Technical Report

UCAM-CL-TR-200
ISSN 1476-2986

Number 200

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Type classes and overloading resolution via order-sorted unification

Tobias Nipkow, Gregor Snelting

August 1990

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Type Classes and Overloading Resolution via Order-Sorted Unification*

Tobias Nipkow†
University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
England

Gregor Snelting‡
Technische Hochschule Darmstadt
Praktische Informatik
Magdalenenstr. 11c
D-61 Darmstadt
Fed. Rep. of Germany

Revised version, June 1991

Abstract

We present a type inference algorithm for a Haskell-like language based on order-sorted unification. The language features polymorphism, overloading, type classes and multiple inheritance. Class and instance declarations give rise to an order-sorted algebra of types. Type inference essentially reduces to the Hindley/Milner algorithm where unification takes place in this order-sorted algebra of types. The theory of order-sorted unification provides simple sufficient conditions which ensure the existence of principal types. The semantics of the language is given by a translation into ordinary λ-calculus. We prove the correctness of our type inference algorithm with respect to this semantics.

*To appear in Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture, Springer Verlag.

†E-mail: Tobias.Nipkow@cl.cam.ac.uk. Research supported by ESPRIT BRA 3245, *Logical Frameworks*.

‡E-mail: snelting@pi.informatik.th-darmstadt.de.

1 Introduction

In a recent paper on order-sorted unification, Meseguer et al. [8] state

> An application of order-sorted unification that seems to have escaped prior notice is to polymorphism for typed functional languages, in the case where there are subtypes as well as polymorphic type constructors.

In the spirit of this remark we show that for languages like Haskell [6] which feature polymorphism, type classes and overloading, order-sorted unification can indeed be the basis of a type inference system which is much simpler than those which have been proposed previously.

The key idea of this paper is to introduce a three-level system of values, types and partially ordered *sorts* that classify types. This is in contrast to many type systems used in the area of term-rewriting, e.g. OBJ [4], where types[1] are partially ordered. Although we assume that the quotation above was meant to apply to these two-level systems, Haskell's combination of polymorphism and type classes cries out for a three-level treatment with ordered sorts. Sorts are useful even for Standard ML [9] where we find a distinction between general types and equality types. This can be described by a signature with only two sorts, Eq and Ω, where $Eq < \Omega$.

Figure 1 defines the abstract syntax of our language *Mini-Haskell* which is the focus of our investigations. Although it is not mandatory that the reader is familiar with either the Haskell Report [6] or the paper by Wadler and Blott [14], that is where he should turn for motivating examples. In addition to the well-known constructs of functional languages, Mini-Haskell offers *class* and *instance* declarations. A class declaration is a named collection of function declarations. The types of these functions depend on a parameter, the instance type. Informally, a type belongs to some class γ if it provides the functions x_i associated with γ. This relationship is expressed formally in an instance declaration: it asserts that some type constructor χ returns a type of class γ if the arguments to χ are of class $\gamma_1, \ldots, \gamma_n$, and it defines the functions x_i introduced in the class declaration for γ. As there can be multiple instances of a class, there can be multiple definitions of the x_i. This provides for a kind of overloading where the type of each x_i is an instance of the generic type as defined in the class declaration. In addition, Mini-Haskell features multiple inheritance: a class γ may depend on superclasses $\gamma_1, \ldots, \gamma_n$, meaning that γ inherits all functions of the γ_i. In accordance with the Haskell report we require that all class and instance declarations must be top level; this avoids some subtle problems with nested classes.

Mini-Haskell enforces a restriction also present in the Haskell Report: classes express properties of individual types, not relationships between types. The latter, more general interpretation was introduced in [14], but we allow only type classes with one type parameter. Hence the type variable a in the Haskell notation[2] (γ_1 a, ..., γ_m a) => γ a and (γ_1 a_1, ..., γ_n a_n) => $\gamma(\chi\, a_1 \ldots a_n)$, where χ is an n-ary type constructor, can be dropped. Instead we write $\gamma \leq \gamma_1, \ldots, \gamma_n$ and $\chi : (\gamma_1, \ldots, \gamma_n)\gamma$.

For reasons of presentation, we have restricted the types of the functions x_i to be of the form $\forall\alpha_\gamma.\tau_i$, rather than $\forall\alpha_\gamma.\sigma_i$, where σ_i could again be quantified. This eliminates a certain amount of notational overhead, without being unreasonable: all of the examples in the Haskell Report fit the simpler scheme. In addition, several other context conditions must be satisfied, e.g. "No type constructor can be declared as an instance of a particular class more than once in the same scope"; these conditions are discussed later.

[1] which are usually called *sorts* in that context, making the terminology somewhat confusing.

[2] Note that each argument of χ is determined by a single class γ_i, rather than some finite list of classes as in Haskell. This is nonessential (see Section 5). We also assume that all a_i are distinct. We believe that this was the intention of the Haskell definers. It is not clear whether our ideas work without this linearity assumption.

Type classes	γ
Type variables	α_γ
Type constructors	χ
Types	$\tau = \alpha_\gamma \mid \chi(\tau_1, \ldots, \tau_n)$
Type schemes	$\sigma = \tau \mid \forall \alpha_\gamma . \sigma$
Identifiers	x
Expressions	$e = x$
	$\mid (e_0\ e_1)$
	$\mid \lambda x.e$
	$\mid$ **let** $x = e_0$ **in** e_1
Declarations	$d =$ **class** $\gamma \leq \gamma_1, \ldots, \gamma_m$ **where** $x_1 : \forall \alpha_\gamma . \tau_1, \ldots, x_k : \forall \alpha_\gamma . \tau_k$
	$\mid$ **inst** $\chi : (\gamma_1, \ldots, \gamma_n)\gamma$ **where** $x_1 = e_1, \ldots, x_k = e_k$
Programs	$p = d_1; \ldots d_n; e$

Ω is the universal type class

α means α_Ω

int, *float*, *char*, $list(\alpha)$, $pair(\alpha, \beta)$, $\alpha \to \beta$ are type constructors[3]

Figure 1: Syntax of Mini-Haskell types and expressions

Wadler and Blott [14] present a type inference system for what appears to be a superset of Mini-Haskell. The main difference is that in their syntax, class definitions are no longer present. Instead, they introduce so-called predicated types, as well as inference rules for the introduction and elimination of such types. The resulting inference system is rather complicated, and it is not clear whether it leads to a terminating algorithm.

Upon analysis of their article, we detected that a much simpler inference system for Mini-Haskell can be constructed, provided we replace ordinary unification in the Hindley-Milner system with order-sorted unification. Therefore, we first recall basic facts about order-sorted unification. We then present our type inference system, and provide some illuminating examples. In addition, we discuss conditions which ensure the existence of principal types. Finally, we define a translation from Mini-Haskell to Mini-ML [1] and show that the inferred types and the semantics given by the translation fit together.

2 Order-sorted terms and order-sorted unification

An *order-sorted signature* is a triple $(S, \leq, \Sigma)$, where S is a set of sorts, $\leq$ a partial order on S, and Σ a family $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ of not necessarily disjoint sets of operator symbols. We assume that S and Σ are finite. For notational convenience, we often write $f : (w)s$ instead of $f \in \Sigma_{w,s}$; $(w)s$ is called an *arity*[4] and $f : (w)s$ a *declaration*. The signature $(S, \leq, \Sigma)$ is often identified with Σ. An S-sorted variable set is a family $V = \{V_s \mid s \in S\}$ of disjoint, nonempty sets. For $x \in V_s$ we also write $x : s$ or x_s.

The set of *order-sorted terms* of sort s freely generated by V, $T_\Sigma(V)_s$, is the least set satisfying

- if $x \in V_{s'}$ and $s' \leq s$, then $x \in T_\Sigma(V)_s$
- if $f \in \Sigma_{w,s'}$, $w = s_1..s_n$, $s' \leq s$, and $t_i \in T_\Sigma(V)_{s_i}$ for all $i = 1..n$, then $f(t_1, \ldots, t_n) \in T_\Sigma(V)_s$.

[3] Except for $\alpha \to \beta$, the collection is arbitrary. Type constructors have global scope and are predefined.

[4] The term *type* and the notation $w \to s$ are reserved for the types in Mini-Haskell.

In contrast to sort-free terms and variables, order-sorted variables and terms always have a sort. Terms must be sort-correct, that is, subterms of a compound term must be of an appropriate sort as required by the arities of the term's operator symbol. Note that an operator symbol may have not just one arity (as in classical homogeneous or heterogeneous term algebras), but may have *several* arities. As a consequence, each term may have several sorts. $T_\Sigma(V) := \bigcup_{s\in S} T_\Sigma(V)_s$ denotes the set of all order-sorted terms over Σ freely generated by V. The set of all ground terms over Σ is $T_\Sigma := T_\Sigma(\{\})$.

A signature is called *regular*, if each term $t \in T_\Sigma(V)$ has a least sort. It is decidable if a signature is regular:

Theorem 2.1 (Smolka et al. [12, 15]) *A signature $(S, \leq, \Sigma)$ is regular iff for every $f \in \Sigma$ and $w \in S^*$ the set $\{s \mid \exists w' \geq w.\ f : (w')s\}$ either is empty or contains a least element.*

As an example of a simple non-regular signature, consider $(\{s_0, s_1, s_2\}, \{s_1 \leq s_0, s_2 \leq s_0\}, \Sigma_{\epsilon,s_1} = \{a\}, \Sigma_{\epsilon,s_2} = \{a\})$: the constant a has two sorts which are incomparable, hence it does not have a minimal sort.

A *substitution* θ from a variable set Y into the term algebra $T_\Sigma(V)$ is a mapping from Y to $T_\Sigma(V)$, which additionally satisfies $\theta(x) \in T_\Sigma(V)_s$ if $x \in V_s$ (that is, substitutions must be sort-correct). As usual, substitutions are extended canonically to $T_\Sigma(V)$. We write $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. If, for $t, t' \in T_\Sigma(V)$, there is a substitution θ such that $t' = \theta(t)$, t' is called an *instance* of t. Similarly, a substitution θ' is called an instance of a substitution θ w.r.t. a set of variables W, written $\theta \succeq \theta'$ $[W]$, if there is a substitution γ such that $\theta'(x) = \gamma(\theta(x))$ for all $x \in W$.

A *unifier* of a set of equations Γ is a substitution θ such that $\theta(s) = \theta(t)$ for all equations $s =^? t$ in Γ. A set of unifiers U of Γ is called *complete* (and denoted by CSU), if for every unifier θ' of Γ there exists $\theta \in U$ such that θ' is an instance of θ w.r.t. the variables in Γ. As usual, a signature is called *unitary (unifying)* if for all equation sets Γ there is a complete set of unifiers containing at most one element; it is called *finitary*, if there is always a finite and complete set of unifiers. For non-regular signatures, unification can be infinitary even if the signature is finite [11]. But we have the following

Theorem 2.2 (Schmidt-Schauß [11]) *In finite and regular signatures, finite sets of equations have finite, complete, and effectively computable sets of unifiers.*

Waldmann [15, Thm 9.5] provides a succinct characterization of unitary signatures. For our purposes the following sufficient conditions are more interesting. Call a signature *downward complete* if any two sorts have either no lower bound or an infimum, and *coregular* if for every f and s the set

$$D(f, s) = \{w \mid \exists s'.\ f : (w)s' \wedge s' \leq s\}$$

either is empty or has a greatest element. Smolka et al. [12] show

Theorem 2.3 *Every finite, regular, coregular, and downward complete signature is unitary.*

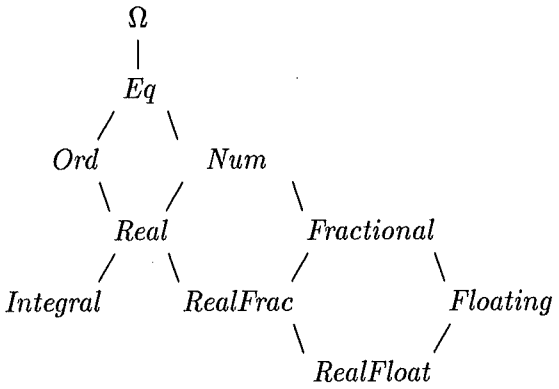
From this theorem it is easy to derive the following specialization:

Corollary 2.4 *Every finite, regular, and downward complete signature is unitary if it is*

- injective*: $f : (w)s$ and $f(w')s$ imply $w = w'$, and*
- subsort reflecting*: $f : (w')s'$ and $s' \leq s$ imply $f : (w)s$ for some $w \geq w'$.*

Injectivity and subsort reflection imply coregularity, but not the other way around. Nevertheless one can show that the two criteria are essentially equivalent [10].

For unitary signatures, order-sorted unification can be implemented in quasi-linear time [8]. For finitary signatures, order-sorted unification in general is NP-complete [11], but in most cases can be implemented efficiently. We do not intend to present algorithms for order-sorted unification; this has been done elsewhere [15, 8, 10]. We merely provide some examples which will be used later. Consider the sort hierarchy[5]



and the operator *list* with multiple declarations

$$\begin{aligned} list:&\ (\Omega)\Omega \\ list:&\ (Eq)Eq \\ list:&\ (Ord)Ord \\ list:&\ (Num)Num \end{aligned}$$

Now we want to unify $x_{Eq} =^?_\Sigma list(y_\Omega)$. Since $list(y_\Omega)$ has sort Ω, we need a substitution θ such that $list(\theta(y))$ has a sort $\leq Eq$, by choosing a suitable arity for *list*. This process is known as *weakening*[6]. Choose $\theta = \{y \mapsto z_{Eq}, x_{Eq} \mapsto list(z_{Eq})\}$, which is sort-correct (because $list : (Eq)Eq$), and constitutes a complete (singleton) set of unifiers. But note what happens if the declaration $list : (Eq)Eq$ is removed from the signature: the above substitution is no longer sort-correct, since $list(z_{Eq})$ has only sort Ω. Instead, the weakening process must go even further down in the sort hierarchy, and we obtain the complete 2-element set of unifiers $\{\{y \mapsto z_{Ord}, x \mapsto list(z_{Ord})\}, \{y \mapsto z_{Num}, x \mapsto list(z_{Num})\}\}$. Now let us unify $x_{Ord} =^?_\Sigma y_{Num}$. We obtain the singleton CSU $\theta = \{x_{Ord} \mapsto z_{Real}, y_{Num} \mapsto z_{Real}\}$. If Ord and Num had no lower bound, these terms would not unify, unless we added a new constant a of sort both Ord and Num: the reader should convince herself that in this case the set $\{\{x \mapsto list^n(a), y \mapsto list^n(a)\} \mid n \in \mathbb{N}_0\}$ is a complete and minimal, yet infinite set of unifiers! This strange behaviour is due to the fact that the signature is not regular: $list(a)$ does not have a minimal sort.

3 The type inference system

Our type inference system for Mini-Haskell replaces ordinary unification in the Hindley-Milner system with order-sorted unification, where the signature of the order-sorted algebra of types is constructed from the class and instance declarations. The inference system described below is a simple extension of the system DM' due to Clément et al. [1], which in turn is a variant of the classical Damas-Milner system [3]. In fact, if we remove class and instance declarations

[5] It is not an accident that this sort hierarchy coincides with the Haskell numeric class hierarchy [6, p. 50].

[6] In our setting, *strengthening* would be more appropriate: the lower we are in the sort hierarchy, the more we know.

TAUT $$\frac{A(x) \succeq_\Sigma \tau}{(A,\Sigma) \vdash x : \tau}$$

APP $$\frac{(A,\Sigma) \vdash e_0 : \tau \to \tau' \qquad (A,\Sigma) \vdash e_1 : \tau}{(A,\Sigma) \vdash (e_0\ e_1) : \tau'}$$

ABS $$\frac{(A + [x \mapsto \tau], \Sigma) \vdash e : \tau'}{(A,\Sigma) \vdash \lambda x.e : \tau \to \tau'}$$

LET $$\frac{(A,\Sigma) \vdash e_0 : \tau \qquad FV(\tau, A) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_k}\} \qquad (A + [x \mapsto \forall\overline{\alpha_{\gamma_k}}.\tau], \Sigma) \vdash e_1 : \tau'}{(A,\Sigma) \vdash \textbf{let } x = e_0 \textbf{ in } e_1 : \tau'}$$

CLASS $$(A,\Sigma) \vdash \textbf{class } \gamma \leq \gamma_1, \ldots, \gamma_n \textbf{ where } x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k : (A + [x_i \mapsto \forall\alpha_\gamma.\tau_i \mid i = 1..k], \Sigma + \{\gamma \leq \gamma_j \mid j = 1..n\})$$

INST $$\frac{A(x_i) = \forall\alpha_\gamma.\tau_i \qquad (A,\Sigma) \vdash e_i : \tau_i[\chi(\overline{\alpha_{\gamma_n}})/\alpha_\gamma] \qquad i = 1..k}{(A,\Sigma) \vdash \textbf{inst } \chi : (\overline{\gamma_n})\gamma \textbf{ where } x_1 = e_1, \ldots, x_k = e_k : (A, \Sigma + \chi : (\overline{\gamma_n})\gamma)}$$

PROG $$\frac{(A_{i-1}, \Sigma_{i-1}) \vdash d_i : (A_i, \Sigma_i) \qquad i = 1..n \qquad (A_n, \Sigma_n) \vdash e : \tau}{(A_0, \Sigma_0) \vdash d_1; \ldots; d_n; e : \tau}$$

Figure 2: Type inference rules

from our syntax, our system reduces to DM': the order-sorted algebra of types is the trivial one with only one sort Ω, order-sorted unification reduces to classical Robinson unification, and the inference rules simplify to the original DM' rules. Overloading resolution is performed by order-sorted unification alone. Since we have already seen that order-sorted signatures allow multiple operator arities, and that the unification algorithm will select the right one(s) which must be used in order to unify two terms, the reader might already get an idea of what we are aiming for.

As usual, inferences in our system depend on type assumptions. A type assumption is a finite mapping from variables to types, just as in the ordinary Damas-Milner algorithm. The only difference is that in our system every type variable has a sort; in case there are no class declarations, all type variables have sort Ω. In addition, our inferences depend on the signature of an order-sorted algebra which can be seen as a compact representation of the class and instance declarations found so far. If there are no class definitions, this signature contains only declarations of the from $\chi : (\Omega^n)\Omega$ for all n-ary type constructors χ. A class declaration will add a new sort and subsort relations to the signature, an instance declaration will add a new arity for the type constructor involved.

The following convention is used: $\overline{\alpha_{\gamma_k}}$ denotes the list $\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_k}$, with the understanding that the α_{γ_i} are distinct type variables.

The first four rules in the type inference system in Figure 2 are of the form $(A,\Sigma) \vdash e : \sigma$ and are almost identical to the DM' rules. There are two differences: all inferences depend on the signature Σ of the type algebra as well as the set of type assumptions A. Furthermore, generic instantiation in rule TAUT must respect Σ. This is written $\sigma \succeq_\Sigma \tau$, meaning that σ has the form $\forall\overline{\alpha_{\gamma_n}}.\tau_0$, there are τ_i of sort γ_i, and $\tau = \tau_0[\tau_1/\alpha_{\gamma_1}, \ldots, \tau_n/\alpha_{\gamma_n}]$. In the rule LET we use the notation $FV(\tau)$, which denotes the set of free type variables in τ; $FV(\tau, A)$ denotes $FV(\tau) - FV(A)$.

If no class and instance declarations are present,

$$\Sigma_0 = (\{\Omega\}, \{\Omega \leq \Omega\}, \{int : \Omega, float : \Omega, char : \Omega, list : (\Omega)\Omega, pair : (\Omega^2)\Omega, \_ \to \_ : (\Omega^2)\Omega\})$$

is the trivial order-sorted signature, and from the facts mentioned above it is clear that we have

Lemma 3.1 *For a Mini-Haskell expression e without class and instance declarations, our system and Damas-Milner compute the same type:*

$$A_0 \vdash_{DM'} e : \tau \;\Leftrightarrow\; (A_0, \Sigma_0) \vdash_{NS} e : \tau$$

where A_0 is any initial set of type assumptions.

The rules CLASS and INST are of course at the heart of our inference system, although they are remarkably simple. Both rules are of the form $(A, \Sigma) \vdash d : (A', \Sigma')$, thus declarations do not have a type, but extend a given set of type assumptions and a given signature.

The declaration **class** $\gamma \leq \gamma_1, \ldots, \gamma_n$ **where** $x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k$ introduces a new class γ, which should not have been declared before. This class is added to Σ by extending the ordering with $\gamma \leq \gamma_i$ for all super-classes γ_i of γ. For all overloaded identifiers x_i introduced in the construct, their generic type scheme $\forall\alpha_\gamma.\tau_i$, where τ_i should be well-formed w.r.t. Σ, is added to the set of type assumptions A. As a consequence, the x_i may be used, for example, in subsequent let-definitions. But note that they cannot be applied to real data, unless an instance of γ is declared: by the definition of generic instantiation, the instance of α must by a type of sort γ. Without instance declarations for γ, the only types of this sort are type variables! It should be pointed out that because the context A associates at most one type with any identifier, different declarations of the same identifier overwrite each other. In particular it means that if there are two classes which share some of their method names x_i, only those of the latter class are visible — there is a limit to the amount of overloading that can be expressed. This is in contrast to some object-oriented languages, but agrees with the Haskell definition and in a canonical way resolves naming problems caused by multiple inheritance.

Typing **inst** $\chi : (\overline{\gamma_n})\gamma$ **where** $x_1 = e_1, \ldots, x_k = e_k$ requires that the x_i have some generic type $A(x_i) = \forall\alpha_\gamma.\tau_i$. Thus, no instance declaration is allowed without a corresponding class declaration. In Section 6 a stronger context condition is imposed: the x_i defined in an instance must be exactly those declared in the enclosing class declaration for γ. The type constructor χ is declared to be in class γ, provided its arguments are in classes $\gamma_1, \ldots, \gamma_n$. Hence the instance declaration extends Σ with the declaration $\chi : (\overline{\gamma_n})\gamma$. Note that w.r.t. the extended Σ there are now type terms of sort γ which are not type variables, hence the x_i can now be applied to values of type χ. The e_i need to have the type obtained by replacing the α_γ in their generic type $\forall\alpha_\gamma.\tau_i$ by $\chi(\overline{\alpha_{\gamma_n}})$. Note that our language does not have explicit recursion. If it did, the e_i could be referring to the very instances of the x_i they define. In that case each e_i would need to be typed in the extended signature as well:

$$(A, \Sigma + \chi : (\overline{\gamma_n})\gamma) \vdash e_i : \tau_i[\chi(\overline{\alpha_{\gamma_n}})/\alpha_\gamma] \qquad (1)$$

Finally, the rule PROG simply states that the type of a program is the same as the type of its constituting expression, provided this expression is typed in the extended signature and type assumptions which have been produced by the declarations. Note that according to this rule a function definition in an instance declaration cannot refer to functions from subsequent classes. This is however allowed in Haskell and would require that all e_i in all instance declarations are typed in the final signature Σ_n; for the sake of readability, we stick to the simpler system.

```
class Eq a where (==) ::  a -> a -> bool
class Eq a => Num a where ...
instance Eq char where (==) = eqChar
instance Eq a => Eq [a] where
   [] == []      =  true
   [] == x:xs    =  false
   x:xs == []    =  false
   x:xs == y:ys  =  (x==y) & (xs == ys)
['a','b','c'] == ['d','e','f']
```

Figure 3: An example program

4 Some examples

Figure 3 shows a small Haskell program (adapted from [14]) in concrete syntax. The translation into Mini-Haskell is straightforward.

The type constants $int : \Omega$ and $char : \Omega$ are assumed to be predefined, as well as *eqInt* and *eqChar*, which have type $int \to int \to bool$ resp. $char \to char \to bool$[7]. Upon analysis of the second instance declaration, the signature contains the sorts Ω, Eq and Num, where $Num < Eq < \Omega$, and *char* has the additional sort Eq. The type assumptions are just $[(==) \mapsto \forall\alpha_{Eq}.\alpha_{Eq} \to \alpha_{Eq} \to bool]$. The second instance declaration causes a new arity of the type constructor *list* to be added to the signature, namely $list : (Eq)Eq$.

Before we look at the type inference for the second instance declaration in detail, we trace the simpler problem `['a','b','c'] == ['d','e','f']`. Since character literals have type *char* in Haskell, both lists have type $list(char)$. The order-sorted nature of the problem becomes important only with the application of "==". According to TAUT, the type of "==" must be an instance of $\alpha_{Eq} \to \alpha_{Eq} \to bool$. According to APP, α_{Eq} must be unified with $list(char)$. Since the current signature contains the declarations $char : Eq$ and $list : (Eq)Eq$, $list(char)$ is of sort Eq and the resulting CSU is the substitution $\{\alpha_{Eq} \mapsto list(char)\}$. A similar unification is needed for the second argument of "==", and we can infer that this particular use of "==" has type $list(char) \to list(char) \to bool$. Note that the actual overloading resolution is performed by the unification algorithm: the arities of type constructors used during unifications determine the instance to be used. In our example the second instance of "==" is identified, since the inferred instance of "=="'s generic type uses the arity of list which was introduced through the second instance declaration. If the second instance declaration were missing, $list(char)$ could not have sort Eq, thus the above unification would fail and the last line would not be typable.

The body of the second instance declaration translates into something like the following Mini-Haskell definition:

```
(==) = fix(λeql.λx.λy.
          if(null x, null y, if(null y, false,
                                hd x == hd y & eql (tl x) (tl y))))
```

The lack of pattern matching and recursion forces us to use the fixpoint combinator `fix`. This simplifies the typing problem, since there is no recursive use of "==". To make this

[7]The standard Haskell prelude defines *char* to be an instance of *Enum* and *int* to be an instance of *Num*, but for reasons of presentation we assume that all type constructors have default sort Ω. Of course, our system can handle the real situation as well.

example more interesting, we ignore the translation and type check the original code using the modified INST rule shown in (1). We must infer that the definition of "==" has type $list(\alpha_{Eq}) \to list(\alpha_{Eq}) \to bool$, which is the specific instance of the generic type of "==" required here. For the first three clauses this is immediate. Now consider the last one. From the use of ":" we can infer that xs and ys have type $list(\alpha_\Omega)$. A similar computation as above will then determine that in the second recursive call "==" has type $list(\alpha_{Eq}) \to list(\alpha_{Eq}) \to bool$. For the first recursive call, the situation is more tricky. Since we do not know anything about the types of x and y, their initial type is α_Ω. Unification with the argument type of the generic type of "==" will weaken this to α_{Eq}. Hence this use of "==" has type $\alpha_{Eq} \to \alpha_{Eq} \to bool$. As this type contains free type variables not of sort Ω, we cannot resolve overloading and do not know which instance of "==" to use for the comparison of list elements; this decision must be postponed until runtime. But we can at least successfully type the definition, because we now know that the arguments are lists with elements of sort Eq, and the result type is $bool$.

If we want to type `[1,2,3]==[4,5,6]`, we have a different situation. In Haskell, integer literals have any numeric type, and type inference will thus infer the type $list(\alpha_{Num})$ for both lists. Since Num is a subsort of Eq, the unification of α_{Eq} with $list(\alpha_{Num})$ is possible, and the expression can be typed. This mirrors the fact that a subclass inherits all functions of its superclasses (thus "==" is automatically defined for integer literals). In the absence of the second class declaration, there is no sort-correct unifier, and the expression is not typable.

Now consider the (illegal!) program

```
class γ1 ...
class γ2 ...
class δ1 a where f ::  a -> bool
class δ1 a => δ2 a where ...
instance γ1 a => δ1 [a] where f = ...
instance γ2 a => δ2 [a] where ...
λx.f [x]
```

Suppose int is a member of γ_2. Hence $list(int)$ is a member of δ_2 and hence of δ_1. According to the type system, `f` is defined on objects of type $list(int)$. But the program does not define its value, since the only `f` that is defined requires its argument to be of type $list(\alpha_{\gamma_1})$! This incoherence is reflected in our system as follows. The expression $\lambda x.f[x]$ has to be typed in the regular signature with sort ordering $\gamma_1 \leq \Omega$, $\gamma_2 \leq \Omega$, $\delta_2 \leq \delta_1$ and declarations $list : (\gamma_1)\delta_1$ and $list : (\gamma_2)\delta_2$, and under type assumption $[f \mapsto \forall\alpha_{\delta_1}.\alpha_{\delta_1} \to bool]$. This leads to the unification problem $list(\alpha_\Omega) =^? \beta_{\delta_1}$ which has the two solutions $\{\alpha_\Omega \mapsto \alpha_{\gamma_1}, \beta_{\delta_1} \mapsto list(\alpha_{\gamma_1})\}$ and $\{\alpha_\Omega \mapsto \alpha_{\gamma_2}, \beta_{\delta_1} \mapsto list(\alpha_{\gamma_2})\}$. These two solutions are incomparable (because γ_1 and γ_2 are), and there is no solution subsuming both of them. Hence $\lambda x.f[x]$ has the two types $\alpha_{\gamma_1} \to bool$ and $\alpha_{\gamma_2} \to bool$. This ambiguity is caused by the second instance declaration which also violates a Haskell context condition [6, p. 27]. Thus the above incoherence reveals itself by the absence of a principal type. The next section introduces restrictions similar to those in Haskell which ensure the existence of principal types. As Section 6 shows, principal types help to ensure semantic well-definedness.

5 Computing principal types

The type inference system gives rise to an algorithm, just by reading the rules backwards. In fact, we have implemented our overloading resolution in Prolog. If an expression has several incomparable types, our Prolog program will compute them one by one upon backtracking. The termination of this algorithm is obvious (provided unifications are finitary), since every rule

decomposes an abstract syntax tree into its components. Instead of guessing the required instantiation in TAUT, it is inferred by order-sorted unification. This raises certain computability questions.

Theorem 2.2 states that regularity implies finiteness and decidability of order-sorted unification. Unfortunately, we have no guarantee that the signature built up during type inference is regular. In fact, in many cases it will not be. However, by going to the powerset of sorts, regularity can be achieved. Furthermore, we show that two semantically motivated context conditions lead to unitary signatures, i.e. principal types exist.

The main idea for regularity is quite trivial: two classes γ_1 and γ_2 can always be combined to form a subclass $\gamma = \gamma_1 \wedge \gamma_2$ of both of them which provides the union of the operations available in each of them. In fact, this can already be done inside the language by writing

$$\textbf{class } \gamma \leq \gamma_1, \gamma_2 \textbf{ where } ;$$

The idea of conjunctive sorts[8] can be integrated into the framework by defining a new set of sorts $\hat{S}$, the set of all non-empty sets of incomparable class names, and imposing the following ordering on them:

$$S_1 \preceq S_2 \;\Leftrightarrow\; \forall s_2 \in S_2\ \exists s_1 \in S_1.\ s_1 \leq s_2$$

This gives rise to a lower semi-lattice $(\hat{S}, \preceq)$, the free lower semi-lattice on the poset $(S, \leq)$, where $S_1 \wedge S_2 = S_1 \cup S_2$ if S_1 and S_2 are incomparable. In Haskell this extension is expressed by multiple class assertions for type variables.

Conjunctive sorts alone do not guarantee regularity. It is also necessary to realize that the behaviour of type constructors on $\gamma_1 \wedge \gamma_2$ is determined by their behaviour on γ_1 and γ_2. More precisely, we can add a closure condition to signatures:

$$\frac{\chi : (\overline{\gamma_n})\gamma \qquad \chi : (\overline{\delta_n})\delta}{\chi : (\overline{\gamma_n \wedge \delta_n})\gamma \wedge \delta} \tag{2}$$

which tells us that type constructors are homomorphisms w.r.t. conjunction. Again, there is a corresponding construction inside the language. Given

$$\textbf{inst } \chi : (\gamma_1)\gamma \textbf{ where } x = e;$$
$$\textbf{inst } \chi : (\delta_1)\delta \textbf{ where } x' = e';$$

we can add $\textbf{inst } \chi : (\gamma_1 \wedge \delta_1)\gamma \wedge \delta \textbf{ where } ;$.

Conjunctive sorts together with the homomorphism condition (2) guarantee regularity: either the set defined in Theorem 2.1 is empty, or its least element is the conjunction of all its elements.

For reasons of space we do not present the type inference rules in terms of conjunctive sorts. Instead we assume that the user or the system provide the required additional class and instance declarations shown above.

Regularity alone does not ensure the existence of principal types, as we have seen in the last example of the previous section. According to Theorem 2.2, it at least guarantees finite complete sets of types, a familiar picture for languages with classes and inheritance [16]. The ambiguity in that example is ruled out by subsort reflection as defined in Section 2. It follows from Corollary 2.4 that the extended system with conjunctive sorts has principal types if injectivity and subsort reflection are enforced. In Haskell this is indeed the case. Section 4.3.2 of the Haskell Report states that "A type may not be declared as an instance of a particular class more than once in the same scope", which is equivalent to injectivity. The same section, at the bottom of page 27, introduces a context condition which amounts to the following: if $\gamma_1, \ldots, \gamma_m$ are

[8] By analogy with conjunctive *types*, first explored by Coppo et al. [2].

the immediate super-classes of δ, a declaration **inst** $\chi : (\overline{\delta_n})\delta$ must be preceded by declarations **inst** $\chi : (\overline{\gamma_n^i})\gamma_i$ such that δ_j is a subclass of γ_j^i for all $i = 1 \ldots m$ and $j = 1 \ldots n$. Any signature built up from such a sequence of declarations is easily seen to be subsort reflecting. It follows that imposing these two conditions on Mini-Haskell guarantees the existence of principal types. However, we will see in Section 6 that principal types do not preclude semantic ambiguity.

6 Translation

So far nothing has been said about the semantics of our language. This will be given by a translation into a well-understood sub-language consisting just of identifiers, abstraction, application and **let** — essentially Mini-ML [1]. Type classes and instances are eliminated in favour of so called *(method) dictionaries* which contain all the functions associated with a class. The scheme presented below generalizes ideas from [14], and the reader should consult this article for intuition and examples.

The formal definition of the translation in terms of inference rules is given in Figure 4. For an expression e, the judgement $(A, \Sigma) \vdash e : \tau \rightsquigarrow e'$ should be pronounced "in the context (A, Σ) e has type τ and translates to e'". Declarations produce a **let**-expression without body, which introduces dictionaries and access functions. Thus, in the notation $(A, \Sigma) \vdash d : (A', \Sigma') \rightsquigarrow d'$, d' is of the form **let** $x_1 = e_1, ..., x_j = e_j$.

The effect of the translation can best be explained in terms of types. In the following we use the phrase "ML-type" to distinguish the translated types from the original ones which may contain sorted type variables. Below we show how the declaration of a class γ gives rise to an ML-type $\gamma(\alpha)$ of γ-dictionaries, where α is the instance type. Assuming $\gamma(\alpha)$ we define

$$ML(\forall \overline{\alpha_{\gamma_k}}.\tau) \;=\; \forall \overline{\alpha_k}.\gamma_1(\alpha_1) \to \cdots \to \gamma_k(\alpha_k) \to \tau[\alpha_1/\alpha_{\gamma_1}, \ldots, \alpha_k/\alpha_{\gamma_k}]$$

which translates order-sorted types to ML-types: type restrictions are turned into additional dictionary arguments. This means that the translation of a function definition has to provide abstractions for these new arguments and the application has to provide corresponding dictionary arguments.

The target language has the following special features: it contains all n-ary product types $\alpha_1 * \cdots * \alpha_n$, with values $(a_1, \ldots, a_n)$ and projection functions $\pi_i^n : \alpha_1 * \cdots * \alpha_n \to \alpha_i$. In addition to ordinary identifiers (x) the translation introduces α_γ (parameter representing γ-dictionaries), γ_χ (γ-dictionary for type χ), and γ_δ (function extracting the γ-dictionary from a δ-dictionary).

We go through the rules one by one.

Conceptually, a class declaration **class** $\gamma \leq \gamma_1, \ldots, \gamma_n$ **where** $x_1 : \forall\alpha_\gamma.\tau_1, \ldots, x_k : \forall\alpha_\gamma.\tau_k$ introduces a new dictionary of ML-type

$$\gamma(\alpha) = \tau_1[\alpha/\alpha_\gamma] * \cdots * \tau_k[\alpha/\alpha_\gamma] * \gamma_1(\alpha) * \cdots * \gamma_n(\alpha) \tag{3}$$

The type parameter α stands for the type of the instance. The first k components of $\gamma(\alpha)$ are the functions added in the declaration of class γ. The next n components are the dictionaries for all immediate super-classes of γ. Note that $\Omega(\alpha)$ is the empty product type.

Instead of defining this ML-type explicitly, the translation just defines the relevant access functions: x_i accesses the i-th component and hence the instance of function x_i; γ_{j_γ} accesses the super-dictionary corresponding to the super-class γ_j, i.e. γ_{j_γ} takes a γ dictionary and returns a γ_j dictionary.

An instance declaration **inst** $\chi : (\overline{\gamma_n})\gamma$ **where** $x_1 = e_1, \ldots, x_k = e_k$ introduces γ_χ of ML-type

$$\forall \overline{\alpha_n}.\gamma_1(\alpha_1) \to \cdots \to \gamma_n(\alpha_n) \to \gamma(\chi(\overline{\alpha_n})).$$

TAUT
$$\frac{A(x) = \forall\overline{\alpha_{\gamma_k}}.\tau}{(A,\Sigma) \vdash x : \tau[\tau_1/\alpha_{\gamma_1},\ldots,\tau_k/\alpha_{\gamma_k}] \rightsquigarrow (x\ \mathsf{dict}_\Sigma(\tau_1,\gamma_1)\ \ldots\ \mathsf{dict}_\Sigma(\tau_k,\gamma_k))}$$

APP
$$\frac{(A,\Sigma) \vdash e_0 : \tau \to \tau' \rightsquigarrow e_0' \qquad (A,\Sigma) \vdash e_1 : \tau \rightsquigarrow e_1'}{(A,\Sigma) \vdash (e_0\ e_1) : \tau' \rightsquigarrow (e_0'\ e_1')}$$

ABS
$$\frac{(A + [x \mapsto \tau],\Sigma) \vdash e : \tau' \rightsquigarrow e'}{(A,\Sigma) \vdash \lambda x.e : \tau \to \tau' \rightsquigarrow \lambda x.e'}$$

LET
$$\frac{(A,\Sigma) \vdash e_0 : \tau \rightsquigarrow e_0' \quad FV(\tau,A) = \{\overline{\alpha_{\gamma_k}}\} \quad (A + [x \mapsto \forall\overline{\alpha_{\gamma_k}}.\tau],\Sigma) \vdash e_1 : \tau' \rightsquigarrow e_1'}{(A,\Sigma) \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 : \tau' \rightsquigarrow \mathbf{let}\ x = \lambda\overline{\alpha_{\gamma_k}}.e_0'\ \mathbf{in}\ e_1'}$$

CLASS
$$\begin{aligned}&(A,\Sigma) \vdash \mathbf{class}\ \gamma \le \gamma_1,\ldots,\gamma_n\ \mathbf{where}\ x_1 : \forall\alpha_\gamma.\tau_1,\ldots,x_k : \forall\alpha_\gamma.\tau_k :\\ &\qquad (A + [x_i \mapsto \forall\alpha_\gamma.\tau_i \mid i = 1..k],\Sigma + \{\gamma \le \gamma_j \mid j = 1..n\})\\ &\qquad \rightsquigarrow \mathbf{let} x_1 = \pi_1^{k+n},\ldots,x_k = \pi_k^{k+n},\ \gamma_{1\gamma} = \pi_{k+1}^{k+n},\ldots,\gamma_{n\gamma} = \pi_{k+n}^{k+n}\end{aligned}$$

INST
$$\frac{\begin{array}{c}\mathsf{super}_\Sigma(\gamma) = \{\gamma^1,\ldots,\gamma^s\}\\ A(x_i) = \forall\alpha_\gamma.\tau_i \qquad (A,\Sigma) \vdash e_i : \tau_i[\chi(\overline{\alpha_{\gamma_n}})/\alpha_\gamma] \rightsquigarrow e_i' \qquad i = 1..k\end{array}}{\begin{array}{l}(A,\Sigma) \vdash \mathbf{inst}\ \chi : (\overline{\gamma_n})\gamma\ \mathbf{where}\ x_1 = e_1,\ldots,x_k = e_k : (A,\Sigma + \chi : (\overline{\gamma_n})\gamma)\\ \quad \rightsquigarrow \mathbf{let}\ \gamma_\chi = \lambda\overline{\alpha_{\gamma_n}}.\,(e_1',\ldots,e_k',\\ \qquad\qquad (\gamma_\chi^1\ \mathsf{cast}_\Sigma(\alpha_{\gamma_1},\gamma_1^1)\ \ldots\ \mathsf{cast}_\Sigma(\alpha_{\gamma_n},\gamma_n^1)),\\ \qquad\qquad \vdots\\ \qquad\qquad (\gamma_\chi^s\ \mathsf{cast}_\Sigma(\alpha_{\gamma_1},\gamma_1^s)\ \ldots\ \mathsf{cast}_\Sigma(\alpha_{\gamma_n},\gamma_n^s)))\end{array}}$$

PROG
$$\frac{(A_{i-1},\Sigma_{i-1}) \vdash d_i : (A_i,\Sigma_i) \rightsquigarrow d_i' \quad i = 1..n \qquad (A_n,\Sigma_n) \vdash e : \tau \rightsquigarrow e'}{(A_0,\Sigma_0) \vdash d_1;\ldots;d_n;e : \tau \rightsquigarrow d_1'\ \mathbf{in}\ldots\mathbf{in}\ d_n'\ \mathbf{in}\ e'}$$

Figure 4: Translation of expressions

Given the required dictionaries for the arguments $\overline{\alpha_n}$ of χ, γ_χ produces a dictionary of ML-type $\gamma(\chi(\overline{\alpha_n}))$. The λ-bound variables $\overline{\alpha_{\gamma_n}}$ represent the argument dictionaries. In case χ is a constant like *int* or *char*, γ_χ is a dictionary of ML-type $\gamma(\chi)$.

The first k components of the result type $\gamma(\chi(\overline{\alpha_n}))$ are the translated expressions defining the methods x_1 to x_k. Since e_i has type $\tau_i[\chi(\overline{\alpha_{\gamma_n}})/\alpha_\gamma]$, the expression e_i' depends on n dictionaries of ML-type $\gamma_j(\alpha_j)$, $j = 1..n$. By some mechanism that is explained in connection with the translation of identifiers, this implies that e_i' contains free variables $\overline{\alpha_{\gamma_n}}$ waiting to be instantiated with dictionaries.

The last s components are the dictionaries of the immediate super-classes of γ:

$$\mathsf{super}_\Sigma(\gamma) \;=\; \{\gamma' \mid \gamma < \gamma' \wedge \nexists\delta.\ \gamma < \delta < \gamma'\}$$

At this point we assume that the current expression is in the scope of χ instance declarations for all γ^1 to γ^s (context condition 3 below). Hence the dictionary generators γ_χ^j have been defined. However, they may not expect dictionaries for the classes $\overline{\gamma_n}$, but for some super-classes thereof. Therefore the relevant super-dictionaries need to be extracted from the $\overline{\alpha_{\gamma_n}}$: if $\gamma \le_\Sigma \gamma'$, $\mathsf{cast}_\Sigma(\alpha_\gamma,\gamma')$ expands into an expression which does just that by inserting the appropriate sequence of coercers which have been defined by the translation of the class declarations for all

the classes between γ and γ'.

$$\mathsf{cast}_\Sigma(\alpha_\gamma, \gamma') = \begin{cases} \alpha_\gamma & \text{if } \gamma = \gamma' \\ (\gamma'_\delta\ \mathsf{cast}_\Sigma(\alpha_\gamma, \delta)) & \text{if } \gamma \leq \delta \wedge \gamma' \in \mathsf{super}_\Sigma(\delta) \end{cases}$$

If there is more than one path from γ to γ' w.r.t. $\leq_\Sigma$, cast_Σ chooses an arbitrary fixed one.

Notice that the positional scheme used for arranging the methods and the super-dictionaries in a dictionary requires some fixed ordering on both the x_i and the γ_j, e.g. lexicographic. Otherwise it is not clear in which order the x_i and γ_j appear: the signature Σ does not record this information.

The translation of an identifier in rule TAUT is determined by its use. If x has type $\forall\overline{\alpha_{\gamma_k}}.\tau$, its definition has translated into an function of ML-type $ML(\forall\overline{\alpha_{\gamma_k}}.\tau)$ which requires k dictionaries. Exactly which dictionaries are passed with this call to x is determined by the τ_i. Expanding $dict_\Sigma(\tau, \gamma)$ produces the code representing the dictionary of ML-type $\gamma(\tau)$ as defined in (3).

$$\begin{aligned} \mathsf{dict}_\Sigma(\alpha_{\gamma'}, \gamma) &= \mathsf{cast}_\Sigma(\alpha_{\gamma'}, \gamma) \\ \mathsf{dict}_\Sigma(\chi(\tau_1, \ldots, \tau_n), \gamma) &= \gamma_\chi\ (\mathsf{dict}_\Sigma(\tau_1, \gamma_1))\ \ldots\ (\mathsf{dict}_\Sigma(\tau_n, \gamma_n)) \quad \text{where } \chi \in \Sigma_{\tau_1 \ldots \tau_n, \tau} \end{aligned}$$

It is a homomorphism which maps type constructors to their corresponding dictionary generators declared in the translation of the respective instance declarations. Type variables map to dictionary variables (with suitable coercers inserted by **cast**). In case x is a subexpression of e_0 in LET or of some e_i in INST, these type variables are λ-abstracted later on. Otherwise the represent a semantic ambiguity as in Example 6.1 below.

The rules APP and ABS do not need any explanation. Note that λ-bound identifiers cannot be polymorphic. Thus for such identifiers $k = 0$ in TAUT, and the identifier translates into itself.

The correctness of our translation depends on a number of context conditions:

1. No type constructor can be declared as an instance of a particular class more than once in the same scope. This is ruled out in Haskell as well [6, p. 29].
2. Each instance declaration lists exactly those x_i that the corresponding class declaration listed (and in the same order, which is simply a technical device). This means that operations of super-classes cannot be redefined in subclasses. A more sophisticated translation scheme can easily avoid this restriction.
3. Every instance declaration **inst** $\chi : (\overline{\gamma_n})\gamma$, where $\mathsf{super}_\Sigma(\gamma) = \{\gamma^1, \ldots, \gamma^s\}$ at that point, has to be nested inside instance declarations **inst** $\chi : (\overline{\gamma_n^i})\gamma^i$ for all $i = 1..s$. In particular $\overline{\gamma_n} \leq_\Sigma \overline{\gamma_n^i}$ must hold. It can be shown that this is equivalent to the context condition in Section 4.3.2 of the Haskell Report and implies subsort reflection and hence the existence of principal types.

If these conditions are violated, the translated expression may not be well-formed. Hence the original expression has no semantic meaning. But even if an expression conforms to the context conditions, can be typed and translated, it may still be regarded as ambiguous:

Example 6.1 Let $\Sigma = (\{\Omega, Eq\}, \{Eq < \Omega\}, \{list : (\Omega)\Omega, list : (Eq)Eq\})$ and $A = \{[] \mapsto \forall\alpha.list(\alpha), (==) \mapsto \forall\alpha_{Eq}.\alpha_{Eq} \to \alpha_{Eq} \to bool\}$, and $e = [] == []$. Because the element type of [] is undetermined, it is not clear which Eq-dictionary should be passed to ==, and hence e is ambiguous. Nevertheless, we have $(A, \Sigma) \vdash e : bool \rightsquigarrow$ (==) α_{Eq} [] [] where the free variable α_{Eq} signifies an undetermined dictionary.

Haskell deals with this problem by inferring the type `Eq a => bool` for e, concluding that e is ambiguous because the class assumption `Eq a` contains a type variable `a` not present in the type `bool` [6, p. 30].

In our approach the ambiguity reveals itself by looking at the translation which contains the free variable α_{Eq} not present in *bool*.

The final theorem shows that the inferred types and the semantics given by the translation fit together. Let $ML(A) = \{x \mapsto ML(\sigma) \mid A(x) = \sigma\}$

Theorem 6.2 *Let all types in the range of A be closed. If $(A, \Sigma) \vdash_{NS} e : \tau \rightsquigarrow e'$ and $FV(\tau) = FV(e') - FV(e) = \{\alpha_{\gamma_1}, \ldots, \alpha_{\gamma_n}\}$, then $ML(A) \vdash_{DM} \lambda\overline{\alpha_{\gamma_n}}.e' : ML(\forall\overline{\alpha_{\gamma_n}}.\tau)$.*

Roughly speaking, this says that the if e has type τ, the translation e' of e has the translated type $ML(\tau)$. The implication is trivial if $FV(\tau) \neq FV(e') - FV(e)$. This is precisely the case if e' contains a free dictionary variable α_γ not free in τ, which in turn corresponds exactly to the Haskell ambiguity definition quoted in Example 6.1 above. Hence we know that the type of unambiguous expressions matches their semantics.

7 Conclusion

We have presented a type inference system and algorithm for a Haskell-like language, which is based on order-sorted unification. In contrast to the type system of Wadler and Blott [14] and a similar system by Kaes [7][9], our system is the first one to utilize full order-sorted unification. It grew out of efforts to integrate polymorphism into a generic theorem prover [10] which required more control over the instantiation of type variables than is available in ML. A weaker form of order-sorted unification is also used in the concept of context relations [13] which is a generic and incremental type inference mechanism and can be used for the resolution of user-defined overloading as for example in ADA [5]. But context relations do not allow user-defined classes: the signature must be fixed at language definition time.

Acknowledgements The first author would like to thank Uwe Waldmann for e-mail consultations and Eugenio Moggi and Larry Paulson for detailed comments. The second author would like to thank Stefan Kaes for stimulating discussions. We are grateful to Mark Lillibridge and Stephen Blott who pointed out some subtle errors in a previous version of this paper.

References

[1] 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.

[2] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In R. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms*. Academic Press, 1980.

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

[4] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. Principles of Programming Languages*, pages 52–66, 1985.

[9] Kaes' language does not have explicit type classes; such classes arise only implicitly.

[5] F. Grosch and G. Snelting. Inference-based overloading resolution for ADA. In *Proc. 2nd Conf. Programming Language Implementation and Logic Programming*, pages 30–44. LNCS 456, 1990.

[6] P. Hudak and P. Wadler. Report on the programming language Haskell. Version 1.0, April 1990.

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

[8] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, 8:383–413, 1989.

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

[10] T. Nipkow. Higher-order unification, polymorphism, and subsorts. In *Proc. 2nd Int. Workshop Conditional and Typed Rewriting Systems.* LNCS ???, 1991.

[11] M. Schmidt-Schauß. A many-sorted calculus with polymorphic functions based on resolution and paramodulation. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 1162–1168, 1985.

[12] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2*, pages 297–367. Academic Press, 1989.

[13] G. Snelting. The calculus of context relations. *Acta Informatica*, 1991. To appear.

[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.

[15] U. Waldmann. Unification in order-sorted signatures. Technical Report 298, Fachbereich Informatik, Universität Dortmund, 1989.

[16] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 4th IEEE Symp. Logic in Computer Science*, pages 92–97, 1989.