rule LET, as shown in figure 2.3 on the following page, differs from this rule because we want let-bound variables to be *polymorphic* inside the body of the **let**. For example, the following expression is not typeable without let-polymorphism, because the two occurrences of *id* need to be assigned the non-unifiable types BOOL $\rightarrow$ BOOL and $[\alpha] \rightarrow [\alpha]$:

let *id* $x = x$ **in** P (*id* TRUE) (*id* NIL).

To achieve this, the types of let-bound variables are generalised (by adding an appropriate $\forall$ quantifier) in the context of the body. Of course, for polymorphic variables to be usable, we also have to add a new inference rule for polymorphic variables, that mirrors the CON rule, for instantiating polymorphic types.

$$\frac{(x :: \sigma) \in \Gamma \qquad \tau \in \text{inst}(\sigma)}{\Gamma \vdash x :: \tau} \quad (\textsc{PolyVar})$$

$$\frac{\begin{array}{l} \Gamma' = \Gamma_1; \ldots; \Gamma_n \\ \Gamma; \Gamma' \vdash D_1 \dashv \Gamma_1 \quad \cdots \quad \Gamma; \Gamma' \vdash D_n \dashv \Gamma_n \\ \Gamma'' = \{x :: \sigma | (x :: \tau) \in \Gamma', \sigma = \text{gen}(\Gamma, \tau)\} \\ \Gamma; \Gamma'' \vdash E :: \tau \end{array}}{\Gamma \vdash \textbf{let } D_1 \ldots D_n \textbf{ in } E :: \tau} \quad (\textsc{Let})$$

Figure 2.3: New typing rules for Λ^{let}

3. Type inference for Hindley-Milner type systems

A type inference algorithm is one that, given an expression E and an initial environment Γ_0, either returns a positive result of a type τ such that $\Gamma_0 \vdash E : \tau$, or a negative result of some error message about E not being typeable. As an added requirement, we also expect algorithms for *HM* to return, in the positive case, the principal type.

Two important properties of type inference algorithms are *soundness* and *completeness*. A given algorithm is sound if its positive answer is always correct, and it is complete if it returns a negative answer only when the given expression is not typeable in the given type system.

3.1 Algorithm $\mathcal{W}$

Milner's original paper presenting the (let-polymorphic) Hindley-Milner type system[4] included algorithm $\mathcal{W}$ that computes the principal type of a given expression. The algorithm works by always assigning new type variables to the parameters of λ-abstractions, and collecting type substitutions on the way. The type substitutions are generated by solving type constraints arising from applications. Somewhat later, Damas[23] proved algorithm $\mathcal{W}$ to be complete.

The general form of $\mathcal{W}$ is the following:

$$\boxed{\begin{array}{lrl} \mathcal{W}(\Gamma, E) = (\Psi, \tau) & & \\ \quad \text{where} & \Gamma : & \text{a type context, mapping variables to types} \\ & E : & \text{the expression whose type we are to infer} \\ & \Psi : & \text{a substitution, mapping type variables to types} \\ & \tau : & \text{the inferred type of } E \end{array}}$$

Recurrences in the definition of $\mathcal{W}$ always refer to smaller subexpressions; thus, the definition is well-founded.

We omit the definition of $\mathcal{W}$ for **case** and **let** expressions here, and only focus on the core language of variables, function application and λ-abstraction.

3.1.1 Variables

$$
\begin{array}{ll}
\mathcal{W}(\Gamma, x) = (\emptyset, \tau) & \text{if } (x :: \tau) \in \Gamma \\
\mathcal{W}(\Gamma, x) = (\emptyset, \{\overline{\alpha} \rightsquigarrow \overline{\beta}\}\tau) & \text{if } (x :: \forall\overline{\alpha}.\tau) \in \Gamma \\
\qquad \text{where } \ \overline{\beta} \text{ new} &
\end{array}
$$

The inference rules for variable occurrences is simply a matter of looking up the variable in the type context Γ. Variables with polymorphic types are instantiated differently for every occurrence. Since a variable occurrence, by itself, imposes no type constraints, no substitution is required.

3.1.2 λ-abstraction

$$
\begin{array}{l}
\mathcal{W}(\Gamma, \lambda x \mapsto E) = (\Psi, \Psi\beta \to \tau) \\
\qquad \text{where} \quad (\Psi, \tau) = \mathcal{W}(\Gamma; (x :: \beta), E) \\
\qquad\qquad\qquad\quad \beta \text{ new}
\end{array}
$$

Inferring the type of a λ-abstraction entails inferring the type of its body, using an extended type context. The variable of the λ-abstraction is inserted as a monomorphic variable into the context; recall that we assume all variable names to be distinct (by a separate scoping transformation before type inference), so there can be no conflicts when adding $(x :: \beta)$ to Γ.

Constraints on the type of the argument can, of course, be imposed by the body of the λ-abstraction. These constraints ultimately generate substitutions of the form $\beta \rightsquigarrow \tau'$, which is then applied to the left-hand side of the function type returned by $\mathcal{W}(\Gamma, \lambda x \mapsto E)$ to produce the type $\tau' \to \tau$.

3.1.3 Function application

$$
\begin{aligned}
\mathcal{W}(\Gamma, E\ F) &= (\Psi \circ \Psi_2 \circ \Psi_1, \Psi\beta) \\
\text{where} \quad (\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1\Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2\tau_1 \sim \tau_2 \to \beta) \\
\beta &\text{ new}
\end{aligned}
$$

To infer the type of applying the function E on the operand F, the type of both E and F needs to be inferred. The compatibility requirement $\tau_E \sim \tau_F \to \tau_{E\ F}$ is a straightforward transliteration of the type inference rule App.

Since both E and F may refer to the same variables, constraints arising from one can affect the other. This is captured by the fact that $\mathcal{W}$ recurses on F with a modified environment $\Psi_1\Gamma$. In the next chapter, we shall see how this is an important point with regards to reporting type errors.

3.2 Algorithm $\mathcal{M}$

There is no clear origin of algorithm $\mathcal{M}$, another type assignment algorithm for Hindley-Milner type systems; Lee and Yi presented a formal treatment of the by-then widespread folklore algorithm in [24], proving its soundness and completeness. Their paper also proves an advantage of $\mathcal{M}$, which is that $\mathcal{M}$ stops earlier than $\mathcal{W}$ for non-typeable inputs.

The basic idea behind algorithm $\mathcal{M}$ is that instead of the $\mathcal{M}$ function returning the inferred type of subexpressions, the expected type is passed to it as an argument. The well-foundedness of the recursion is, again, ensured by always recursing on smaller subexpressions.

$$
\begin{aligned}
\mathcal{M}(\Gamma, E, \tau) &= \Psi \\
\text{where} \quad \Gamma &: \text{ a type context, mapping variables to types} \\
E &: \text{ the expression to typecheck} \\
\tau &: \text{ the expected type of } E \\
\Psi &: \text{ a substitution, mapping type variables to types}
\end{aligned}
$$

3.2.1 Variables

The inference rule for variables closely mirrors that of $\mathcal{W}$: the type of the variable as stored in Γ is checked against the expected type from the function argument:

$$
\begin{array}{ll}
\mathcal{M}(\Gamma, x, \tau) = \mathcal{U}(\tau \sim \tau') & \text{if } (x :: \tau') \in \Gamma \\
\mathcal{M}(\Gamma, x, \tau) = \mathcal{U}(\tau \sim \{\overline{\alpha} \rightsquigarrow \overline{\beta}\}\tau') & \text{if } (x :: \forall \overline{\alpha}.\tau') \in \Gamma \\
\qquad \text{where } \overline{\beta} \text{ new} &
\end{array}
$$

3.2.2 λ-abstraction

$$
\begin{array}{l}
\mathcal{M}(\Gamma, \lambda x \mapsto E, \tau) = \Psi_2 \circ \Psi_1 \\
\qquad \text{where} \quad \Psi_1 = \mathcal{U}(\tau \sim \alpha \to \beta) \\
\qquad \qquad \qquad \Psi_2 = \mathcal{M}(\Psi_1\Gamma; (x :: \Psi_1\alpha), E, \Psi_1\beta) \\
\qquad \qquad \qquad \quad \alpha, \beta \text{ new}
\end{array}
$$

Since λ-abstractions are always typed as $\tau_1 \to \tau_2$, the expected type must have this schema as well. This is expressed by the first unification step that deconstructs τ into an argument type α and a result type β. Note that the results of this unification can impose additional constraints on the type of variables in Γ, which is why it is the substituted context $\Psi_1\Gamma$ that is extended before recursing into E.

3.2.3 Function application

$$
\begin{array}{l}
\mathcal{M}(\Gamma, E\ F, \tau) = \Psi_2 \circ \Psi_1 \\
\qquad \text{where} \quad \Psi_1 = \mathcal{M}(\Gamma, E, \beta \to \tau) \\
\qquad \qquad \qquad \Psi_2 = \mathcal{M}(\Psi_1\Gamma, F, \Psi_1\beta) \\
\qquad \qquad \qquad \quad \beta \text{ new}
\end{array}
$$

For the application $E\ F$ to have type τ, E must be of some type $\tau' \to \tau$, and F of type τ'. This connection between E and F is implemented in $\mathcal{M}$ via the shared type variable β.

3.3 Unification

The type constraint solver used by $\mathcal{W}$ and $\mathcal{M}$ is not at all specific to these type inference algorithms: a simplified special-case version of the universal predicate resolution algorithm described in [25] is used to calculate the most generic unifier for a set of type equations of the form $\tau \sim \tau'$. Figure 3.1 shows the definition of $\mathcal{U}$; this function is shared by all type inference algorithms presented here, including the compositional algorithm introduced in chapter 5.

Note that although the set of equations can grow when recursing, there is a simple tree measure of types with which the total measure of the set of equations is always strictly decreasing, thus ensuring well-foundedness.

$$
\begin{aligned}
\mathcal{U}(\emptyset) &= \emptyset \\
\mathcal{U}(\{\alpha \sim \alpha\} \cup \Sigma) &= \mathcal{U}(\Sigma) \\
\mathcal{U}(\{\alpha \sim \tau\} \cup \Sigma) &= \begin{cases} \text{error: Infinite type} & \text{if } \alpha \in \operatorname{vars} \tau \\ \mathcal{U}(\Psi\Sigma) \circ \Psi & \text{otherwise} \end{cases} \\
&\quad \text{where } \Psi = \{\alpha \rightsquigarrow \tau\} \\
\mathcal{U}(\{\tau \sim \alpha\} \cup \Sigma) &= \mathcal{U}(\{\alpha \sim \tau\} \cup \Sigma) \\
\mathcal{U}(\{\tau \to u \sim \tau' \to u'\} \cup \Sigma) &= \mathcal{U}(\Sigma \cup \{\tau \sim \tau', u \sim u'\}) \\
\mathcal{U}(\{T\ \tau_1 \ldots \tau_n \sim T\ \tau'_1 \ldots \tau'_n\} \cup \Sigma) &= \mathcal{U}(\Sigma \cup \{\tau_1 \sim \tau'_1, \ldots, \tau_n \sim \tau'_n\}) \\
\mathcal{U}(\{\tau \sim \tau'\} \cup \Sigma) &= \text{error: Contradicting constraints}
\end{aligned}
$$

Figure 3.1: Calculating a most generic unifier

4. Reporting and explaining type errors

In this chapter, we take a brief detour to analyse the error reporting capabilities of $\mathcal{W}$ and $\mathcal{M}$. Problems discovered here will serve as our motivation for a compositional type system in the next chapter.

4.1 Linearity

Both $\mathcal{W}$ and $\mathcal{M}$ infer the type of composite expressions by inferring one subexpression (in some sense, the "first" one) and using its results in inferring the type of the next one. They are *linear* in the sense that partial results are threaded throughout the type inference.

For example, recall the definition of $\mathcal{W}$ for applications:

$$
\begin{aligned}
\mathcal{W}(\Gamma, E\ F) &= (\Psi \circ \Psi_2 \circ \Psi_1, \Psi\beta) \\
\text{where}\quad (\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1\Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2\tau_1 \sim \tau_2 \to \beta) \\
\beta &\ \text{new}
\end{aligned}
$$

It first recurses on the left-hand expression E in context Γ, and then uses the result of this inference, Ψ_1, to recurse on F. If E and F are both well-typed, but there is a contradiction between the two (by e.g. not agreeing on the type of a variable in Γ), this error will always be discovered, and reported, as a fault with F, even though F would be well-typed by itself.

4.2 Errors discovered linearly

The effect of linearity on type inference is that certain subexpressions (those that are processed earlier) can have greater influence on the typing of other subexpressions. This is bad because it imposes a hierarchy on the subexpressions that is determined solely by the actual type checking algorithm, not by the type system; thus, it can lead to misleading error messages for the programmer.

For example, consider the following program:

$$\begin{aligned}\Upsilon &:= \{\, (\alpha, \beta) = (\alpha, \beta),\\ &\qquad \textsc{char} = \texttt{'A'} \mid \ldots \mid \texttt{'Z'} \mid \texttt{'a'} \mid \ldots \mid \texttt{'z'},\\ &\qquad \textsc{bool} = \textsc{true} \mid \textsc{false}\}\\ \Gamma_0 &:= \{\mathit{toUpper} :: \textsc{char} \to \textsc{char},\ \mathit{not} :: \textsc{bool} \to \textsc{bool}\}\end{aligned}$$

$$\mathit{test} := \lambda\, x \mapsto (\mathit{toUpper}\ x, \mathit{not}\ x)$$

The critical part is the application of $((\cdot\,,\ \cdot)\ (\mathit{toUpper}\,x))$ on $(\mathit{not}\,x)$. $\mathcal{W}$ will first assign some type variable α to x, then solve the type equation $\alpha \sim \textsc{char}$ generated by *toUpper x*, yielding the substitution $\alpha \rightsquigarrow \textsc{char}$. This leads to $\Gamma' = \{x :: \textsc{char}\}$ going into the second part of the tuple, generating the unsolvable type equation $\textsc{char} \sim \textsc{bool}$. Thus, the second part, *not x*, is what is reported to be not well-typed.

We could have changed the order in which $\mathcal{W}$ traverses the subexpressions of function application (and as we'll see below, some implementations use this order), but that would have merely resulted in entering *toUpper x* with $\Gamma' = \{x :: \textsc{bool}\}$, resulting in the unsolvable type equation $\textsc{bool} \sim \textsc{char}$.

4.3 A survey of Haskell compilers

Below is the output of some popular Haskell compilers demonstrating the above, when trying to compile the following program:

```
test x = (toUpper x, not x)
```

All three of them correctly discover the type error, but they all report it as though it would be a problem with one of the parts of the tuple *per se*, not the combining of the two.

- GHC[17] 6.12: The subexpression `toUpper x` is processed first. The error message shows *x* to be typed CHAR.

  ```
  Couldn't match expected type 'Bool'
         against inferred type 'Char'
  In the first argument of 'not', namely 'x'
  In the expression: not x
  In the expression: (toUpper x, not x)
  ```

- Hugs 98[26] seems to process application in the reversed order: `not x` is checked first, leading to an error in `toUpper x`:

  ```
  ERROR "test.hs":1 - Type error in application
  *** Expression     : toUpper x
  *** Term           : x
  *** Type           : Bool
  *** Does not match : Char
  ```

- Helium[13] 1.6 gives the same result as GHC above. The output is more verbose, showing the inferred type of both the applied expression and the argument; but otherwise it presents the same judgement that the expression `not x` is at fault by itself.

  ```
  (1,29): Type error in application
   expression        : not x
   function          : not
     type            : Bool -> Bool
   1st argument      : x
     type            : Char
     does not match  : Bool
  ```

4.4 Explanation of type judgements

When the programmer is presented with an error message from the type checker, fixing the problem requires understanding the result of the type inference. The problem with linear type inference algorithms is that type judgements for subexpressions cannot be explained without reference to other subexpressions.

Presented with the above error messages, the programmer might wonder why the expression *not x* is not well-typed. It certainly looks well-typed by itself. Also, there is no apparent reason for the judgement x :: CHAR, referred to in the error messages. Focusing on *not x*, the expression where the error is reported, gives no insight on the underlying problem.

Showing the programmer the type context $\Gamma = \{x :: \text{CHAR}\}$ is not of much help, either. While it does explain the source of the type equation $\text{CHAR} \sim \text{BOOL}$, the programmer is still left out in the cold about the actual problem: with this Γ, it is trivial to see that *not x* is not well-typed, but this only leads to frustration because now it is clear that the problem stems not from the expression reported by the type checker as the source of the error.

4.5 Other aspects of type error reporting

Of course, there are many kinds of type errors and many possible presentations of them. Here, we have focused on examples where linearity is of great hindrance, because this is what we intend to fix; [15] gives another great example.

As for possible improvements of type checkers for *HM*, [14] presents a thorough overview of the challenges and solutions of reporting good type error messages while staying inside the conceptual framework of *HM*, and thus, linearity.

5. A compositional type system

In this chapter, we present an alternative type system for the Λ^{let} language. Its main distinguishing property is that it allows for a compositional type checking algorithm, in contrast to the linear nature of $\mathcal{W}$ and $\mathcal{M}$. As we'll see, this can remedy some of the problems discussed in the previous chapter.

We define here a compositional type system as one where the type of subexpressions is not dependent on other expressions that are either above or beside it. Of course, this cannot be achieved in the framework of the Hindley-Milner type system; for example, consider the following expression:

$$\textbf{let}\ \textit{id}\ x = x\ \textbf{in}\ \boxed{\textit{id}}\ \textsc{true},$$

in which the type of the marked occurrence of *id* is very much determined by the type of its sibling expression TRUE. Hence the need for not just another algorithm, but a whole alternative type system.

5.1 Motivation

A compositional type system assigns types to expressions independent of their surrounding. Following up on our previous example presented in chapter 4:

$$\begin{aligned}\Upsilon := \{\, & (\alpha, \beta) = (\alpha, \beta), \\ & \textsc{char} = \texttt{'A'} \mid \ldots \mid \texttt{'Z'} \mid \texttt{'a'} \mid \ldots \mid \texttt{'z'}, \\ & \textsc{bool} = \textsc{true} \mid \textsc{false}\} \\ \Gamma_0 := \{ & \textit{toUpper} :: \textsc{char} \rightarrow \textsc{char},\ \textit{not} :: \textsc{bool} \rightarrow \textsc{bool}\}\end{aligned}$$

$$\textit{test} := \lambda\, x \mapsto (\textit{toUpper}\ x, \textit{not}\ x),$$

we expect a compositional type system to assign meaningful results to the subexpressions *toUpper x* and *not x*, since both of these are well-typed *by themselves*. Using this information, the programmer can then realize that the real cause of the error is ununifiable view of the two expressions on what the type of *x* should be; the programmer can then see both views and then decide how to fix the problem.

Our implementation of the type system described in this chapter outputs the following error message for the program above:

```
input/test.hs:1:8-25:
(toUpper x, not x)
Cannot unify 'Char' with 'Bool' when unifying 'x':
       toUpper x    not x
       Char         Bool
x ::   Char         Bool
```

Note that the type of `toUpper x` and `not x` is correctly inferred despite the problems with assigning a type to `x`; this enables type checking to go on, finding other problems in the same run.

5.2 Typings

The key to our compositional type system is the notion of *typings*[27]. A typing captures all of the constraints imposed by an expression on its environment. For example, for the expression $f\ x\ y$ to be well-typed, we need f to be a binary function, and the type of its arguments must match that of x and y:

Type of expression:	$f\ x\ y :: \gamma$
Constraints on variables:	$f :: \alpha \to \beta \to \gamma$
	$x :: \alpha$
	$y :: \beta$

We defined Λ^{let} to only allow simple recursion: when an expression references a variable, it is either a function argument, a previously defined function, or a recursive reference. The type of functions already defined cannot be changed by a usage site, so that case yields no constraints, and function arguments and recursive references are monomorphic. This means that the typing of an expression contains constraints only on the type of monomorphic variables.

Using the notation of [15], a typing, written $\Delta \vdash \tau$, consists of a type environment $\Delta = \{x :: \tau_1, y :: \tau_2, \ldots\}$ mapping variables to monomorphic types, and a type τ. Our previous example $f\ x\ y$ thus has typing $\{f :: \alpha \to \beta \to \gamma,$ $x :: \alpha,\ y :: \beta\} \vdash \gamma$.

5.3 Polymorphic and monomorphic variables

As we have previously seen, the type environment part of a typing contains only monomorphic variables. Let-bound variables, however, are polymorphic, and only the type of their specific occurrence can be influenced by a usage site.

Thus, we split the variables into two groups: polymorphic and monomorphic. Polymorphic variables are associated with a typing, and that typing is instantiated at usage sites, yielding no additional constraints except those in the typing itself. Monomorphic variables are associated with a monomorphic type, and all usage sites of a monomorphic variable must agree on its type.

The type of polymorphic variables is stored in the polymorphic environment $\Gamma = \{f \mapsto \Delta \vdash \tau, \ldots\}$; this environment is extended by bindings in **let** expressions. Monomorphic variables are stored in the monomorphic environment $\Delta = \{x :: \tau', \ldots\}$. This environment is extended by every variable occurrence, and every expression unifies the monomorphic environments of its subexpressions.

Why do we need to store typings in the polymorphic environment instead of just types? It is because the definition of a let-bound variable can refer to monomorphic variables, and thus its usage can impose additional constraints. Consider the following example:

$$\Upsilon := \{\textsc{Char} = \ldots,\ \textsc{Bool} = \ldots,\ (\alpha, \beta) = \ldots,\ [\alpha] = \ldots\}$$
$$\Gamma_0 := \{\mathit{map} \mapsto \emptyset \vdash (\alpha \to \beta) \to [\alpha] \to [\beta],\ \mathit{toUpper} \mapsto \ldots,\ \mathit{not} \mapsto \ldots\}$$

$$\begin{aligned} \mathit{test} := \lambda\ \mathit{xs} \mapsto\ & \textbf{let}\ \mathit{xform}\ f = \mathit{map}\ f\ \mathit{xs} \\ & \textbf{in}\ (\mathit{xform}\ \mathit{toUpper}, \mathit{xform}\ \mathit{not}) \end{aligned}$$

The typing of *xform* after the generalisation is $\{\mathit{xs} :: [\alpha]\} \vdash (\alpha \to \beta) \to [\beta]\}$. This typing can be instantiated to both $\{\mathit{xs} :: [\textsc{Char}]\} \vdash (\textsc{Char} \to \gamma) \to [\gamma]$ and $\{\mathit{xs} :: [\textsc{Bool}]\} \vdash (\textsc{Bool} \to \delta) \to [\delta]$ for the two usages, and both maintain a constraint on the type of the monomorphic variable *xs*.

This also eliminates the need for explicit $\forall$ quantification, because *every* type variable is renamable in a typing. The difference between a polymorphic and a monomorphic variable is that the latter contains itself in its typing, thus, its instantiated typing will still be monomorphic.

In the example above, the monomorphic type environment of *xform* contains types in which α occurs, thus, it is not polymorphic in α. This is witnessed by the fact that all usages of *xform* must agree on the specific instantiation of α, but not on β.

5.4 Unification of typings

By now, it should be clear that the crucial step in the type system we're describing is the unification of typings. We extend the unification algorithm from chapter 3 by allowing an additional parameter consisting of a list of monomorphic environments $\Delta_1, \ldots, \Delta_n$. Since monomorphic type environments must agree on the type of all variables involved, from this list we generate a set of type equations expressing this desired congruity:

$$\boxed{\begin{aligned}&\mathcal{U}(\{\Delta_1, \ldots, \Delta_n\}, \Sigma) = \mathcal{U}(\Sigma' \cup \Sigma)\\ &\qquad \text{where } \ \Sigma' = \{\alpha(x) \sim \Delta_i(x) \mid i = 1 \ldots n,\ x \in \operatorname{dom} \Delta_i,\ \alpha(x) \text{ new}\}\end{aligned}}$$

5.5 Type system *C*

The type system *C* we present here is a type system for Λ^{let} based on [15]. Contrary to *HM*, it is constructive; thus, its inference rules can also be read as an algorithm, with judgement $\Gamma; \Delta \vdash E :: \tau$ interpreted as a function mapping the pair of a polymorphic context and an expression (Γ, E) to a typing $\Delta \vdash \tau$. Since the predicates of the inference rules always refer to smaller expressions, the recursion defined by this reading of the inference rules is not well-founded.

We make no claims about the performance of *C* as a type inference algorithm. If performance turns out to be poor in real-world applications, one can run one of either $\mathcal{M}$ or $\mathcal{W}$ to see if there are any type errors, and then run *C* only when errors are found.

5.5.1 Constants and variables

Constructors impose no constraints on monomorphic variables. References to previously defined, and thus polymorphic variables import the respective monomorphic type environment. We instantiate typings by consistently renaming type variables into fresh ones.

$$\frac{(c :: \tau) \in \Upsilon \qquad \emptyset \vdash \tau' = \mathrm{inst}\,\emptyset \vdash \tau}{\Gamma; \emptyset \vdash c :: \tau'} \quad (\textsc{Con})$$

$$\frac{\Gamma(x) = \Delta \vdash \tau \qquad \Delta' \vdash \tau' = \mathrm{inst}\,\Delta \vdash \tau}{\Gamma; \Delta' \vdash x :: \tau'} \quad (\textsc{PolyVar})$$

Monomorphic variables are those that have no associated typings in the polymorphic type environment Γ. Such occurrences are typed by recording the newly-introduced monomorphic variable in Δ.

$$\frac{x \notin \mathrm{dom}\,\Gamma \qquad \alpha \text{ new}}{\Gamma; \{x :: \alpha\} \vdash x :: \alpha} \quad (\textsc{MonoVar})$$

Note that this formulation doesn't detect out-of-scope references. Scope checking is assumed to have been done in a previous pass, because it is already needed to flatten out **let** declarations into non-mutually-recursive partitions.

5.5.2 λ-abstraction

The variable of a λ-abstraction is monomorphic in the abstraction's body. This means the polymorphic context Γ is not extended by the argument variable before descending into the body; instead, the type of the λ-abstraction is determined by looking at the monomorphic type of the variable from the typing of the body.

$$\frac{\Gamma; \Delta \vdash E :: \tau \qquad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

$$\frac{\Gamma; \Delta \vdash E :: \tau \qquad x \notin \operatorname{dom} \Delta \qquad \alpha \text{ new}}{\Gamma; \Delta \vdash \lambda x \mapsto E :: \alpha \to \tau} \quad (\textsc{Abs}')$$

The two cases correspond to λ-abstractions with and without references to their argument. In the former case, the argument variable is removed from the typing of the body in the resulting typing, since different usages of the same λ-abstraction don't have to agree on the argument type.

Consider the following example:

$$\Upsilon := \{\textsc{bool} = \ldots, (\alpha, \beta) = \ldots, [\alpha] = \ldots\}$$
$$\textit{idPair} := \textbf{let } \textit{id} = \lambda x \mapsto x \textbf{ in } (\textit{id } \textsc{true}, \textit{id } \textsc{nil}).$$

The inferred typing of x is $\{x :: \alpha\} \vdash \alpha$; thus, the typing of *id* would be $\{x :: \alpha\} \vdash \alpha \to \alpha$ if x wasn't removed from the monomorphic environment. This would lead to the contradicting typings $\{x :: \textsc{bool}\} \vdash \textit{id } \textsc{true} :: \textsc{bool}$ and $\{x :: [\beta]\} \vdash \textit{id } \textsc{nil} :: [\beta]$; in effect, losing let-polymorphism.

5.5.3 Function application

The inference rule for function application consists simply of inferring the typing of the function and the argument, and unifying the two. However, behind this simplicity lies compositionality: unlike *HM*, there is no directionality and no flow of information between the descendings into E and F.

$$\frac{\Gamma; \Delta_1 \vdash E :: \tau' \qquad \Gamma; \Delta_2 \vdash F :: \tau''}{\Gamma; \Delta \vdash E\ F :: \tau} \quad (\textsc{App})$$

where α new

$$\Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \to \alpha\})$$
$$\Delta = \Psi\Delta_1 \cup \Psi\Delta_2$$
$$\tau = \Psi\alpha$$

The combining operator $\Delta \cup \Delta'$ is defined to be a union of two monomorphic environments, provided they agree on their common monomorphic variables. Here, the two results Δ_1 and Δ_2 are combined using the unifier substitution Ψ, ensuring that $\Psi\Delta_1$ and $\Psi\Delta_2$ agree on their intersection.

5.5.4 Case expressions

The inference rule for **case** expressions is pretty straightforward. Patterns generate monomorphic type environments to allow for the necessary connection between patterns and cases. Note that variables defined in patterns don't leak out; this is necessary for the same reason we removed the argument variable from the typing of λ-abstractions.

$$\frac{\begin{array}{rl} & \Gamma;\Delta_0 \vdash E :: \tau_0 \\ \Delta'_1 \vdash P_1 :: \tau'_1 & \Gamma;\Delta_1 \vdash E_1 :: \tau_1 \\ & \vdots \\ \Delta'_n \vdash P_n :: \tau'_n & \Gamma;\Delta_n \vdash E_n :: \tau_n \end{array}}{\Gamma;\Delta \vdash \textbf{case}\ E\ \textbf{of}\ P_1 \mapsto E_1 \ldots P_n \mapsto E_n :: \tau} \quad (\textsc{Case})$$

$$\begin{aligned} \text{where}\quad & \alpha \text{ new} \\ & \Psi = \mathcal{U}(\{\Delta_0, \Delta_1, \Delta'_1, \ldots, \Delta_n, \Delta'_n\}, \{\tau_0 \sim \tau'_i, \tau_i \sim \alpha \mid i = 1 \ldots n\}) \\ & \Delta = \Psi\Delta_0 \cup \bigcup_{i=1}^{n} (\Psi\Delta_i \setminus \operatorname{dom} \Delta'_i) \\ & \tau = \Psi\alpha \end{aligned}$$

The inference rules for patterns is basically the same as for their counterparts in expressions. The differences arise only because pattern-bound variables are always monomorphic, and because constructor patterns have to be fully applied.

$$\frac{\alpha \text{ new}}{\{x :: \alpha\} \vdash x :: \alpha} \quad (\textsc{VarPat})$$

$$\frac{\begin{array}{c}(c :: \tau_1 \to \cdots \to \tau_n \to T\ \overline{\tau}) \in \Upsilon \\ \Delta_1 \vdash P_1 :: \tau_1' \quad \cdots \quad \Delta_n \vdash P_n :: \tau_n'\end{array}}{\Delta \vdash c\ P_1 \ldots P_n :: \Psi(T\ \overline{\tau})} \quad (\textsc{ConPat})$$

$$\begin{aligned}\text{where}\quad & \Psi = \mathcal{U}(\{\Delta_1, \ldots, \Delta_n\}, \{\tau_1 \sim \tau_1', \ldots, \tau_n \sim \tau_n'\}) \\ & \Delta = \Psi\Delta_1 \cup \cdots \cup \Psi\Delta_n\end{aligned}$$

5.5.5 Let bindings

The key step in the inference rule for **let** is the construction of the monomorphic environment $\Psi\Delta' \cup \Psi\Delta_0$. Each definition is typechecked separately, using the original polymorphic environment. As we've shown before, this ensures monomorphic recursion by collecting recurrence constraints in $\Delta_1 \ldots \Delta_n$. The substitution Ψ_0 unifies the view on both the recurrence's type, and other, external monomorphic variables (e.g. from an enclosing λ-abstraction).

$$\frac{\begin{array}{rcl}\Gamma; \Delta_1 & \vdash & f\ \overline{P}_1 = E_1 \\ & \vdots & \\ \Gamma; \Delta_n & \vdash & f\ \overline{P}_n = E_n \\ \Gamma'; \Delta' & \vdash & E :: \tau\end{array}}{\Gamma; \Delta \vdash \textbf{let} f\ \overline{P}_1 = E_1 \ \cdots\ f\ \overline{P}_n = E_n \ \textbf{in}\ E :: \Psi\tau} \quad (\textsc{Let})$$

$$\begin{aligned}\text{where}\quad & \alpha \text{ new} \\ & \Psi_0 = \mathcal{U}(\{\Delta_1, \ldots, \Delta_n\}, \{\Delta_i(f) \sim \alpha \mid i = 1 \ldots n\} \\ & \Delta_0 = \bigcup_{i=1}^{n} \Psi\Delta_i \setminus \{f\} \\ & \Gamma' = \Gamma; \{f \mapsto \Delta_0 \vdash \Psi_0\alpha\} \\ & \Psi = \mathcal{U}(\{\Delta_0, \Delta'\}) \\ & \Delta = \Psi\Delta' \cup \Psi\Delta_0\end{aligned}$$

The newly-introduced variable f is removed from the resulting environment, leading to let-polymorphism as we have seen previously.

The inference rule for individual definitions forces the inclusion of the currently-defined variable in the monomorphic environment, to communicate the type of the right-hand sides of the equations even when no recursion occurs.

$$\frac{\begin{array}{rcl} \Delta_1 & \vdash & P_1 :: \tau_1 \\ & \vdots & \\ \Delta_n & \vdash & P_n :: \tau_n \\ \Gamma, \Delta' & \vdash & E :: \tau_0 \end{array}}{\Gamma; \Delta \vdash f\ P_1 \cdots P_n = E} \quad (\textsc{Def})$$

$$\begin{aligned} \text{where} \quad \Delta_0 &= \{f :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_0\} \\ \Psi &= \mathcal{U}(\{\Delta_0, \Delta_1, \ldots, \Delta_n, \Delta'\}) \\ \Delta &= (\Psi\Delta_0 \cup \Psi\Delta') \setminus \bigcup_{i=1}^{n} \operatorname{dom} \Delta_i \end{aligned}$$

The absence of mutual recursion, and the restriction of one variable binding per **let** are exploited only to simplify the formal statement of the inference rule; the above definition is easily adaptable for languages like Haskell that directly permit mutual recursion.

5.6 Principal typings, *C* and *HM*

Previously, when discussing type judgements of the form $\vdash E :: \tau$, we saw that a given expression can inhabit many types. The same holds for typings as well: $\{f :: \textsc{bool} \rightarrow [\textsc{bool}] \rightarrow \textsc{char},\ x :: \textsc{bool},\ y :: [\textsc{bool}]\} \vdash \textsc{char}$ is a perfectly valid typing for $f\ x\ y$.

Principal typings are defined in such a way as to relate *C* to *HM*: given an expression E and a polymorphic environment Γ, we say $\Delta \vdash \tau$ is *principal* if

- It is a correct typing of E, that is, $\Gamma; \Delta \vdash E :: \tau$
- Every other, *HM*-correct typing is just a substitution away: if $(\Gamma')^{\forall} \cup \Delta' \vdash_{HM} E :: \tau'$, then there exists a substitution Ψ such that $\Delta' = \Psi\Delta$ and $\tau' = \Psi\tau$. Here, $(\Gamma)^{\forall}$ denotes the polymorphic *HM*-environment created from a *C* environment by adding appropriate $\forall$-qualifiers to every type.

The algorithm directly constructible from *C* computes principal typings. Because of the definition of principal typings, a corollary of this is that every expression typeable in the system *HM* is also typeable in *C* ([28] via [15]).

6. Type class polymorphism

In this chapter we extend *HM* with overloaded functions in the form of type classes. This extended type system HM^κ is a suitably close model of the type system of the Haskell 98 language. The next chapter presents the changes to the compositional type system *C* to support HM^κ in the same way vanilla *C* supports *HM*.

6.1 Non-parametric polymorphism

The polymorphic expressions we've seen in previous chapters had no way of depending on the actual types involved at a usage site. The polymorphic function *map* defined in section 2.1 on page 14, of type $\forall\alpha,\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$, acts in the same way for any choice of α and β.

However, sometimes it is desirable to reuse names without reusing definitions. For example, it is very convenient to have a function $(\equiv) :: \alpha \to \alpha \to \textsc{bool}$ that means different things depending on the actual types involved, so that we can use it as $x \equiv$ `'a'` or $map\ f\ y \equiv \textsc{nil}$, but define it differently for characters and lists.

Overloaded functions like $(\equiv)$ allow for generic definitions like the following:

$$
\begin{aligned}
elem := \textbf{let}\ & elem\ \ x\ \ \textsc{nil} && = \textsc{false} \\
& elem\ \ x\ \ (\textsc{cons}\ y\ ys) && = (x \equiv y) \lor (elem\ x\ ys) \\
\textbf{in}\ & elem.
\end{aligned}
$$

But what should be the type of *elem*? It cannot be $\forall\alpha.\alpha \to [\alpha] \to \textsc{bool}$, because equality might not be defined for all types. Picking a single type for which we know equality exists, e.g. typing *elem* as $\textsc{char} \to [\textsc{char}] \to \textsc{bool}$, means giving up the whole "genericity" point.

What's needed is a way to express the middle ground between a $\forall$-quantified polymorphic type and a concrete type; one that expresses the additional restriction that you can't choose any type for α, it has to have an associated definition of $(\equiv)$.

6.2 Type classes and instances

Instead of singular overloaded functions like $(\equiv)$ in the example above, many functional programming languages like *Clean* and *Haskell 98* lump overloaded variable declarations into so-called *type classes*[29]. A type τ is an *instance* of a type class if there are variables defined for that type corresponding to the declarations of the type class.

For example, in Haskell 98, the type class *Eq* is a specification of two overloaded functions for a given type α, with types $(\equiv) :: \alpha \to \alpha \to \textsc{bool}$ and $(\not\equiv) :: \alpha \to \alpha \to \textsc{bool}$. Note that the type variable α must always occur in overloaded declarations of a class defined in terms of α, since otherwise there would be nothing to dispatch on. On the other hand, the variables declared to be overloaded don't need to be functions, they can be constants as well.

In the specific example of *Eq*, defining one overloaded variable in terms of the other is a straightforward matter; for such cases, Haskell 98 allows default definitions of overloaded variables. In this thesis, we will simplify matters by not allowing default definitions. Since we are only interested in type checking, it doesn't matter where the actual definition of an overloaded function comes from.

For interpreters and compilers, [16] describes a program transformation from a language with overloaded variables to one without them.

6.2.1 Superclasses

A type class can be defined to have other superclasses. If κ' is a superclass of κ, that means every type that is an instance of κ also has to be an instance of κ'. We will use the notation $\kappa < \kappa'$ for the reflexive transitive closure of this relation.

6.2.2 Type class declarations

To be able to define type classes, we would need to extend the Λ^{let} language with new constructs. To simplify matters, just like we did with definitions of algebraic data types, we will assume all classes and instances to be known *a priori*. The datatype context Υ is extended to contain information about all classes and instances. In particular, the information stored in Υ about classes is:

- List of type classes: The statement $\Upsilon \vdash \kappa$ holds if κ is a type class.
- Superclasses: given two type classes κ and κ', the statement $\Upsilon \vdash \kappa < \kappa'$ holds if $\kappa' = \kappa$ or κ' is a (direct or indirect) superclass of κ.

The types of the actual overloaded variables are stored in Γ as polymorphic variables. If the variable v is declared to be an overloaded variable in class $\kappa\ \alpha$ having type $\forall\overline{\beta}.\theta \Rightarrow \tau$, then $(v\ ::\ \forall\alpha, \overline{\beta}.\{\kappa\ \alpha\} \cup \theta \Rightarrow \tau)$ is included in the initial Γ_0.

6.2.3 Instance definitions

Types are not directly defined to be instances of a type class; instead, only type constructors can be made instances. Given a type class κ and a type constructor T with arity n, and some type variables $\alpha_1, \ldots, \alpha_k$ with $k \leq n$, the partially applied type constructor $T\ \alpha_1 \ldots \alpha_k$ can be made an instance of κ by supplying the definitions of the overloads declared in κ.

Note that this effectively means at most one instance for a given pair (κ, T) because the type of the overloaded variables declared in κ uniquely determines the arity of the type constructors which can be instances of κ.

Since the definition of an instance may require the type arguments to be themselves instances of some other type classes, statements about instances are of the form $\Upsilon \vdash \{\kappa_1\ \alpha_{i_1}, \ldots, \kappa_m\ \alpha_{i_m}\} \Rightarrow \kappa\ (T\ \alpha_1, \ldots, \alpha_k)$.

6.3 Overloaded and polymorphic types

With type classes, the type of the previously defined variable *elem* should express that it can inhabit the type $\alpha \rightarrow [\alpha] \rightarrow \textsc{bool}$ if α is an instance of class *Eq*. This requires an extension of the type system *HM*; we will call this extended system HM^κ. Figure 6.1 shows the syntax of types in HM^κ, with the changes compared to HM highlighted.

Type variable:	α		
Type constructor:	T		
Simple type:	τ	$=$	α
		$\mid$	$T\ \overline{\tau}$
		$\mid$	$\tau \to \tau$
Type class:	κ		
Overloaded context:	ϕ	$=$	$\overline{\kappa\ \tau}$
Overloaded type:	ρ	$=$	$\phi \Rightarrow \tau$
Polymorphic context:	θ	$=$	$\overline{\kappa\ \alpha}$
Polymorphic type:	σ	$=$	$\forall \overline{\alpha}.\theta \Rightarrow \tau$

Figure 6.1: Syntax of types of HM^κ

Using this syntax, we will write the type of *elem* as $\forall \alpha.\{Eq\ \alpha\} \Rightarrow \alpha \to [\alpha] \to \textsc{Bool}$. This is an example of a *polymorphic type* in HM^κ. Much like polymorphic types in HM, it allows occurrences of *elem* to inhabit different type instantiations.

The result of such an instantiation is an *overloaded type* recording the required instance context. For example, by instantiating α to $[\beta]$, we get the overloaded type $\{Eq\ [\beta]\} \Rightarrow [\beta] \to [[\beta]] \to \textsc{Bool}$. The predicate $Eq\ [\beta]$ may or may or may not be a satisfiable, depending on whether [] is an instance of Eq. For example, it may be the case that there is a universal instance of Eq [] (i.e. $\Upsilon \vdash Eq\ [\beta]$), in which case the predicate can be resolved into the empty polymorphic context, as in $\{\} \Rightarrow [\beta] \to [[\beta]] \to \textsc{Bool}$.

A more realistic example is to define equality on lists using equality on elements, expressed by the instance proposition $\Upsilon \vdash Eq\ \beta \Rightarrow Eq\ [\beta]$. In that case, the predicate of the overloaded type $\{Eq\ [\beta]\} \Rightarrow [\beta] \to [[\beta]] \to \textsc{Bool}$ is resolved into $\{Eq\ \beta\} \Rightarrow [\beta] \to [[\beta]] \to \textsc{Bool}$.

The following figure shows the definition of the predicate resolution operator $\phi \twoheadrightarrow \theta$:

$$\frac{}{\kappa\ \alpha \twoheadrightarrow \{\kappa\ \alpha\}}$$

$$\frac{\begin{array}{c}\Upsilon \vdash \{\kappa_1\ \alpha_{i_1}, \ldots, \kappa_m\ \alpha_{i_m}\} \Rightarrow \kappa\ (T\ \alpha_i \ldots \alpha_n) \\ \kappa_1\ \tau_{i_1} \twoheadrightarrow \theta_1 \quad \cdots \quad \kappa_m\ \tau_{i_m} \twoheadrightarrow \theta_m\end{array}}{\kappa\ (T\ \tau_1 \ldots \tau_n) \twoheadrightarrow \theta_1 \cup \cdots \cup \theta_m}$$

Note that after the predicates in an overloaded context are resolved, the result is always a polymorphic context.

6.4 Inference rules for *HM*$^\kappa$

Compared to *HM*, the most fundamental change in the inference rules is the addition of a local instance environment Φ. It is a set of overloaded predicates like ϕ and allows type inference rules to be formulated on simple types.

There are two ways to move between simple types, overloaded types and polymorphic types: Inst instantiates polymorphic types in some local instance environment into simple types in an extended environment, and Over simplifies Φ by moving predicates from the environment to the overloaded type.

$$\frac{\Gamma, \Phi \vdash E :: \sigma \qquad \phi \Rightarrow \tau \in \text{inst}(\sigma)}{\Gamma, \Phi \cup \phi \vdash E :: \tau} \quad (\textsc{Inst})$$

$$\frac{\Gamma, \Phi \cup \phi \vdash E :: \tau}{\Gamma, \Phi \vdash E :: \phi \Rightarrow \tau} \quad (\textsc{Over})$$

6.4.1 Constructors and variables

The inference rules for constructors and variables are direct transliterations of the rules of *HM*. Note that occurrences of monomorphic (overloaded) variables require the local instance environment to contain the necessary predicates.

$$\frac{(c :: \theta \Rightarrow \tau) \in \Upsilon \qquad \overline{\alpha} = \text{vars}\ \tau}{\Gamma, \Phi \vdash c :: \forall \overline{\alpha}.\theta \Rightarrow \tau} \quad (\textsc{Con})$$

$$\frac{(x :: \phi \Rightarrow \tau) \in \Gamma}{\Gamma, \Phi \cup \phi \vdash x :: \tau} \quad (\textsc{MonoVar})$$

$$\frac{(x :: \sigma) \in \Gamma}{\Gamma, \Phi \vdash x :: \sigma} \quad (\textsc{PolyVar})$$

6.4.2 Application, λ-abstraction and pattern matching

These too are directly derived from the respective rules of *HM*. Patterns have overloaded types as opposed to simple types because the type signature of constructors can contain predicates.

$$\frac{\Gamma, \Phi \vdash E :: \tau' \to \tau \qquad \Gamma, \Phi \vdash F :: \tau'}{\Gamma, \Phi \vdash E\ F :: \tau} \quad (\textsc{App})$$

$$\frac{(\Gamma; (x :: \tau')), \Phi \vdash E :: \tau \qquad \Upsilon \vdash \tau'}{\Gamma, \Phi \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

$$\frac{\begin{array}{lllr} & & \Gamma, \Phi \vdash & E :: \tau_0 \\ P_1 :: \phi_1 \Rightarrow \tau_0 \dashv \Gamma_1 & \Upsilon \vdash \phi_1 \sqsubseteq \Phi & \Gamma \cup \Gamma_1, \Phi \vdash & E_1 :: \tau \\ & & \vdots & \\ P_n :: \phi_n \Rightarrow \tau_0 \dashv \Gamma_n & \Upsilon \vdash \phi_n \sqsubseteq \Phi & \Gamma \cup \Gamma_n, \Phi \vdash & E_n :: \tau \end{array}}{\Gamma, \Phi \vdash \textbf{case}\ E\ \textbf{of}\ P_1 \mapsto E_1 \ldots P_n \mapsto E_n :: \tau} \quad (\textsc{Case})$$

$$\frac{\Upsilon \vdash \tau}{x :: \phi \Rightarrow \tau \dashv (x :: \tau)} \quad (\textsc{VarPat})$$

$$\frac{\begin{array}{l} (c :: \sigma) \in \Upsilon \\ (\phi \Rightarrow \tau_1 \to \cdots \to \tau_n \to T\ \overline{\tau}) \in \mathrm{inst}(\sigma) \\ \Upsilon \vdash \phi \sqsubseteq \Phi \\ P_1 :: \phi_1 \Rightarrow \tau_1 \dashv \Gamma_1 \quad \cdots \quad P_n :: \phi_n \Rightarrow \tau_n \dashv \Gamma_n \end{array}}{c\ P_1 \ldots P_n :: \phi \cup \phi_1 \cup \cdots \cup \phi_n \Rightarrow T\ \overline{\tau} \dashv \Gamma_1 \cup \ldots \cup \Gamma_n} \quad (\textsc{ConPat})$$

Here, the judgement $\Upsilon \vdash \phi \sqsubseteq \phi'$ on overloaded contexts is defined to be true if for every predicate $\kappa\ \tau$ in ϕ, there is a subclass $\kappa' < \kappa$ such that $\kappa'\ \tau$ is in ϕ'.

6.4.3 Let bindings

In *HM*, let-bound variables are monomorphic in recurrences, and their type is generalised for the body of the **let**. Similarly, in HM^{κ} recurrences are typed with overloaded types, and the generalisation results in a polymorphic type.

$$\frac{\begin{array}{c} P_1 :: \phi_1 \Rightarrow \tau_1 \dashv \Gamma_1 \qquad \Upsilon \vdash \phi_1 \sqsubseteq \phi \\ \vdots \\ P_n :: \phi_n \Rightarrow \tau_n \dashv \Gamma_n \qquad \Upsilon \vdash \phi_n \sqsubseteq \phi \\ \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n, \Phi \vdash E :: \phi \Rightarrow \tau \end{array}}{\Gamma, \Phi \vdash f\ P_1 \ldots P_n = E \dashv (f :: \phi \Rightarrow \tau_1 \to \ldots \to \tau_n \to \tau)} \quad (\textsc{Def})$$

$$\frac{\begin{array}{l} \Gamma' = \Gamma_1; \ldots; \Gamma_n \\ \Gamma \cup \Gamma', \Phi \vdash D_1 \dashv \Gamma_1 \quad \cdots \quad \Gamma \cup \Gamma', \Phi \vdash D_n \dashv \Gamma_n \\ \Gamma'' = \{x :: \sigma \mid (x :: \rho) \in \Gamma', \sigma = \mathrm{gen}(\Gamma, \rho)\} \\ \Gamma \cup \Gamma'', \Phi \vdash E :: \tau \end{array}}{\Gamma, \Phi \vdash \textbf{let}\ D_1 \ldots D_n\ \textbf{in}\ E :: \tau} \quad (\textsc{Let})$$

An important aspect of the type generalisation $\sigma = \mathrm{gen}(\Gamma, \rho)$ is checking for ambiguous predicates: for every polymorphic predicate $\kappa\ \beta$ of the polymorphic type $\forall \overline{\alpha}.\theta \Rightarrow \tau$, the type variable β must occur in τ.

To understand this requirement, consider the following expression:

$$\begin{aligned} \Upsilon &:= \{\, [\alpha] = \ldots, (\alpha, \beta) = \ldots, (\alpha, \beta, \gamma) = \ldots, \textsc{char} = \ldots, \\ &\qquad \mathit{Textual}\} \\ \Gamma_0 &:= \{\mathit{show} :: \forall \alpha.\mathit{Textual}\ \alpha \Rightarrow \alpha \to [\textsc{char}], \mathit{read} :: \forall \alpha.\mathit{Textual}\ \alpha \Rightarrow [\textsc{char}] \to \alpha\} \end{aligned}$$

$$\begin{array}{l} \textbf{let}\ \ \mathit{test}_1\ \ x = \mathit{read}\ (\mathit{show}\ x) \\ \qquad \mathit{test}_2\ \ x = \mathit{show}\ (\mathit{read}\ x) \\ \textbf{in}\ (\mathit{test}_1, \mathit{test}_2) \end{array}$$

Here, the type of the two let-bound variables, after generalisation, are:

$$\begin{array}{l} \mathit{test}_1 :: \forall \alpha, \beta.\mathit{Textual}\ \alpha, \mathit{Textual}\ \beta \Rightarrow \alpha \to \beta \\ \mathit{test}_2 :: \forall \alpha.\mathit{Textual}\ \alpha \Rightarrow [\textsc{char}] \to [\textsc{char}] \end{array}$$

The inferred polymorphic type of test_2 fails ambiguity checking, because there is an instance predicate for the type variable α, but α does not occur in the type $[\textsc{char}] \to [\textsc{char}]$. If we were to allow such polymorphic types, then there would be nothing to dispatch on whenever $test_2$ is invoked — in other words, no way to resolve the calls to the overloaded functions *read* and *show*.

6.5 Algorithms for HM^{κ}

HM^{κ} is very much like *HM* in that type inference seemingly requires prescience in the application of rules Inst and Abs. Also, any number of predicates can be moved from the local instance environment into the inferred overloaded type in Over, but only one choice leads to an unambiguous generalised polymorphic type when applying rule Let.

Since instance predicates can only arise from the usage of overloaded variables, the bottom-up algorithm $\mathcal{W}$ can be easily adopted for direct type inference of Λ^{let} in HM^{κ}. Other approaches such as [29] present translations of expressions into languages where implementations of the basic *HM* is sufficient to do type inference. The linearity argument presented in chapter 4 applies to both methods.

7. Compositional type checking for type class polymorphism

In chapter 4, we have demonstrated a limitation of linear type systems when it comes to explaining type errors. Compositional type systems, like the one described in chapter 5, can give better explanations because it becomes meaningful to argue about the type of subexpressions.

Better explanation of type errors is useful for the working programmer — if it applies to a programming language with real-world use. The language Λ^{let} with the type system HM^{κ} presented in chapter 6 is a good enough model of the functional programming language Haskell 98 so that practical results applying to Haskell can be derived from its analysis.

Therefore, in this chapter we extend the type system C to allow for type class polymorphism, resulting in the type system C^{κ} that we've used to implement a compositional type checker for Haskell 98 (see the appendix for details on the implementation).

7.1 Class predicates in typings

The type system C presented in chapter 5 is based on judgements of the scheme $\Gamma, \Delta \vdash E : \tau$, where Γ is the polymorphic environment passed down, and Δ is the monomorphic environment collected from subexpressions.

There are two apparent ways to include class predicates in the typing $\Delta \vdash \tau$. One is to use either overloaded or polymorphic types instead of a simple type, and the other is to include predicates in Δ.

The problem with using an overloaded or a polymorphic type on the right-hand side of a typing (i.e. by allowing typings of the scheme $\Delta \vdash \rho$ or $\Delta \vdash \sigma$) is that class predicate constraints should propagate independently of the actual usage of overloaded variables.

For example, consider the following expression:

$$\Upsilon := \{Textual\}$$
$$\Gamma_0 := \{show\ ::\ \forall\alpha.Textual\ \alpha \Rightarrow \alpha \to [\textsc{char}]\}$$

$$idText :=\ \ \lambda x \mapsto \mathbf{let}\ s = show\ x\ \mathbf{in}\ x.$$

Using the inference rules of HM^κ, we can infer the principal type $\forall\alpha.Textual\ \alpha \Rightarrow \alpha \to \alpha$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash show\ ::\ \forall\alpha.Textual\ \alpha \Rightarrow \alpha \to [\textsc{char}]}{\Gamma' \vdash show\ ::\ Textual\ \alpha \Rightarrow \alpha \to [\textsc{char}]}}{\Gamma', \Phi \vdash show\ ::\ \alpha \to [\textsc{char}]} \qquad \Gamma', \Phi \vdash x\ ::\ \alpha}{\Gamma', \Phi \vdash show\ x\ ::\ [\textsc{char}]}}{\Gamma', \Phi \vdash s = show\ x \dashv \{s\ ::\ [\textsc{char}]\}} \qquad \Gamma', \Phi \vdash x\ ::\ \alpha}{\Gamma, \Phi \vdash \mathbf{let}\ s = show\ x\ \mathbf{in}\ x\ ::\ \alpha}}{\Gamma_0, \Phi \vdash \lambda x \mapsto \mathbf{let}\ s = show\ x\ \mathbf{in}\ x\ ::\ \alpha \to \alpha}}{\Gamma_0 \vdash \lambda x \mapsto \mathbf{let}\ s = show\ x\ \mathbf{in}\ x\ ::\ Textual\ \alpha \Rightarrow \alpha \to \alpha}$$

$$\begin{array}{lll} \text{with} & \Gamma & = \Gamma_0; \{x\ ::\ \alpha\} \\ & \Gamma' & = \Gamma'; \{s\ ::\ [\textsc{char}]\} \\ & \Phi & = \{Textual\ \alpha\} \end{array}$$

As we can see, the definition of s is enough to constraint the type variable α, without s actually occurring anywhere.

Because of this need to make instance requirements introduced by variable definitions independent of variable usage, we record class predicate constraints generated by subexpressions and pass them around just like the monomorphic environment Δ. Type instantiation then affects the whole of Δ and this instance environment Θ in unison.

Overloaded contexts only appear in intermediate steps of substitution: if $\alpha \rightsquigarrow \tau$ is applied to $\kappa\ \alpha$, the resulting overloaded predicate $\kappa\ \tau$ is immediately resolved (using the same resolution operator $\phi \twoheadrightarrow \theta$ as used by HM^κ), and the resulting polymorphic predicates are simply combined (with redundant superclasses removed) into the substituted instance environment Θ'. We will denote this substitution-resolution-combination with the $+$ operator.

7.2 Inference rules

The inference rules of *C* can be adapted to handle class predicates in a straightforward manner. We omit lengthy explanations and let the rules speak for themselves.

7.2.1 Constants and variables

$$\frac{(c :: \theta \Rightarrow \tau) \in \Upsilon \qquad \emptyset; \Theta \vdash \tau' = \text{inst}\, \emptyset; \theta \vdash \tau}{\Gamma; \emptyset; \Theta \vdash c :: \tau'} \quad (\textsc{Con})$$

$$\frac{\Gamma(x) = \Delta; \Theta \vdash \tau \qquad \Delta'; \Theta' \vdash \tau' = \text{inst}\, \Delta; \Theta \vdash \tau}{\Gamma; \Delta'; \Theta' \vdash x :: \tau'} \quad (\textsc{PolyVar})$$

$$\frac{x \notin \text{dom}\, \Gamma \qquad \alpha \text{ new}}{\Gamma; \{x :: \alpha\}; \emptyset \vdash x :: \alpha} \quad (\textsc{MonoVar})$$

7.2.2 λ-abstraction and function application

$$\frac{\Gamma; \Delta; \Theta \vdash E :: \tau \qquad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x; \Theta \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

$$\frac{\Gamma; \Delta; \Theta \vdash E :: \tau \qquad x \notin \text{dom}\, \Delta \qquad \alpha \text{ new}}{\Gamma; \Delta; \Theta \vdash \lambda x \mapsto E :: \alpha \to \tau} \quad (\textsc{Abs}')$$

$$\frac{\Gamma; \Delta_1; \Theta_1 \vdash E :: \tau' \qquad \Gamma; \Delta_2; \Theta_2 \vdash F :: \tau''}{\Gamma; \Delta; \Theta \vdash E\ F :: \tau} \quad (\textsc{App})$$

where α new

$$\Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \to \alpha\})$$
$$\Delta = \Psi\Delta_1 \cup \Psi\Delta_2$$
$$\Theta = \Psi\Theta_1 + \Psi\Theta_2$$
$$\tau = \Psi\alpha$$

7.2.3 Case expressions

$$\frac{\begin{array}{rl} & \Gamma; \Delta_0; \Theta_0 \vdash \ E :: \tau_0 \\ \Delta'_1; \Theta'_1 \vdash P_1 :: \tau'_1 & \Gamma; \Delta_1; \Theta_1 \vdash \ E_1 :: \tau_1 \\ & \vdots \\ \Delta'_n; \Theta'_1 \vdash P_n :: \tau'_n & \Gamma; \Delta_n; \Theta_1 \vdash \ E_n :: \tau_n \end{array}}{\Gamma; \Delta; \Theta \vdash \textbf{case } E \textbf{ of } P_1 \mapsto E_1 \ldots P_n \mapsto E_n :: \tau} \quad (\textsc{Case})$$

where α new

$$\begin{aligned} \Psi &= \mathcal{U}(\{\Delta_0, \Delta_1, \Delta'_1, \ldots, \Delta_n, \Delta'_n\}, \{\tau_0 \sim \tau'_i, \tau_i \sim \alpha \mid i = 1 \ldots n\}) \\ \Delta &= \Psi\Delta_0 \cup \bigcup_{i=1}^{n} (\Psi\Delta_i \setminus \operatorname{dom} \Delta'_i) \\ \tau &= \Psi\alpha \\ \Theta &= \sum_{i=1}^{n} (\Psi\Theta_i + \Psi\Theta'_i) + \Psi\Theta_0 \end{aligned}$$

$$\frac{\alpha \text{ new}}{\{x :: \alpha\}; \emptyset \vdash x :: \alpha} \quad (\textsc{VarPat})$$

$$\frac{\begin{array}{c} (c :: \theta \Rightarrow \tau_1 \to \cdots \to \tau_n \to T\ \overline{\tau}) \in \Upsilon \\ \Delta_1; \Theta_1 \vdash P_1 :: \tau'_1 \quad \cdots \quad \Delta_n; \Theta_n \vdash P_n :: \tau'_n \end{array}}{\Delta; \Theta \vdash c\ P_1 \ldots P_n :: \Psi(T\ \overline{\tau})} \quad (\textsc{ConPat})$$

where
$$\begin{aligned} \Psi &= \mathcal{U}(\{\Delta_1, \ldots, \Delta_n\}, \{\tau_1 \sim \tau'_1, \ldots, \tau_n \sim \tau'_n\}) \\ \Delta &= \Psi\Delta_1 \cup \cdots \cup \Psi\Delta_n \\ \Theta &= \Psi\theta + \Psi\Theta_1 + \cdots + \Psi\Theta_n \end{aligned}$$

Note that although pattern-bound variables are removed from the monomorphic environment Δ in rule Case (which is exactly the same behaviour as C), the class predicates generated from patterns are collected as-is from the predicate environments Θ'_i.

7.2.4 Let bindings

$$\frac{\begin{array}{rcl} \Gamma; \Delta_1; \Theta_1 & \vdash & f\ \overline{P}_1 = E_1 \\ & \vdots & \\ \Gamma; \Delta_n; \Theta_n & \vdash & f\ \overline{P}_n = E_n \\ \Gamma'; \Delta'; \Theta' & \vdash & E :: \tau \\ \multicolumn{3}{l}{\textsc{Unamb}(\Theta_0, \Delta_0 \vdash \tau_0)} \end{array}}{\Gamma; \Delta; \Theta \vdash \mathbf{let}\, f\ \overline{P}_1 = E_1\ \cdots\ f\ \overline{P}_n = E_n\ \mathbf{in}\ E :: \Psi\tau} \quad (\textsc{Let})$$

where α new

$$\Psi_0 = \mathcal{U}(\{\Delta_1, \ldots, \Delta_n\}, \{\Delta_i(f) \sim \alpha \mid i = 1 \ldots n\}$$

$$\Delta_0 = \bigcup_{i=1}^{n} \Psi_0 \Delta_i \setminus \{f\}$$

$$\tau_0 = \Psi_0 \alpha$$

$$\Theta_0 = \sum_{i=1}^{n} \Psi_0 \Theta_i$$

$$\Gamma' = \Gamma; \{f \mapsto \Delta_0 \vdash \tau_0\}$$

$$\Psi = \mathcal{U}(\{\Delta_0, \Delta'\})$$

$$\Delta = \Psi\Delta' \cup \Psi\Delta_0$$

$$\Theta = \Psi\Theta_0 + \Psi\Theta'$$

$$\frac{\begin{array}{rcl} \Delta_1; \Theta_1 & \vdash & P_1 :: \tau_1 \\ & \vdots & \\ \Delta_n; \Theta_n & \vdash & P_n :: \tau_n \\ \Gamma, \Delta'; \Theta' & \vdash & E :: \tau_0 \end{array}}{\Gamma; \Delta; \Theta \vdash f\ P_1\ \cdots P_n = E} \quad (\textsc{Def})$$

where $\Delta_0 = \{f :: \tau_1 \to \ldots \to \tau_n \to \tau_0\}$

$$\Psi = \mathcal{U}(\{\Delta_0, \Delta_1, \ldots, \Delta_n, \Delta'\})$$

$$\Delta = (\Psi\Delta_0 \cup \Psi\Delta') \setminus \bigcup_{i=1}^{n} \operatorname{dom} \Delta_i$$

$$\Theta = \sum_{i=1}^{n} \Psi\Theta_i + \Psi\Theta_0 + \Psi\Theta'$$

7.3 Checking for ambiguous class predicates

An important point of HM^κ is that the class predicates in a polymorphic type can only refer to type variables that occur in the type itself; for example, the type *Textual* $\alpha \Rightarrow$ [CHAR] is not a valid polymorphic type because α doesn't occur in [CHAR]. As we have seen, this rule is enforced by the application of gen in rule LET.

However, there is no explicit type generalisation step in *C*, and class predicates are not associated with types. But there is a point when the polymorphic environment Γ is extended into Γ' by adding the typing of the newly-defined variable,

$$\begin{aligned}\Theta_0 &= \sum_{i=1}^{n} \Psi_0 \Theta_i \\ \Gamma' &= \Gamma; \{f \mapsto \Delta_0 \vdash \Psi_o \alpha\},\end{aligned}$$

where we can check that *usages* of *f* will yield type equations that necessarily fix all type variables occurring in class predicates by requiring the following property UNAMB:

$$\textsc{Unamb}(\Theta, \Delta \vdash \tau) := \forall (\kappa\ \alpha) \in \Theta : \alpha \in \mathrm{vars}\ \tau \vee \alpha \in \bigcup \{\mathrm{vars}\ \tau' \mid (x :: \tau') \in \Delta\}\,,$$

which can be seen in the inference rule LET.

Conclusions

We have presented two type systems for the language Λ^{let} , a practical model of real-life functional languages like Haskell or Clean. One is the well-known Hindley-Milner type system, the canonical type system for let-polymorphic, λ-calculus-based languages; the other is a compositional type system based on typings that allows for better error reporting precisely by being compositional.

By extending both type systems with type class polymorphism, a form of *ad-hoc* polymorphism, we arrive at the feature set of the Haskell 98 programming language. An implementation of C^{κ} can leverage all the advantages of compositional type inference presented in chapter 4, to report type errors in real-world Haskell 98 programs. Appendix A contains notes on our reference implementation.

Future work

The Glasgow Haskell Compiler (GHC) introduces many type system extensions to Haskell 98, some of which directly concern type class polymorphism[30]:

- Multi-parameter type classes
- Functional dependencies[31]
- Flexible contexts & undecidable instances[32]
- Flexible & overlapping instances

A natural way to follow up on the present work would be adding support for these type system features to the compositional type system C^{κ}.

On the practical side, our implementation takes some shortcuts and doesn't support some features of Haskell 98, most notably the module system, record types and the do-notation. A future revision of the implementation should fix these omissions to enable real-world usage.

A. Implementation notes

As an illustration of the type system C^κ, we have created an implementation for the Haskell 98 programming language. This implementation, named Tandoori, is available under the terms of the BSD license from `http://gergo.erdi.hu/projects/tandoori/`.

Using GHC as a basis

The modular design of the Glasgow Haskell Compiler (GHC) enabled us to reuse the implementation of the "boring parts" of writing a type checker for Haskell. Tandoori is inserted into the compiler pipeline replacing the regular, Hindley-Milner-based type checker. This way, we can use the source code parser and the reference resolver from GHC. The latter also sorts definitions via dependency analysis, so its output is already more-or-less in the simplified format that we used when discussing Λ^{let} .

When we started work on Tandoori, the latest stable release of GHC was version 6.10. Adopting the codebase to GHC 6.12 only involved changing the topmost calls to the parser and the resolver since the output format of these steps was unchanged.

High-level design

Tandoori uses an Error/Reader/Writer/State monad[33] to thread state and results throughout the type inference:

- The Error and the Writer part are both used for collecting type errors. Wherever there is a sensible, most generic default to use as the result of type inference, errors detected in that part are tunnelled into the Writer and type inference goes on with the default results. This allows us to detect multiple errors in one go.

- The Reader part contains information on data types, classes and instances (the Υ part of C^κ), and the polymorphic environment for the current scope (Γ). We also store the source code location of the currently-analysed subexpression; this information is used in error messages.
- The State part is a simple infinite supply of type variables, used wherever the inference rules call for a new unique type variable.

The actual sequence of events in the type checker is as follows, with items marked $*$ implemented by GHC:

1. $*$ Parse source code
2. $*$ Resolve references, sort and group definitions using dependency analysis
3. Collect data type definitions
4. Collect class declarations, build superclass graph, calculate the $<$ relation on classes
5. Collect instance declarations
6. Type check variable definitions, record polymorphic types group-by-group in the order given by step 2
7. Type check instance definitions

Note that type class instances are processed twice. The first run merely records the existence of instances, and this information is used when type checking variable definitions. Instance definitions are only checked after the type of all defined variables have been inferred, since instance definitions can contain references to arbitrary top-level variables.

Known limitations

We have taken some shortcuts and omitted some features in Tandoori that would be straightforward but laborious to implement while giving no new insight. Among these are:

- Module inclusion, implicitly including the Haskell Prelude

- Record data types
- Do-notation
- Guards

Hopefully, the community will take up Tandoori and a future revision will address these points, leading to a compositional type checker that can enjoy real-world usage.

Bibliography

[1] J. Abrial, M. Lee, D. Neilson, P. Scharbach, and I. Sørensen, "The B-method," in *VDM '91 Formal Software Development Methods* (S. Prehn and H. Toetenel, eds.), vol. 552 of *Lecture Notes in Computer Science*, pp. 398–405, Springer Berlin / Heidelberg, 1991. 10.1007/BFb0020001.

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[3] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[4] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, Dec. 1978.

[5] W. Howard, "The formulae-as-types notion of construction, To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism (JR Hindley and JP Seldin, eds.)," 1980.

[6] W. Quine, *Word and object*. The MIT Press, 1960.

[7] B. Nordström, K. Petersson, and J. Smith, *Programming in Martin-Löf's type theory*. Citeseer, 1990.

[8] U. Norell, *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, Sept. 2007.

[9] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc, 2004.

[10] C. McBride, "Epigram: Practical programming with dependent types," *Advanced Functional Programming*, pp. 130–170, 2005.

[11] P. Hudak, P. Wadler, A. Brian, B. J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, S. P. Jones, M. Reeve, D. Wise, and J. Young, "Report on the programming language Haskell: A non-strict, purely functional language," *ACM SIGPLAN Notices*, vol. 27, 1992.

[12] T. Brus, M. van Eekelen, M. van Leer, and M. Plasmeijer, "Clean – a language for functional graph rewriting," in *Functional Programming Languages and Computer Architecture* (G. Kahn, ed.), vol. 274 of *Lecture Notes in Computer Science*, pp. 364–384, Springer Berlin / Heidelberg, 1987.

[13] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning Haskell," in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pp. 62–71, ACM, 2003.

[14] B. Heeren *et al.*, "Top Quality Type Error Messages," 2005.

[15] O. Chitil, "Compositional explanation of types and algorithmic debugging of type errors," in *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, (New York, NY, USA), pp. 193–204, ACM, 2001.

[16] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler, "Type classes in Haskell," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 2, pp. 109–138, 1996.

[17] "The Glasgow Haskell Compiler." `http://www.haskell.org/ghc`, Nov. 2010.

[18] R. Rojas, "A tutorial introduction to the lambda calculus," *FU Berlin*, 1997.

[19] A. Church, "A set of postulates for the foundation of logic," *Annals of mathematics*, vol. 33, no. 2, pp. 346–366, 1932.

[20] A. Turing, "Computability and λ-definability," *Journal of Symbolic Logic*, vol. 2, no. 4, pp. 153–163, 1937.

[21] A. Church, "An unsolvable problem of elementary number theory," *American journal of mathematics*, vol. 58, no. 2, pp. 345–363, 1936.

[22] L. Damas and R. Milner, "Principal type-schemes for functional languages," in *Proc. 9th ACM Symp. on Principles of Programming Languages*, pp. 207–212, 1982.

[23] L. M. M. Damas, *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, Apr. 1985. Technical report CST-33-85.

[24] O. Lee and K. Yi, "Proofs about a folklore let-polymorphic type inference algorithm," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 4, p. 723, 1998.

[25] J. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965.

[26] "Hugs 98." http://www.haskell.org/hugs, Nov. 2010.

[27] C. Camarao and L. Figueiredo, "ML Has Principal Typings," in *4th Brazilian Symposium on Programming Languages, Recife, Brazil*, Citeseer, 2000.

[28] J. C. Mitchell, *Foundations of programming languages*. Cambridge, MA, USA: MIT Press, 1996.

[29] 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*, pp. 60–76, ACM, 1989.

[30] S. Jones, M. Jones, and E. Meijer, "Type classes: exploring the design space," in *Haskell workshop*, vol. 1997, 1997.

[31] M. Jones, "Type classes with functional dependencies," *Programming Languages and Systems*, pp. 230–244, 2000.

[32] M. Sulzmann, G. Duck, S. Peyton-Jones, and P. Stuckey, "Understanding functional dependencies via constraint handling rules," *Journal of Functional Programming*, vol. 17, no. 01, pp. 83–129, 2007.

[33] M. Jones, "Functional programming with overloading and higher-order polymorphism," *Advanced Functional Programming*, pp. 97–136, 1995.