Type Checking Type Classes

Tobias Nipkow* and Christian Prehofer**
TU München††

Abstract

We study the type inference problem for a system with type classes as in the functional programming language Haskell. Type classes are an extension of ML-style polymorphism with overloading. We generalize Milner's work on polymorphism by introducing a separate context constraining the type variables in a typing judgement. This leads to simple type inference systems and algorithms which closely resemble those for ML. In particular we present a new unification algorithm which is an extension of syntactic unification with constraint solving. The existence of principal types follows from an analysis of this unification algorithm.

1 Introduction

The extension of Hindley/Damas/Milner polymorphism with the notion of *type classes* in the functional programming language Haskell [6] has attracted much attention. Type classes permit the systematic overloading of function names while retaining the advantages of the Hindley/Damas/Milner system: every expression which has a type has a most general type which can be inferred automatically. Although many extensions to Haskell's type system have already been proposed (and also implemented), we believe that the *essence* of Haskell's type inference algorithm has still not been presented in all its simplicity. The main purpose of this paper is to give what we believe to be the simplest algorithm published so far, a contribution for implementors. At the same time we present a correspondingly simple type inference system, a contribution aimed at users of the language. The algorithm is sound and complete with respect to the inference system, and both are very close to their ML-counterparts. Despite this proximity, the proofs are considerably more involved and only the main steps are shown.

A type class in Haskell is essentially a set of types (which all happen to provide a certain set of functions). The classical example is equality. In old versions of ML, the equality function = has the polymorphic type $\forall\alpha.\alpha \to \alpha \to bool$, where the type variable α ranges over all types. However, = should not be applied to arguments of function type. To fix this problem, Standard ML [11] introduces special type variables that range only over types where equality is defined. Equality differs from other polymorphic functions not just because of its restricted domain but also because of its mixture of polymorphism and overloading: equality on lists is implemented differently from equality on integers.

Type classes treat both issues in a systematic way: the type variable α is restricted to elements of a certain type class, say Eq. Then for each type τ where = should be defined, we have to declare that τ is of class Eq by providing an implementation of = of type $\tau \to \tau \to bool$.

To express the fact that a type τ is in some class C we introduce the judgement $\tau : C$.[1] The idea of viewing Haskell as a three level system of expressions, types and classes, where classes classify types, goes back to Nipkow and Snelting [12]. However, in their system it is impossible to express that a type belongs to more than one class. To overcome this difficulty we introduce *sorts*

*Research supported in part by ESPRIT BRA 3245, *Logical Frameworks*.

**Research supported by the DFG under grant Br 887/4-1, *Deduktive Programmentwicklung*.

††Address: Institut für Informatik, Technische Universität München, Postfach 20 24 20, W-8000 München 2, Germany. Email: {nipkow,prehofer}@informatik.tu-muenchen.de

[0] Appeared in Proceedings of POPL'93

[1] If classes are viewed as predicates on types, this leads to the Haskell notation $C(\tau)$.

as finite sets of classes. The judgement $\tau : \{C_1, \ldots, C_n\}$ is a compact form of the conjunction $\tau : C_1 \wedge \ldots \wedge \tau : C_n$. Alternatively we may think of $\{C_1, \ldots, C_n\}$ as a notation for $C_1 \cap \ldots \cap C_n$, the intersection of the types belonging to the classes C_1 to C_n. This leads to a simple type inference system and algorithm. The former resembles that for Mini-ML [4], the latter is very similar to algorithm $\mathcal{I}$ by Milner [10]. The main difference is that in both cases we also compute a set of constraints of the form $\alpha : \{C_1, \ldots, C_n\}$ where α is a type variable.

2 Mini-Haskell

Since the aim of this paper is simplicity, we treat only the most essential features of Haskell relating to type classes. The resulting language is basically Mini-ML [4] plus *class* and *instance* declarations, *Mini-Haskell* for short. Its syntax is shown in Figure 1. Although the next paragraph provides a brief account of type classes, the reader should consult the Haskell Report [6] or the original paper on type classes [14] for motivations and examples.

Mini-Haskell extends ML by a restricted form of overloading. Each class declaration introduces a new class name C and a new overloaded function name x. Semantically C represents the set of all types which support a function x. For instance

$$\textbf{class } Eq \textbf{ where } eq : \forall\alpha{:}Eq.\alpha \to \alpha \to bool$$

introduces the class Eq of all those types τ which provide a function $eq : \tau \to \tau \to bool$. A class declaration is like a module interface in that it separates declarations from implementations. In order to "prove" that a particular type, say int, is in Eq, a "witness" for the required function eq needs to be provided. This is the purpose of instance declarations. In order to prove $int : Eq$ we instantiate eq by eq_int, some existing function of type $int \to int \to bool$:

$$\textbf{inst } int : Eq \textbf{ where } eq = eq\_int$$

As motivated in the introduction, *sorts* are finite sets of classes. This representation is a key ingredient for the concise treatment of type inference. Yet semantically the sort $\{C_1, \ldots, C_n\}$ should be understood as $C_1 \cap \ldots \cap C_n$. Thus $\{C\}$ and C are equivalent, and the empty set $\{\}$ is the sort/set of all types. If S_1 is more specialized, i.e. represents fewer types, than S_2, we write $S_1 \leq S_2$. Since sorts are interpreted as intersections, $S_1 \leq S_2 \Leftrightarrow S_1 \supseteq S_2$ holds. Hence any two sorts S_1 and S_2 have a greatest lower bound whose representation is their union $S_1 \cup S_2$.

Types in Mini-Haskell are simply terms over variables and constructors of fixed arity. Note that $\to$ is just another type constructor, i.e. $\tau_1 \to \tau_2$ is short for $\to(\tau_1, \tau_2)$. The set of free variables in a type scheme is denoted by $\mathcal{FV}(\sigma)$. Bound variables in type schemes range only over certain subsets of types: $\forall\alpha{:}S.\sigma$ abbreviates all instances $[\tau/\alpha]\sigma$ where $\tau : S$, a judgment defined formally below.

Expressions are λ-terms extended with **let**-definitions. A program is a sequence of declarations followed by an expression.

A Mini-Haskell class declaration **class** C **where** $x : \forall\alpha{:}C.\sigma$ (where σ should contain no free variables except α) corresponds to the Haskell declaration **class** $C\,\alpha$ **where** $x : \sigma$. The translation in the opposite direction is more involved because of the following simplifying assumptions:

1. A class declares only one function symbol, whereas Haskell allows a set of functions. This feature is clearly not essential. Strictly speaking, we could have dropped class names altogether since there is a one to one correspondence between class names and the single function declared in that class. However this would have obscured the connection with Haskell.

2. Classes are not ordered. It has already been observed [3] that subclasses are mere syntactic sugar. Section 6 discusses ways of dealing with subclasses.

A Mini-Haskell instance declaration **inst** $t : (S_1, \ldots, S_n)C$ **where** $x = e$ expresses that $t(\tau_1, \ldots \tau_n)$ is in class C provided the τ_i are of sort S_i. It corresponds to the Haskell declaration **inst** $(con) \Rightarrow C(t\,\alpha_1 \ldots \alpha_n)$ **where** $x = e$ where con is a list consisting of assumptions $C'\,\alpha_i$ with $C' \in S_i$ for all $i = 1 \ldots n$.

In the sequel a list of syntactic objects $s_1, \ldots, s_n$ is abbreviated by $\overline{s_n}$. For instance, $\forall\overline{\alpha_n{:}S_n}.\sigma$ is equivalent with $\forall\alpha_1{:}S_1, \ldots, \alpha_n{:}S_n.\sigma$.

2.1 Classifying Types

Before we embark on type inference, the simpler problem of sort inference has to be settled. In ML and many other languages we have the judgement $e : \tau$, expressing that e is of type τ. Similarly, we classify types by sorts with the judgement $\tau : C$, stating that type τ is in class C. This judgement requires two kinds of information, namely the sorts of the type variables in τ and the "functionality" of the type constructors. The former is recorded in a *sort context* Γ, which is a total mapping from type variables to sorts such that $\mathcal{D}om(\Gamma) = \{\alpha \mid \Gamma\alpha \neq \{\}\}$ is finite. Sort contexts can be written as $[\alpha_1{:}S_1, \ldots, \alpha_n{:}S_n]$. The behaviour of type constructors is specified by declarations of the form $t : (\overline{S_n})C$ which have exactly the same meaning as

Type classes	C
Sorts	$S = \{C_1, \ldots, C_n\}$
Type variables	α
Type constructors	t
Types	$\tau = \alpha \mid t(\tau_1, \ldots, \tau_n)$
Type schemes	$\sigma = \tau \mid \forall\alpha{:}S.\sigma$
Identifiers	x
Expressions	$e = x \mid (e_0\ e_1) \mid \lambda x.e \mid \textbf{let}\ x = e_0\ \textbf{in}\ e_1$
Declarations	$d = \textbf{class}\ C\ \textbf{where}\ x : \forall\alpha{:}C.\sigma \mid \textbf{inst}\ t : (S_1, \ldots, S_n)C\ \textbf{where}\ x = e$
Programs	$p = d; p \mid e$

Figure 1: Syntax of Mini-Haskell types and expressions

the corresponding instance declarations. A set of such declarations is called a *signature* and is denoted by Δ.

Given a context Γ and a signature Δ, we can infer the sort of a type τ using the judgement $\Gamma, \Delta \vdash \tau : S$. The rules are shown in Figure 2. Remember that the sort $\{C\}$ and the class C are equivalent.

Having seen sort inference for Mini-Haskell types we are prepared for our main goal, type inference and type reconstruction for Mini-Haskell programs.

3 Type Inference Systems

In this section we present two type inference systems for Mini-Haskell. We start with a set of inference rules which define the types of Mini-Haskell programs and expressions. Then we proceed to a more restricted, syntax-directed set of rules, which will be the basis for the type inference algorithm.

As usual in type inference for ML-like languages, an *environment* is a finite mapping $E = [x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n]$ from identifiers to types. The *domain* of E is $\mathcal{D}om(E) = \{x_1, \ldots, x_n\}$. $E[x{:}\sigma]$ is a new map which maps x to σ and all other x_i to σ_i, and the free type variables in E are $\mathcal{FV}(E) = \mathcal{FV}(E(x_1)) \cup \ldots \cup \mathcal{FV}(E(x_n))$. If V is a set of type variables the restriction of Γ to variables not in V is $\Gamma \backslash V = [\alpha{:}\Gamma\alpha \mid \alpha \in \mathcal{D}om(\Gamma) - V]$.

The judgement $\Gamma, \Delta, E \vdash p : \sigma$ states that the Mini-Haskell program p is of type σ under Δ and the assumptions in Γ and E.

For a program p, which begins with a series of **class** and **inst** declarations, the typing rules in Figure 3 can be applied backwards to build up Δ and E. For instance, applying rule INST backwards adds the new declaration $t : (\overline{S_n})C$ to Δ in the first premise. The second and third premises compare the types of x and e in order to type-check $x = e$. These assumptions in Δ and E are used in the typing rules for expressions e in Figure 4.

The rules extend the classical system of Damas and Milner [5] by the notion of sorts, which are represented in contexts and in restricted quantifications of type variables. The assumption $\alpha \in \mathcal{FV}(\sigma)$ in $\forall$I is not really essential (for soundness) but simplifies the analysis later on. Its practical significance is discussed in Section 8.

In contrast to the CLASS and INST rules, the signature Δ remains fixed in the typing derivation for an expression.

3.1 Syntax-directed Type Inference

The next step towards a type checking algorithm is a more restricted set of rules that is sufficient for type reconstruction. The application of the rules is determined by the syntax of the expression whose type is to be computed. To distinguish the syntax-directed system we use $\vdash'$, ASM$'$ etc to denote its inferences and rules.

Definition 3.1 The type scheme $\sigma' = \forall\overline{\alpha'_n{:}S'_n}.\tau'$ is a *generic instance* of $\sigma = \forall\overline{\alpha_m{:}S_m}.\tau$ under Γ and Δ, written $\Gamma, \Delta \vdash \sigma \succeq \sigma'$, iff there exists a substitution θ such that

$$\begin{array}{rcl} \theta\tau & = & \tau', \\ \mathcal{D}om(\theta) & \subseteq & \{\overline{\alpha_m}\}, \\ \Gamma[\overline{\alpha'_n{:}S'_n}], \Delta & \vdash & \theta\alpha_i : S_i \quad [i = 1 \ldots m], \\ \{\overline{\alpha'_n}\} \cap \mathcal{FV}(\sigma) & = & \{\}. \end{array}$$

For the syntax-directed system, the rules APP and ABS remain unchanged, the quantifier rules are incorporated into ASM and LET, as shown in Figure 5.

$$\text{SI} \quad \frac{\begin{array}{c}[i=1\ldots n]\\ \vdots \\ \Gamma,\Delta \vdash \tau : C_i\end{array}}{\Gamma,\Delta \vdash \tau : \{C_1,\ldots,C_n\}}$$

$$\text{SE} \quad \frac{\Gamma,\Delta \vdash \tau : \{C_1,\ldots,C_n\}}{\Gamma,\Delta \vdash \tau : C_i} \qquad i=1\ldots n$$

$$\text{TVAR} \quad \frac{\Gamma(\alpha) = S}{\Gamma,\Delta \vdash \alpha : S}$$

$$\text{TCON} \quad \frac{t : (\overline{S_n})C \in \Delta \qquad \begin{array}{c}[i=1\ldots n]\\ \vdots \\ \Gamma,\Delta \vdash \tau_i : S_i\end{array}}{\Gamma,\Delta \vdash t(\overline{\tau_n}) : C}$$

Figure 2: The judgement $\Gamma,\Delta \vdash \tau : S$

$$\text{CLASS} \quad \frac{\Gamma,\Delta,E[x{:}\forall\alpha{:}C.\sigma] \vdash p : \sigma'}{\Gamma,\Delta,E \vdash \textbf{class } C \textbf{ where } x : \forall\alpha{:}C.\sigma; p : \sigma'}$$

$$\text{INST} \quad \frac{\begin{array}{c}\Gamma,\Delta \cup \{t : (\overline{S_n})C\},E \vdash p : \sigma' \\ E(x) = \forall\alpha{:}C.\sigma \quad \Gamma[\overline{\alpha_n{:}S_n}],\Delta,E \vdash e : [t(\overline{\alpha_n})/\alpha]\sigma\end{array}}{\Gamma,\Delta,E \vdash \textbf{inst } t : (\overline{S_n})C \textbf{ where } x = e; p : \sigma'}$$

Figure 3: The judgement $\Gamma,\Delta,E \vdash p : \sigma$

There is a straightforward correspondence between the two systems. The syntax-directed derivations are sound

Theorem 3.2 *If* $\Gamma,\Delta,E \vdash' e : \tau$ *then* $\Gamma,\Delta,E \vdash e : \tau$.

and in a certain sense complete w.r.t. the original system:

Theorem 3.3 *If* $\Gamma,\Delta,E \vdash e : \forall\overline{\alpha_n{:}S_n}.\tau$ *then* $\Gamma[\overline{\alpha_n{:}S_n}],\Delta,E \vdash' e : \tau$.

The last theorem clarifies in what sense $\vdash'$ works differently from $\vdash$: by applying the primed rules backwards, the sort constraints for type variables are stored solely in Γ, and not in the type scheme of e. For instance, the LET$'$ rule explicitly extends Γ. The $\succeq$ operation, used in the ASM$'$ rule, may introduce new type variables, whose sorts must be recorded in Γ. The syntax-directed system already has a very operational flavour. In order to make the transition from a type inference system to an algorithm we need one more ingredient: unification.

4 Unification of Types with Sort Constraints

This section deals with unification in the presence of sort constraints in the form of contexts. This problem can in principle be reduced to order-sorted unification, as done in [12], using the ordering coming from the inclusion between sorts. However, we have refrained from doing so because it is contrary to our quest for simplicity: involving order-sorted unification makes the algorithm appear more complicated than it actually is. In addition, the standard theory of order-sorted unification assumes that variables are tagged and would thus need to be reformulated anyway.

For the remainder of this paper we assume a fixed signature Δ. This is simply a notational device which avoids parameterizing judgements etc. by Δ.

In our setting, a *substitution* is a finite mapping from type variables to types. Substitutions are denoted by θ and δ; $\{\}$ is the empty substitution. Define $\mathcal{D}om(\theta) = \{\alpha \mid \theta\alpha \neq \alpha\}$ and $\mathcal{C}od(\theta) = \bigcup_{\alpha\in\mathcal{D}om(\theta)} \mathcal{FV}(\theta(\alpha))$.

Since sort information is maintained in contexts, we frequently work with pairs of contexts and substitutions.

$$\text{ASM}\quad \frac{}{\Gamma,\Delta,E\vdash x:E(x)}$$

$$\forall\text{E}\quad \frac{\Gamma,\Delta,E\vdash e:\forall\alpha{:}S.\sigma \quad \Gamma,\Delta\vdash\tau:S}{\Gamma,\Delta,E\vdash e:[\tau/\alpha]\sigma}$$

$$\forall\text{I}\quad \frac{\Gamma[\alpha{:}S],\Delta,E\vdash e:\sigma \quad \alpha\in\mathcal{FV}(\sigma)-\mathcal{FV}(E)}{\Gamma,\Delta,E\vdash e:\forall\alpha{:}S.\sigma}$$

$$\text{APP}\quad \frac{\Gamma,\Delta,E\vdash e_1:\tau_2\to\tau_1 \quad \Gamma,\Delta,E\vdash e_2:\tau_2}{\Gamma,\Delta,E\vdash(e_1\ e_2):\tau_1}$$

$$\text{ABS}\quad \frac{\Gamma,\Delta,E[x{:}\tau_1]\vdash e:\tau_2}{\Gamma,\Delta,E\vdash\lambda x.e:\tau_1\to\tau_2}$$

$$\text{LET}\quad \frac{\Gamma,\Delta,E\vdash e_1:\sigma_1 \quad \Gamma,\Delta,E[x{:}\sigma_1]\vdash e_2:\sigma_2}{\Gamma,\Delta,E\vdash\textbf{let}\ x=e_1\ \textbf{in}\ e_2:\sigma_2}$$

Figure 4: The judgement $\Gamma,\Delta,E\vdash e:\sigma$

$$\text{ASM}'\quad \frac{\Gamma,\Delta\vdash E(x)\succeq\tau}{\Gamma,\Delta,E\vdash' x:\tau}$$

$$\text{LET}'\quad \frac{\Gamma[\overline{\alpha_k{:}S_k}],\Delta,E\vdash' e_1:\tau_1 \quad \Gamma,\Delta,E[x{:}\forall\overline{\alpha_k{:}S_k}.\tau_1]\vdash' e_2:\tau_2 \quad \{\overline{\alpha_k}\}=\mathcal{FV}(\tau_1)-\mathcal{FV}(E)}{\Gamma,\Delta,E\vdash'\textbf{let}\ x=e_1\ \textbf{in}\ e_2:\tau_2}$$

Figure 5: The judgement $\Gamma,\Delta,E\vdash' e:\sigma$

A substitution θ obeys the sort constraints of Γ in the context of Γ', written $\Gamma'\vdash\theta:\Gamma$, iff $\Gamma',\Delta\vdash\theta\alpha:\Gamma\alpha$ for all α. Because $\Gamma',\Delta\vdash\theta\alpha:\Gamma\alpha$ is trivially fulfilled if $\Gamma\alpha=\{\}$ it suffices to require $\Gamma',\Delta\vdash\theta\alpha:\Gamma\alpha$ for all $\alpha\in\mathcal{D}om(\Gamma)$.

As usual, we define an ordering on context-substitution pairs:

$$(\Gamma,\theta)\geq(\Gamma',\theta')\quad\Leftrightarrow\quad\exists\delta.\ \delta\theta=\theta'\ \wedge\ \Gamma'\vdash\delta:\Gamma$$

where $\delta\theta$ is their composition: $(\delta\theta)(s)=\delta(\theta(s))$.

The set of unifiers of τ_1 and τ_2 w.r.t. Γ, written $\mathcal{U}(\Gamma,\tau_1{=}\tau_2)$, consists of the following context-substitution pairs:

$$\mathcal{U}(\Gamma,\tau_1{=}\tau_2)=\{(\Gamma',\theta)\mid\theta\tau_1=\theta\tau_2\ \wedge\ \Gamma'\vdash\theta:\Gamma\}$$

A unifier $(\Gamma_0,\theta_0)\in\mathcal{U}(\Gamma,\tau_1{=}\tau_2)$ is *most general* if $(\Gamma_0,\theta_0)\geq(\Gamma_1,\theta_1)$ for all $(\Gamma_1,\theta_1)\in\mathcal{U}(\Gamma,\tau_1{=}\tau_2)$. We say that unification modulo Δ is *unitary* if for all Γ and $\tau_1{=}\tau_2$ the set $\mathcal{U}(\Gamma,\tau_1{=}\tau_2)$ is empty or contains a most general unifier.

A signature Δ is called *coregular* if for all type constructors t and all classes C the set

$$D(t,C)\ =\ \{\overline{S_n}\mid t{:}(\overline{S_n})C\in\Delta\}$$

is either empty or contains a greatest element w.r.t. the component-wise ordering of the $\overline{S_n}$. If Δ is coregular let $\mathcal{D}om(t,C)$ return the greatest element of $D(t,C)$ or fail if $D(t,C)$ is empty.

Sorted unification can be expressed as unsorted unification plus constraint solving. Given a coregular signature Δ, this has the following simple form:

$$\begin{array}{l}unify(\Gamma,\tau_1{=}\tau_2)\ =\\ \quad\mathsf{let}\quad\theta\ =\ mgu(\tau_1{=}\tau_2)\\ \qquad\Gamma_c\ =\ \bigcup_{\alpha\in\mathcal{D}om(\theta)}constrain(\theta\alpha,\Gamma\alpha)\\ \quad\mathsf{in}\ (\Gamma_c\cup(\Gamma\backslash\mathcal{D}om(\theta)),\theta)\end{array}$$

where mgu computes an unsorted mgu (in particular we assume that θ is idempotent and that $\mathcal{D}om(\theta)\cup\mathcal{C}od(\theta)\subseteq\mathcal{FV}(\tau_1{=}\tau_2)$) or fails if none exists.

A context Γ is *more general* than Γ', written $\Gamma\geq\Gamma'$, if $\Gamma\alpha\subseteq\Gamma'\alpha$ for all α. The union of two sort contexts is defined by

$$\Gamma_1\cup\Gamma_2\ =\ [\alpha:\Gamma_1\alpha\cup\Gamma_2\alpha\mid\alpha\in\mathcal{D}om(\Gamma_1)\cup\mathcal{D}om(\Gamma_2)]$$

and $constrain(\tau,S)$ computes the most general context Γ such that $\Gamma,\Delta\vdash\tau:S$:

$$\begin{array}{rcl}constrain(\alpha,S) & = & [\alpha{:}S]\\ constrain(t(\overline{\tau_n}),S) & = & \bigcup_{C\in S}constr(\overline{\tau_n},\mathcal{D}om(t,C))\end{array}$$

$$constr(\overline{\tau_n}, \overline{S_n}) \quad = \quad \bigcup_{i=1\ldots n} constrain(\tau_i, S_i)$$

Thus *unify* fails if *mgu* fails or if some $\mathcal{D}om(t, C)$ used in *constrain* does not exist. Soundness and completeness of *constrain* are captured by the following lemmas:

Lemma 4.1 *$constrain(\tau, S), \Delta \vdash \tau : S$ or $constrain(\tau, S)$ fails.*

Proof by induction on the structure of τ. □

Lemma 4.2 *$constrain(\theta\tau, S) \vdash \theta : constrain(\tau, S)$.*

Proof by comparing the calling trees of $constrain(\theta\tau, S)$ and $constrain(\tau, S)$, using Lemma 4.1 at the leaves of the $constrain(\tau, S)$ tree. □

Lemma 4.3 *If $\Gamma, \Delta \vdash \tau : S$ then $constrain(\tau, S)$ is defined and more general than Γ.*

Proof by induction on the structure of τ. □

Theorem 4.4 *If Δ is coregular, unify computes a most general unifier.*

Proof To show soundness, let $unify(\Gamma, \tau_1{=}\tau_2)$ terminate with result (Γ_0, θ_0). It follows directly that $\theta_0\tau_1 = \theta_0\tau_2$. It remains to be seen that $\Gamma_0 \vdash \theta_0\alpha : \Gamma\alpha$ for all α. If $\alpha \notin \mathcal{D}om(\theta_0)$, then $\Gamma\alpha \subseteq \Gamma_0\alpha$ and the claim follows trivially. If $\alpha \in \mathcal{D}om(\theta_0)$ then $\Gamma \supseteq \Gamma_c = \bigcup_{\beta \in \mathcal{D}om(\theta_0)} constrain(\theta_0\beta, \Gamma\beta) \supseteq constrain(\theta_0\alpha, \Gamma\alpha)$ and the claim follows from Lemma 4.1.

To show completeness let $(\Gamma_1, \theta_1) \in \mathcal{U}(\Gamma, \tau_1{=}\tau_2)$, i.e. $\theta_1\tau_1 = \theta_1\tau_2$ and $\Gamma_1 \vdash \theta_1 : \Gamma$. Since τ_1 and τ_2 have an unsorted unifier θ_1, $mgu(\tau_1{=}\tau_2)$ is defined and yields a substitution θ_0 such that $\theta_1 = \delta\theta_0$ for some δ. Definedness of $unify(\Gamma, \tau_1{=}\tau_2)$ also requires definedness of $constrain(\theta_0\alpha, \Gamma\alpha)$: since $\Gamma_1 \vdash \theta_1\alpha : \Gamma\alpha$, Lemma 4.3 implies definedness of $constrain(\theta_1\alpha, \Gamma\alpha)$ which easily yields definedness of $constrain(\theta_0\alpha, \Gamma\alpha)$. Thus $unify(\Gamma, \tau_1{=}\tau_2)$ terminates with a result (Γ_0, θ_0).

It remains to be shown that $\Gamma_1 \vdash \delta : \Gamma_0$. If $\beta \in \mathcal{D}om(\theta_0)$ then $\Gamma_0\beta = \{\}$ and hence $\Gamma_1 \vdash \delta\beta : \Gamma_0\beta$ holds trivially. Now assume $\beta \notin \mathcal{D}om(\theta_0)$. Thus $\Gamma_0\beta = \Gamma_c\beta \cup \Gamma\beta$. From $\Gamma_1 \vdash \theta_1 : \Gamma$ it follows that $\Gamma_1 \vdash \delta\beta : \Gamma\beta$. Proving $\Gamma_1 \vdash \delta : \Gamma_c$ is more involved. From Lemma 4.2 it follows that $constrain(\theta_1\alpha, \Gamma\alpha) \vdash \delta : constrain(\theta_0\alpha, \Gamma\alpha)$ for any α and hence $\bigcup_{\alpha \in \mathcal{D}om(\theta_0)} constrain(\theta_1\alpha, \Gamma\alpha) \vdash \delta : \bigcup_{\alpha \in \mathcal{D}om(\theta_0)} constrain(\theta_0\alpha, \Gamma\alpha)$, i.e. $\bigcup_{\alpha \in \mathcal{D}om(\theta_0)} constrain(\theta_1\alpha, \Gamma\alpha) \vdash \delta : \Gamma_c$ $(*)$. Since $\Gamma_1 \vdash \theta_1\alpha : \Gamma\alpha$, Lemma 4.3 implies $\Gamma_1 \leq constrain(\theta_1\alpha, \Gamma\alpha)$ and hence $\Gamma_1 \leq \bigcup_{\alpha \in \mathcal{D}om(\theta_0)} constrain(\theta_1\alpha, \Gamma\alpha)$. Thus $\Gamma_1 \vdash \delta : \Gamma_c$ follows from $(*)$ by monotonicity. □

Theorem 4.5 *Unification modulo Δ is unitary iff Δ is coregular.*

Proof The "if" direction is a consequence of Theorem 4.4. For the "only if" direction let Δ not be coregular. Let $\leq$ denote the component-wise ordering of sort tuples $\overline{S_n}$. Thus there are two declarations $t : (\overline{S_n})C$ and $t : (\overline{T_n})C$, $\overline{S_n} \not\leq \overline{T_n}$, and $\overline{S_n} \not\geq \overline{T_n}$, such that there is no third declaration $t : (\overline{U_n})C$, and $\overline{S_n}, \overline{T_n} \leq \overline{U_n}$. Hence the unification problem $([\beta{:}C], t(\overline{\alpha_n}){=}\beta)$ does not have a most general unifier. Two maximal ones are $([\overline{\alpha_n{:}S_n}], \theta)$ and $([\overline{\alpha_n{:}T_n}], \theta)$ where $\theta = \{\beta \to t(\overline{\alpha_n})\}$. □

Thus we have a precise characterization of those signatures where principal types exist.

5 Principal Types and Algorithm $\mathcal{I}$

The above syntax-directed rule system is nondeterministic, since rule ASM′ can choose any instance of the type of the identifier x. To obtain a deterministic algorithm, we refine the syntax directed system such that it keeps types as general as possible. The result is algorithm $\mathcal{I}$ in Figure 6. In this section we assume that Δ is coregular — otherwise *unify* is not well-defined.

Algorithm $\mathcal{I}$ follows the same pattern as Milner's original algorithm of the same name [10]: the type of an expression e is computed by traversing e in a top-down manner. $\mathcal{I}$ returns a quadruple $(V, \Gamma, \theta, \tau)$, where $\theta\tau$ is the type of e under the context Γ. The parameter V contains all "used" variables, i.e. variables that occur in τ or in θ or in E. Thus a type variable $\alpha \notin V$ is a "new" variable. Observe the different let-constructs: the one on the left hand side is in the object language, the ones on the right are part of the type inference algorithm.

For an environment E and a substitution θ, define $\theta E = [x : \theta(E(x)) \mid x \in \mathcal{D}om(E)]$. We say E is *closed* if $\mathcal{FV}(E) = \{\}$. The *free variables of a substitution* θ are defined as $\mathcal{FV}(\theta) = \{\mathcal{FV}(\theta\alpha) \cup \{\alpha\} \mid \alpha \in \mathcal{D}om(\theta)\}$. Let $[]$ denote the empty context.

Theorem 5.1 (Correctness of $\mathcal{I}$) *If $\mathcal{I}(V, \Gamma, \theta, E, e) = (V', \Gamma', \theta', \tau)$ and $\mathcal{D}om(\Gamma) \cup \mathcal{FV}(\theta) \cup \mathcal{FV}(E) \subseteq V$ then $\Gamma', \Delta, \theta'E \vdash' e : \theta'\tau$.*

Proof by induction on the structure of e, using a number of auxiliary lemmas. □

A type reconstruction algorithm should compute most general types, usually called principal types, if they exist.

Definition 5.2 Let E be a closed environment. The type scheme σ is a *principal type* of an expression e w.r.t. Δ and E if $[], \Delta, E \vdash e : \sigma$ and if $[], \Delta, E \vdash e : \sigma'$ implies $[], \Delta \vdash \sigma \succeq \sigma'$.

$$
\begin{array}{rcl}
\mathcal{I}(V,\Gamma,\theta,E,e) = \mathsf{case}\ e\ \mathsf{of} & & \\
x & \Rightarrow & \mathsf{let}\ \forall\overline{\alpha_n{:}S_n}.\tau = E(x) \\
 & & \qquad \beta_i \notin V \quad [i=1\ldots n] \\
 & & \mathsf{in}\ (V \cup \{\overline{\beta_n}\}, \Gamma[\overline{\beta_n{:}S_n}], \theta, [\overline{\beta_n/\alpha_n}]\tau) \\
\lambda x.e & \Rightarrow & \mathsf{let}\ \alpha \notin V \\
 & & \qquad (V',\Gamma',\theta',\tau) = \mathcal{I}(V\cup\{\alpha\},\Gamma,\theta,E[x{:}\alpha],e) \\
 & & \mathsf{in}\ (V',\Gamma',\theta',\alpha\rightarrow\tau) \\
(e_1\ e_2) & \Rightarrow & \mathsf{let}\ (V_1,\Gamma_1,\theta_1,\tau_1) = \mathcal{I}(V,\Gamma,\theta,E,e_1) \\
 & & \qquad (V_2,\Gamma_2,\theta_2,\tau_2) = \mathcal{I}(V_1,\Gamma_1,\theta_1,E,e_2) \\
 & & \qquad \alpha \notin V_2 \\
 & & \qquad (\Gamma',\theta') = unify(\Gamma_2,\theta_2\tau_1 = \theta_2\tau_2 \rightarrow \alpha) \\
 & & \mathsf{in}\ (V_2\cup\{\alpha\},\Gamma',\theta'\theta_2,\alpha) \\
\textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \Rightarrow & \mathsf{let}\ (V_1,\Gamma_1,\theta_1,\tau_1) = \mathcal{I}(V,\Gamma,\theta,E,e_1) \\
 & & \qquad \{\overline{\alpha_n}\} = \mathcal{FV}(\theta_1\tau_1) - \mathcal{FV}(\theta_1 E) \\
 & & \mathsf{in}\ \mathcal{I}(V_1,\Gamma_1\backslash\{\overline{\alpha_n}\},\theta_1,E[x:\forall\overline{\alpha_n{:}\Gamma_1\alpha_n}.\theta_1\tau_1],e_2)
\end{array}
$$

Figure 6: Algorithm $\mathcal{I}$

The following lemma is crucial for establishing the principal type theorem.

Lemma 5.3 *If* $\Gamma',\Delta,E' \vdash' e:\tau'$ *where* $E' = \delta'\theta E$ *and* $\Gamma' \vdash \delta' : \Gamma$ *then there exist* V *and* δ'_1 *such that*

$$
\begin{array}{l}
\mathcal{D}om(\Gamma) \cup \mathcal{FV}(\theta) \cup \mathcal{FV}(E) \subseteq V, \\
\mathcal{I}(V,\Gamma,\theta,E,e) = (V_1,\Gamma_1,\theta_1,\tau_1), \\
E' = \delta'_1\theta_1 E, \\
\tau' = \delta'_1\theta_1\tau_1, \\
\Gamma' \vdash \delta'_1 : \Gamma_1.
\end{array}
$$

Proof by induction on the structure of e, using a number of auxiliary lemmas. □

Theorem 5.4 *If* $[],\Delta,E \vdash' e:\tau'$ *and* E *is closed then* $\mathcal{I}(\{\},[],\{\},E,e) = (V,\Gamma,\theta,\tau)$ *and* $\forall\overline{\alpha_n{:}\Gamma\alpha_n}.\theta\tau$ *is a principal type of* e *w.r.t.* Δ *and* E*, where* $\overline{\alpha_n} = \mathcal{FV}(\theta\tau)$.

Proof follows from Lemma 5.3. □

We may restrict our attention to closed environments because CLASS and INST declarations cannot introduce free variables into an environment E: type schemes in CLASS declarations must be closed.

Algorithm $\mathcal{I}$ relies on *unify* which has only been defined for coregular signatures. Hence it remains to be seen if Mini-Haskell's CLASS and INST declarations yield coregular signatures. In fact they do if restricted by the following context condition which is the result of translating the restrictions actually adopted in Haskell [6, Sec. 4.3.2] to Mini-Haskell:

> For every class C and type constructor t there is at most one instance declaration **inst** $t : (\overline{S_n})C$.

The more complex restrictions for Haskell collapse to this single requirement because of the absence of subclasses.

One can easily see that a signature is coregular if it is derived from a set of instance declarations which meet the above restriction. The converse does not hold: the following declarations violate the above restriction

$$
\begin{array}{l}
\textbf{class}\ C\ \textbf{where}\ c : \forall\alpha{:}C.\alpha; \\
\textbf{inst}\ int : ()C\ \textbf{where}\ c = 1; \\
\textbf{inst}\ int : ()C\ \textbf{where}\ c = 2;
\end{array}
$$

although they give rise to the coregular signature $\{int : ()C\}$[2]. The reason is that although coregularity suffices for the existence of principal types, it does not preclude semantic ambiguities: the expression $c+1$ is definitely of type *int* but may evaluate to either 2 or 3, depending on which instance is chosen for c. Hence the above additional restriction which rules out situations like these.

6 Haskell = Mini-Haskell + Subclasses

Compared to standard Haskell, the main missing type class feature of Mini-Haskell is the notion of subclasses. Subclasses are clearly beneficial as far as methodical aspects are concerned. However, for semantics and type reconstruction, subclasses are syntactic sugar and can be eliminated. We present two methods for handling subclasses. Subclasses can be integrated into our type inference system by slightly changing some definitions.

[2] It is easy to give similar examples where the *INST* declarations not only differ in the function (or constant) provided.

As an alternative, we give a method for eliminating subclasses. We assume the restrictions on instance declarations adopted in actual Haskell, which are the reason why the latter method is particularly simple.

Assume a set of Haskell classes C with an ordering $\preceq$. If $C \preceq D$, we say that C is a subclass of D and D is a superclass of C. To accommodate subclasses, we extend the Mini-Haskell judgement that defines sort membership of types, $\Gamma, \Delta \vdash \tau : S$ to the judgement $\Gamma, \Delta \vdash^H \tau : S$. The rules SI, SE, TVAR, and TCON are the same for the Haskell-judgement $\vdash^H$; we only have one additional rule for the subclass ordering:

$$\text{SUB} \quad \frac{\Gamma, \Delta \vdash^H \tau : C \quad C \preceq D}{\Gamma, \Delta \vdash^H \tau : D}$$

6.1 Integrating Subclasses

Our unification and type inference algorithms easily accommodate subclasses. We give a brief sketch of the necessary extensions. In essence, only the unification algorithm is affected by the integration of subclasses. The key idea is to redefine the ordering on sorts:

$$S \leq S' \quad \Leftrightarrow \quad \forall C' \in S' \exists C \in S.\ C \preceq C'$$

As above, the ordering $\preceq$ extends in the componentwise way to sequences of sorts $\overline{S_n}$. Then for a type constructor t and a class C, the definition of $D(t, C)$ is generalized to

$$D(t, C) = \{\overline{S_n} \mid \exists D \preceq C . t : (\overline{S_n})D \in \Delta\}$$

Coregularity is defined as above: $D(t, C)$ must be empty or have a maximal element for all t and C. Similar to above, it can be shown that coregularity is necessary and sufficient for unitary unification and hence principal types. The restrictions adopted in Haskell imply coregularity also in the presence of subclasses and are also motivated by semantic reasons, as discussed in Section 8.

6.2 Eliminating Subclasses

The idea behind translating Haskell type classes into Mini-Haskell is to replace Haskell classes by the set of their super sorts. Let $C\uparrow = \{C' \mid C \preceq C'\}$ and correspondingly $S\uparrow = \bigcup_{C \in S} C\uparrow$. Then a Haskell-like class declaration

$$\textbf{class } C \preceq S \textbf{ where } x = \forall \alpha{:}C.\forall \overline{\alpha_n{:}S_n}.\tau$$

states that C is a subclass of all $C' \in S$. Assume that the subclass ordering $\preceq$ has been built up from the CLASS declarations. Then the above CLASS declaration translates into the Mini-Haskell declaration

$$\textbf{class } C \textbf{ where } x = \forall \alpha{:}C.\forall \overline{\alpha_n{:}S_n\uparrow}.\tau$$

A Haskell instance declaration

$$\textbf{inst } t : (\overline{S_n})C \textbf{ where } x = e$$

simply translates into

$$\textbf{inst } t : (\overline{S_n\uparrow})C \textbf{ where } x = e$$

In the last translation it is not necessary to expand C into several declarations, one for each element of $C\uparrow$, since Haskell requires that if $t(\ldots)$ is declared as an instance for C, then it also has to be an instance of all super classes of C. Another consequence of this restriction is that the rule SUB for type classification is redundant in the case of Haskell.

Assume a Mini-Haskell class signature Δ that also includes an ordering $\preceq$ on classes. For this Haskell-like signature, let $\Delta\uparrow$ denote the pure Mini-Haskell signature obtained by translating all instance declarations in Δ with the above rule. Define further $\Gamma\uparrow = [\alpha : (\Gamma\alpha)\uparrow \mid \alpha \in \mathcal{D}om(\Gamma)]$. We show the correctness of the translation with the following theorem.

Theorem 6.1 *Let Δ be a Mini-Haskell signature with an ordering on classes. Then $\Gamma, \Delta \vdash^H \tau : C$ iff $\Gamma\uparrow, \Delta\uparrow \vdash \tau : C\uparrow$.*

It should be noted that this translation loses some information about the class ordering. The difference to the former method emerges when Mini-Haskell is translated into a functional language without classes, e.g. ML.

For instance, assume the class ordering $Ord \preceq Eq$. Then a type τ of sort Ord would be translated into a type of sort $\{Ord, Eq\}$ by the latter method. This is equivalent in the sense that an object of type τ still provides the same functions. The particular instances of these functions cannot be computed at compile time and are usually carried around at run time in the form of dictionaries (of functions). Now we can see the disadvantage of the latter method: an object of type τ requires two dictionaries instead of only one. In contrast, after integrating subclasses, the two sorts $\{Ord, Eq\}$ and Ord are equivalent (since $\{Ord, Eq\} \preceq Ord$ and $Ord \preceq \{Ord, Eq\}$) and only one dictionary has to be used for either one.

7 Related Work

The structure of algorithm $\mathcal{I}$ is very close to that of Milner's algorithm of the same name [10]. Apart from the fact that our version of $\mathcal{I}$ is purely applicative (hence we carry the substitution and the set of used variables around explicitly), the main difference is that we also have to maintain a set of constraints Γ. In fact, this

is the only real difference to Milner's algorithm. It is interesting to note that Milner's first formulation of type inference uses a purely functional algorithm $\mathcal{W}$ which is non-incremental, i.e. does not take a given substitution θ and produce an extended one θ', but computes the result substitution from scratch. Whereas he considers $\mathcal{I}$ merely a more efficient refinement, in our case there is a very strong simplicity argument in favour of the incremental version: when typing an application $(e_1\ e_2)$ it is far simpler to type e_2 under the constraints due to e_1 than compute two separate constraints Γ_1 and Γ_2 and having to merge them afterwards.

Probably the first combination of ML-style polymorphism and parametric overloading (as opposed to finite overloading as in Hope [2]) was presented by Kaes [8]. His language is in fact very close to our Mini-Haskell, except that he does not introduce classes explicitly. More importantly, he does not use contexts to record information about type variables but tags the type variables directly.

The original version of type classes as presented by Wadler and Blott [14] was significantly more powerful than what went into Haskell, the reason being that the original system was undecidable, as shown later by Volpano and Smith [13]. The relationship to Haskell proper is discussed in Section 2.

Nipkow and Snelting [12] realized that type inference for type classes can be formulated as an extension of ordinary ML-style type inference with order-sorted unification, i.e. simply by changing the algebra of types and the corresponding unification algorithm. Although this was an interesting theoretical insight, it only lead to a simple algorithm for a restricted version of Haskell where each type variable is constrained by exactly one class. In addition it was not possible to identify ambiguous typings like $[\alpha{:}C], \Delta, E \vdash e : int$ because there was no notion of contexts and type variables were tagged with their sort. Both problems have been eliminated in the present paper.

An interesting extension of Haskell using the notion of "predicated types" was designed and implemented by Mark Jones [7]. The main difference is that he allows arbitrary predicates $P(\tau_1, \ldots, \tau_n)$ over types as opposed to our membership constraints $\alpha : S$.

Independently of our own work Chen, Hudak and Odersky [3] developed an extension of type classes using similar techniques and arriving at a similar type reconstruction algorithm. Since their type system is more general, they use different and more involved formalisms, in particular for unification. In contrast, we reduce unification to its essence by splitting it into standard unification plus constraint solving. This enables us to give a sufficient and necessary criterion for unitary unification, which is required for principal types. As discussed in Section 5, the restrictions in Haskell easily archive unitary unification. An example where unitary unification is more difficult is the integration of subclasses (see Sec. 6.1), where $Dom(t, C)$ is defined differently.

Kaes [9] presents an extension of Hindley/Milner polymorphism with overloading, subtypes and recursive types. Due to the overall complexity of the resulting system, the simplicity of the pure system for overloading is lost.

8 Ambiguity

We would like to conclude this paper with a discussion of the ambiguity problem which affects most type systems with overloading. It is caused by the fact that although a program may have a unique type, it's semantics is not well-defined. According to our rules, the program

$$\begin{array}{l}\textbf{class}\ C\ \textbf{where}\ f : \forall\alpha{:}C.\alpha \to int;\\ \textbf{class}\ D\ \textbf{where}\ c : \forall\alpha{:}D.\alpha;\\ (f\ c)\end{array}$$

has type int in any context containing an assumption $\alpha : \{C, D\}$. Yet the program has no semantics because there are no instances of f and c at all. If there were multiple instances of both C and D, it would be impossible to determine which one to use in the expression $(f\ c)$.

Motivated by such examples, a typing $\Gamma, \Delta, E \vdash e : \sigma$ is usually defined to be *ambiguous* if there is a type variable in Γ which does not occur free in σ or E. However, the only formal proof of a relationship between such ambiguous types and ill-defined semantics that we are aware of is due to Blott [1]. Since we have not provided a semantics for our language, we have not introduced ambiguity formally. Nevertheless there is one place in our inference system where we anticipate a particular treatment of ambiguity. In rule $\forall$I, the proviso $\alpha \in \mathcal{FV}(\sigma)$ is intended to propagate ambiguity problems: with this restriction, the expression $\textbf{let}\ x = (f\ c)\ \textbf{in}\ 5$ (preceded by classes C and D as declared above) has type int only in a context containing an assumption $\alpha : \{C, D\}$. If the proviso is dropped, the expression also has type int in the empty context, thus disguising the local ambiguity. The reason is that x can be given the ambiguous type $\forall\alpha{:}\{C, D\}.int$, but since x does not occur in 5, this does not matter. Although in a lazy language x need not be evaluated and hence the semantics of the whole **let** is indeed unambiguous, we would argue that for pragmatic reasons it advisable to flag ambiguities whenever they arise.

From this discussion it is obvious that a semantics and a coherence proof for the type system w.r.t. a semantics are urgently needed.

References

[1] S. Blott. *An approach to overloading with polymorphism.* PhD thesis, Dept. of Computing Science, University of Glasgow, 1992.

[2] R. Burstall, D. MacQueen, and D. Sannella. Hope: an experimental applicative language. In *Proc. 1980 LISP Conference*, pages 136–143, 1980.

[3] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Proc. ACM Conf. Lisp and Functional Programming*, 1992. To appear.

[4] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[5] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. Principles of Programming Languages*, pages 207–212, 1982.

[6] P. Hudak, S. P. Jones, and P. Wadler. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[7] M. P. Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *Proc. European Symposium on Programming*, pages 287–306. LNCS 582, 1992.

[8] S. Kaes. Parametric overloading in polymorphic programming languages. In *Proc. 2nd European Symposium on Programming*, pages 131–144. LNCS 300, 1988.

[9] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. ACM Conf. Lisp and Functional Programming*, 1992. To appear.

[10] R. Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

[11] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[12] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, pages 1–14. LNCS 523, 1991.

[13] D. M. Volpano and G. S. Smith. On the complexity of ML typability with overloading. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, pages 15–28. LNCS 523, 1991.

[14] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 60–76, 1989.