Type Inference for Overloading without Restrictions, Declarations or Annotations

Carlos Camarão[1] and Lucília Figueiredo[2]

[1] Universidade Federal de Minas Gerais, DCC-ICEX
Belo Horizonte 31270-010, Brasil
[2] Universidade Federal de Ouro Preto, DECOM-ICEB
Ouro Preto 35400-000, Brasil

Abstract. This article presents a type system based on the Damas-Milner system[DM82], that supports overloading. Types of overloaded symbols are constrained polymorphic types. The work is related to Haskell type classes[Wad89,NP93,HHJW96], System O[OWW95] and other similar type systems[Kae88,Smi91,Jon94,DCO96]. Restrictions imposed in these systems with respect to overloading are eliminated. User-defined global and local overloading is supported without restrictions. There is no need for declarations or annotations of any sort. No language construct is added in order to cope with overloading. The type system uses a context-dependent overloading policy, specified by a predicate used in a single inference rule. Overloading of functions defined over different type constructors is supported, as done with Haskell's constructor classes. No monomorphism restriction is required in order to solve ambiguity problems. The system uses an open-world approach, in which new overloaded definitions can be introduced with types automatically reflecting the new definitions. The article also presents a type inference algorithm for the system, which is proved to be sound and to compute principal typings.

1 Introduction

The problems with the treatment of overloading in languages that provide (parametric) polymorphism and type inference, such as SML[MTH89,Pau96] and Miranda[Tur85], have been discussed for example by Wadler and Blott[Wad89]. For instance, `square x = x * x` cannot be written in SML, the reason being that `*` is overloaded for integers and reals. Equality is treated in a special way in SML. For example, the type of function `member`, that tests membership in a list, namely `''a list -> ''a -> bool`, involves a special type variable `''a`, constrained so that its instances must admit equality. In Miranda, this type is not constrained in this way; applying `member` to lists whose elements are functions generates a run-time error.

In Haskell[Pe97,Tho96], type classes[NP93,HHJW96] allow overloaded symbols to have different meanings in contexts requiring different types. Type classes represented a big step forward, in the direction of providing comprehensive support for overloading, together with polymorphism and type inference. However, the following points can be viewed as disadvantages of type classes:

1. Class declarations, as well as type annotations and type declarations, are not needed for the inference of types of overloaded symbols, as is shown in this paper. Thus, a programmer should not be obliged to make a class declaration or type annotation in order to specify the type of a given symbol (he may still write type annotations for documentation and other purposes).
2. The declaration of type classes involves an issue which is separate from overloading resolution: to group logically related names in the same construct. As pointed out by Odersky, Wadler and Wehr[OWW95], "eliminating class declarations means one needs no longer decide in advance which operations belong together in a class. In many situations, this will be a positive advantage. For instance, if we're dealing with pairs, we want only `first` and `second` grouped together, but if we're dealing with triples, we will want `third` as well. As a further example, consider the difficulties that the Haskell designers had deciding how to group numeric operators into classes. This design is still argued: should `+` and `*` be in a 'ring' class?". Grouping logically related symbols together in a language construct involves the definition of a structure of program interfaces; this is intrinsic to modular software development, but overloading resolution is a separate issue.
3. For unrelated definitions of the same name in a program, the programmer is forced to create a class declaration with the type of the overloaded name.
4. For any new definition of any given overloaded name, either its type must be an instance of the type specified in its class declaration, or the class declaration must be modified.
5. The use of classes in type annotations conflicts with data abstraction. If an implementation of (say) x is changed so that an overloaded symbol is no longer used, or a new overloaded symbol is used, the types of all symbols defined by using x may have to be modified.

With respect to specific approaches in implementations[J$^+$98,Jon98a]:

6. Definitions of overloaded symbols must occur in global instance declarations, and overloading is thus restricted to definitions that occur at the outermost level. Local overloading (from bindings occurring inside lambda-abstractions) is not allowed. Local overloading allows a programmer to make definitions where appropriate; he is not forced to make global declarations just because symbols being defined happen to be overloaded.
   Local overloading also allows (as an option for a language designer) the use of symbols defined in an outer level, despite of redefinition in an inner level (with a distinct type).
7. Haskell makes a few restrictions on the form of class and instance declarations, certainly with good reasons, a discussion of these being outside the scope of this paper (see [Jon95,JJM97,Jon98b,Jon94]).

System O[OWW95] is an improvement in relation to type classes with respect to items (1) to (5) above.[1] System O uses universally quantified types, with a constraint on the quantified variable that is a (possibly empty) set of bindings $o :: \alpha \rightarrow \tau$, indicating that there must exist an overloaded operator $o :: p \rightarrow (\tau[p/\alpha])$, for any instance p of α.[2] Although System O was an important step towards the solution to the problem of providing type inference for a language that supports polymorphism and overloading without the need for class declarations, it did not provide a satisfactory solution. Significant limitations of System O are as follows. It supports only context-independent overloading. This means that constants cannot be overloaded, nor function symbols with types like, for example, $\texttt{read} : \texttt{String} \rightarrow \texttt{a}$, even though they could be used in contexts that resolve the overloading (e.g. `read str == "foo"`). The type systems presented in [Kae88,Smi91,DCO96] also support only context-independent overloading. System O requires explicit annotation of the type of each overloaded function symbol. Thus, there is no type inference for overloaded symbols. System O requires all type constructors of the arguments of overloaded functions to be pairwise distinct. This restricts even more the set of types of definitions of an overloaded symbol (this restriction is also imposed in the earlier work of Kaes[Kae88]). System O only allows definitions of overloaded symbols at the outermost level (this restriction is also imposed in the works of Kaes[Kae88], Geoffrey Smith[Smi91] and Mark Jones[Jon94]).

2 Overview of System CT

The type system presented in this article, called System CT, allows multiple occurrences of overloaded symbols in a typing context, each occurrence with a distinct type. Typings for overloaded symbols are introduced in the context after corresponding let-bindings for these symbols (which correspond to instance declarations in Haskell, and `inst` declarations in System O). The overloading policy determines whether or not a given type may be the type of an overloaded symbol in a given context.

System CT uses context-dependent overloading. Overloading of constants is allowed. Consider $\texttt{let one} = \texttt{1 in let one} = \texttt{1.0 in} \ldots$, for example (we assume that literals `0, 1` etc. have type `Int` and literals `o0.0, 1.0` etc. have type `Float`). After the let-bindings, the typing context, say Γ_o, contains the typings $\texttt{one} : \texttt{Int}$ and $\texttt{one} : \texttt{Float}$. The type derived for `one` in context Γ_o is $\{\texttt{one} : \alpha\}.\alpha$, indicating a type for which there is a constant `one` of that type, in Γ_o.

All other type systems use a different approach, as far as we know. The type of a given overloaded symbol is either selected among the set of typings for that symbol in the typing context (e.g. [Kae88,Smi91,OWW95,DCO96]) or, for each overloaded symbol, a fixed typing is used, together with a set of predicates (i.e. a

[1] Although System O does not require class and instance declarations, type annotations are required in the definition of overloaded symbols.

[2] The notation $\sigma[\tau/\alpha]$ indicates the usual textual substitution.

set of available type instances) and a constraint elimination rule, for transforming constrained into unconstrained types (e.g. [NP93,Jon94,HHJW96]). With overloading and this context independent constraint elimination, some difficulties arise in detecting ambiguity. A simple example is expression $\mathtt{g\ one}$ in typing context Γ_{g} with typings $\mathtt{one}:\mathtt{Int}, \mathtt{one}:\mathtt{Float}, \mathtt{g}:\mathtt{Int}\rightarrow\mathtt{Int}, \mathtt{g}:\mathtt{Float}\rightarrow\mathtt{Int}$. The ambiguity in this expression would not be detected without additional rules in the type system, as both $\Gamma_{\mathrm{g}}\vdash\mathtt{one}:\mathtt{Int}$ and $\Gamma_{\mathrm{g}}\vdash\mathtt{g}:\mathtt{Int}\rightarrow\mathtt{Int}$ could be derived, obtaining $\Gamma_{\mathrm{g}}\vdash\mathtt{g\,one}:\mathtt{Int}$, or $\Gamma_{\mathrm{g}}\vdash\mathtt{one}:\mathtt{Float}$ and $\Gamma_{\mathrm{g}}\vdash\mathtt{g}:\mathtt{Float}\rightarrow\mathtt{Int}$, obtaining again $\Gamma_{\mathrm{g}}\vdash\mathtt{g\,one}:\mathtt{Int}$. This is a known problem, of *coherence*[Jon94]. To solve it, restrictions are imposed in order to detect ambiguity.

For example, the type systems of Haskell and Hugs detect this error by means of the following restriction on the syntax of type expressions (called Restriction A for further reference): "If $P\Rightarrow\tau$ is a type, then $tv(P)\subseteq tv(\tau)$", where P specifies a set of constraints, τ is a simple type and tv gives the set of free type variables of its argument. Note that Restriction A must not consider in P type variables that are free in the typing context, so that, for example, $\mathtt{g\,x}$ is not considered ambiguous, in a context as Γ_{g} above but where $\mathtt{x}$ is a lambda-bound variable. In such a context, $\mathtt{g\,x}$ is well typed. Unfortunately, Restriction A prevents the use of expressions which are unambiguous. Consider for example $\Gamma_{\mathrm{f}}=\{\mathtt{f}:\mathtt{Int}\rightarrow\mathtt{Float}, \mathtt{f}:\mathtt{Float}\rightarrow\mathtt{Int}, \mathtt{one}:\mathtt{Int}, \mathtt{one}:\mathtt{Float}\}$. Expression $\mathtt{f\,one}+\mathtt{1}$ (in a context such as Γ_{f}) is a simple example of an unambiguous expression that cannot be written in Haskell, nor in Hugs[Jon98a]. Consider:

```
class F a b where f::a -> b
class O a where one::a
instance F Int Float where ...
instance F Float Int where ...
instance O Int where ...
instance O Float where ...
main = print (f  one + (1::Int))
```

In Hugs and GHC 3.02[J$^+$98], which allow multiple parameter type classes, type `(F a b, O a) => b` (which would be the type of `f one`) is considered ambiguous (characterizing an overloading that cannot be resolved in any context).

Letting `h` be defined by `h x = True`, another example is `h one`, which has type `Bool` in System CT and is considered ambiguous in Haskell or Hugs.

System CT adopts a novel approach in the basic rule, (VAR), which gives the type of an overloaded symbol as the least common generalisation of the types of this symbol in a typing context. As a natural consequence, instantiation and constraint elimination are controlled in rule (APPL), by the use of unification.

For example, in Γ_{o} above (with typings $\mathtt{one}:\mathtt{Int}$ and $\mathtt{one}:\mathtt{Float}$), type $\{\mathtt{one}:\alpha\}.\,\alpha$ is derived for `one`; `Int` and `Float` are considered as instances of this type, due to the possible substitutions for unifying α with types of `one` in Γ_{o}.

In Γ_{f}, typing $\mathtt{f\,one}:\{\mathtt{f}:\alpha\rightarrow\beta, \mathtt{one}:\alpha\}.\,\beta$ indicates that this expression works like an overloaded constant; it can be used in a context requiring either a value of type `Int` or `Float`. Again, this is due to the possible substitutions for unifying types in $\{\mathtt{f} : \alpha \to \beta, \mathtt{one} : \alpha\}$ with typings for `f` and `one` in $\Gamma_{\mathtt{f}}$.

In System CT, the type of any overloaded symbol is computed automatically from the types given by definitions of this symbol that are "visible in the relevant scope" (i.e. that occur in the relevant typing context). The programmer need not anticipate all possible definitions an overloaded symbol might have, something which is needed in order to specify a (least common generalisation) type T for this symbol in a class declaration (so that all definitions of this symbol have types that are instances of T). In System CT, a new definition is not necessarily an instance of the least common generalisation of types given by previous definitions. Instead, any new definition may imply the assignment of a more general type than that computed according to previous definitions (as if a class declaration, if it existed, was automatically modified to reflect the new definition). In this sense, System CT uses in fact an approach that is "more open" than that used with type classes in Haskell.

System CT uses the predicate *ambiguous* for detection of ambiguous overloading (see section 3). With the use of this predicate, expressions with the following types are not considered ambiguous:

1. Types for which $tv(P) \cap tv(\tau) \neq \emptyset$, despite the fact that $tv(P) \not\subseteq tv(\tau)$. This is the case of example `f one` above. Another example is `map f` $\circ$ `map g`, where the standard `map` function is overloaded to operate, for example, on lists and trees, and $\circ$ is the function composition operator.
2. Types for which P has type variables that occur free in the typing context, despite the fact that $tv(P) \cap tv(\tau) = \emptyset$. This is the case of `g x` above (4).

Constraints that apply to the type of $e\ e'$ are only those that have type variables that occur free in the typing context (e.g. `g x`), or occur in the "unconstrained part" of the type of $e\ e'$ (e.g. `snd (True, one)`). In examples for which a component of a value is selected, constraints may not apply to the type of the selected component. For example, `fst (True, one)` where `one` is overloaded and `fst` selects the first component of a pair. In System CT, the type of `fst (True, one)` is `Bool`; it is considered ambiguous in Haskell or Hugs.

Another related example (taken from the Haskell mailing list) follows. Suppose that function `read` is overloaded, having a type that would be written as $\forall\alpha.\ \{\mathtt{read} : \mathtt{String} \to (\alpha, \mathtt{String})\}.\ \mathtt{String} \to (\alpha, \mathtt{String})$ in System CT, and consider: `read2 s = let {(a, s1) = read s; (b, s2) = read s1} in (a, b)`. In System CT, the type of `s1` is `String`, irrespective of whether functions `fst` and `snd` are used in this example instead of pattern-matching.

System CT supports global and local overloading unrestrictively, without the need for declarations or annotations. Overloaded definitions can occur inside lambda-abstractions, and can use lambda-bound variables. No monomorphism restriction[Pe97,Jon94] is necessary in order to solve ambiguity problems.

System CT also supports overloading of functions defined over different type constructors, as done with Haskell's constructor classes[Jon95]. In System CT this issue has been solely a matter of giving a more informative type, since the original system, without constructor variables, also supported overloading of functions defined over different type constructors. The modifications required in the original system, in order to give such more informative types for overloaded symbols defined over different type constructors, were very simple.

For example, System CT allows the writing of reusable code that works for a variety of essentially similar bulk types (see [Jon96]), in a simple way. Assuming `singleton` and `union` to be overloaded to operate on some bulk types (for example `List` and `Queue`), we can write:

```
leaves (Leaf a) = singleton a
leaves (Branch t1 t2) = leaves t1 ‘union‘ leaves t2
```

Type $\{\mathtt{singleton}: \mathtt{a} \to \mathtt{c\,a}, \mathtt{union}: \mathtt{c\,a} \to \mathtt{c\,a} \to \mathtt{c\,a}\}. \mathtt{Tree\,a} \to \mathtt{c\,a}$ is infered for `leaves`. Due to being defined in terms of overloaded functions, function `leaves` works itself just like an overloaded function. Its constrained polymoprhic type reflects this fact. In the type of `leaves`, `c` is a *constructor variable*.

System CT enables a clear separation between the issues of overloading resolution and definition of the structure of program interfaces. The type inference algorithm for System CT is a revised and extended version of a type inference algorithm used for computing *principal pairs* for core-ML[Mit96].

The rest of the paper is organized as follows. Section 3 introduces the type rules of our system. Section 4 presents the type inference algorithm and proves theorems of soundness and computation of principal types. Section 5 concludes.

3 Type System

We use a language similar to core-ML[Mil78,DM82,Mit88,Mit96], but new overloadings can be introduced in let-bindings. Recursive let-bindings are not considered (since they can be replaced by a fix point operator). For simplicity, let-bindings do not introduce nested scopes. Thus: $\mathtt{let}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{let}\ \mathtt{y} = \mathtt{e}_2\ \mathtt{in}\ \mathtt{e}$ is viewed as in a form $\mathtt{let}\ \{\mathtt{x} = \mathtt{e}_1;\ \mathtt{y} = \mathtt{e}_2\}\ \mathtt{in}\ \mathtt{e}$, with a list of declarations, as usually found in functional programming languages (in our system allowing $\mathtt{x} \equiv \mathtt{y}$ for the introduction of new overloadings).

Meta-variables α and β are used for type variables. We include value constructors ($k \in \mathcal{K}$) and type constructors ($C \in \mathcal{C}$). For simplicity, term variables ($x \in X$) are divided into two disjoint groups: let-bound ($o \in O$) and lambda-bound ($u \in U$). Meta-variable κ denotes a set $\{o_i : \tau_i\}^{i=1..n}$, called a set of constraints. A value constructor is considered as a let-bound variable which has a closed type with an empty set of constraints, fixed in a global typing context. Each type constructor C in $C\,\tau_1 \ldots \tau_n$ and each type variable α in $\alpha\,\tau_1 \ldots \tau_n$ has an arity, which is the value of n. It is assumed that there is a function type constructor ($\to$), which has arity 2 and is used in infix notation. A type variable with an arity greater than zero is called a constructor variable. Meta-variable χ is used to denote either a constructor variable or a type constructor.

Figure 1 gives the syntax of pre-terms and types of System CT.

A *typing context* Γ is a set of pairs $x : \sigma$. A let-bound variable can occur more than once in a typing context. A pair $x : \sigma$ is called a *typing for* x; if $\{x : \sigma_i\}^{i=1..n}$ is the set of typings for x in Γ, then $\Gamma(x) = \{\sigma_i\}^{i=1..n}$ is the set of types of x in Γ.

A *type substitution* (or simply substitution) is a function from type variables to simple types or type constructors, that differ from the identity function (id) only on finitely many variables. If σ is a type and S is a substitution, then $S\sigma$ denotes the type obtained by substituting $S(\alpha)$ for each occurrence of free type variable α in σ. Similarly, for a typing context Γ, $S\Gamma$ denotes $\{x : S\sigma \mid \ x : \sigma \in \Gamma\}$, and for a set of constraints κ, $S\kappa$ denotes $\{o : S\tau \mid \ o : \tau \in \kappa\}$.

$S|_V$ stands for the restriction of S to type variables in V: $S|_V(\alpha)$ is equal to $S(\alpha)$ if $\alpha \in V$, and to α otherwise.

$tv(\sigma)$ stands for the set of free type variables of σ; $tv(\kappa)$ and $tv(\Gamma)$ are defined similarly, considering types in κ and Γ, respectively. $tv(t_1, \dots, t_n)$ abbreviates $tv(t_1) \cup \dots \cup tv(t_n)$. For clarity, we sometimes drop the superscripts in, for example, $\{x_i\}^{i=1..n}$ and $\forall(\alpha_j)^{j=1..m}.\, \kappa.\, \tau$, assuming then systematically that i ranges from 1 to n and j ranges from 1 to m (where $m, n \geq 0$).

Types are modified, with respect to the type system of core-ML, to include constrained types. Typing formulas have the form $\Gamma \vdash e : (\kappa.\, \tau, \Gamma')$, where Γ' contains typings of let-bound variables used for checking satisfiability of constraints in κ (see description of *sat* below).

Quantification is defined over constrained types. Type σ is *closed* if $tv(\sigma) = \emptyset$. Renaming of bound variables in quantified types yield syntactically equal types.

The restriction of a set of constraints κ to a set of type variables V, denoted by $\kappa|_V$, is defined inductively as follows:

$$\begin{aligned}
&\emptyset|_V = \emptyset \\
&\{o : \tau\}|_V = \textbf{if } tv(\tau) \cap V = \emptyset \textbf{ then } \emptyset \textbf{ else } \{o : \tau\} \\
&(\{o : \tau\} \cup \kappa')|_V = \{o : \tau\}|_V \ \cup \ \kappa'|_V
\end{aligned}$$

The closure of restricting a set of constraints κ to a set of type variables V, denoted by $\kappa|^*_V$, is defined as follows:

$$\kappa|^*_V = \begin{cases} \kappa|_V & \text{if } tv(\kappa|_V) \subseteq V \\ \kappa|^*_{tv(\kappa|_V)} & \text{otherwise} \end{cases}$$

Informally, $\kappa|_V$ is obtained from κ by keeping only typings which have type variables that are in V. $\kappa|^*_V$ keeps typings that have type variables in V or in any typing kept "earlier". $S\kappa|^*_V$ means $(S\kappa)|^*_V$.

The intersection of a set of substitutions $\mathbb{S}$, denoted by $\bigcap \mathbb{S}$, is given by:

$$\begin{aligned}
&\textstyle\bigcap \{S\} = S \\
&\textstyle\bigcap (\{S\} \cup \mathbb{S}) = S|_V \text{ where } V = \{\alpha \mid \ S(\alpha) = S'(\alpha), \text{ and } S' = \bigcap \mathbb{S}\}
\end{aligned}$$

The type rules are given in Figure 2.

$$\begin{array}{lll}
\text{Terms} & e ::= x \mid \lambda u.\, e \mid e\, e' \mid \texttt{let } o = e \texttt{ in } e' & \\
\text{Simple Types} & \tau ::= C\, \tau_1 \ldots \tau_n \mid \alpha\, \tau_1 \ldots \tau_n & (n \geq 0) \\
\text{Constrained Types} & \Delta ::= \{o_i : \tau_i\}.\, \tau & (n \geq 0) \\
\text{Types} & \sigma ::= \forall \alpha_i.\, \Delta & (n \geq 0)
\end{array}$$

Figure 1: Abstract Syntax of System CT

$$\Gamma \vdash x : pt(x, \Gamma) \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\, \tau_1, \Gamma_1) \qquad \Gamma, o : close(\kappa_1.\, \tau_1, \Gamma) \vdash e_2 : (\kappa_2.\, \tau_2, \Gamma_2)}{\Gamma \vdash \texttt{let } o = e_1 \texttt{ in } e_2 : (\kappa|^*_{tv(\tau,\Gamma)}.\, \tau, \Gamma')} \quad \mathbb{S} \neq \emptyset \qquad \text{(LET)}$$

where $\mathbb{S} = sat\,(\kappa_1 \cup \kappa_2, \Gamma')$, $\quad S_\Delta = \bigcap \mathbb{S}|_{tv(\kappa_1 \cup \kappa_2) - tv(\Gamma)}$
$\tau = S_\Delta \tau_2$, $\Gamma' = \Gamma_1 \cup \Gamma_2$, $\kappa = unresolved\,(S_\Delta(\kappa_1 \cup \kappa_2), \Gamma')$

$$\frac{\Gamma, u : \tau' \vdash e : (\kappa.\, \tau, \Gamma')}{\Gamma \vdash \lambda u.\, e : (\kappa.\, \tau' \to \tau, \Gamma')} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash e_1 : (\kappa_1.\, \tau_1, \Gamma_1) \qquad \Gamma \vdash e_2 : (\kappa_2.\, \tau_2, \Gamma_2)}{\Gamma \vdash e_1\, e_2 : (\kappa|^*_{tv(\tau,\Gamma)}.\, \tau, \Gamma')} \quad \begin{array}{l} S = Unify(\{\tau_1 = \tau_2 \to \alpha\}, tv(\Gamma)) \\ \mathbb{S} \neq \emptyset \qquad \text{(APPL)} \\ not\ ambiguous(tv(S_\Delta S \kappa_1), \kappa, tv(\tau, \Gamma)) \end{array}$$

where $\mathbb{S} = sat\,(S(\kappa_1 \cup \kappa_2), \Gamma')$, $S_\Delta = \bigcap \mathbb{S}|_{tv(S(\kappa_1 \cup \kappa_2)) - tv(\Gamma)}$, α is a fresh type variable
$\tau = S_\Delta S \alpha$, $\Gamma' = \Gamma_1 \cup \Gamma_2$, $\kappa = unresolved\,(S_\Delta S(\kappa_1 \cup \kappa_2), \Gamma')$

Figure 2: Type Rules of System CT

Overloading is controlled by predicate ρ, in rule (LET). In this rule "$\Gamma, x : \sigma$" is a notation for updating Γ with $x : \sigma$, which requires $\rho(\sigma, \Gamma(x))$.[3] $\rho(\sigma, \mathcal{T})$ basically tests if σ is not unifiable with types in $\mathcal{T}$:

$$\Gamma, x : \sigma = \begin{cases} (\Gamma - \{x : \sigma'\}) \cup \{x : \sigma\} & \text{if } x \in \mathcal{U} \text{ and } x : \sigma' \in \Gamma \\ \Gamma \cup \{x : \sigma\} & \text{if } x \in O \text{ and } (\rho(\sigma, \Gamma(x)) \text{ holds, if } \Gamma(x) \neq \emptyset) \end{cases}$$

$$\begin{array}{l} \rho(o : \forall \alpha_i.\, \kappa.\, \tau, \mathcal{T}) = \\ \quad unify(\{\tau = \tau'\}) \text{ fails, for each } \forall \beta_j.\, \kappa'.\, \tau' \in \mathcal{T}, \\ \qquad \text{where type variables in } \{\alpha_i\} \text{ and } \{\beta_j\} \text{ are renamed to be distinct} \end{array}$$

[3] Rule (LET) could be modified in order to allow overlapping declarations (for which ρ, as defined, gives false). In this case, there should exist a mechanism for choosing a default implementation (usually the most specific one) for overloading resolution, in cases where overloading should have been resolved. This facility is not discussed in this paper and has been left for future work.

Rule (VAR) assigns to a given symbol the least common generalisation of the types of this symbol in the given typing context. For overloading resolution, this contrasts with a type system that enables the type of an overloaded symbol to be any of a given set of types in a typing context.

In System CT, instantiation is done on application, where substitution of type variables is determined by unification. The use of unification in rule (APPL) is a fundamental characteristic of the treatment of overloading in System CT. $Unify(E, V)$ is the standard algorithm for computing the most general unifying substitution for the set of type equations E[Rob65,Mit96], but considers that type variables in V are not unifiable (type variables in V are free type variables, originally introduced in types of lambda-bound variables). For reasons of space, we do not present the (slightly modified) unification algorithm.

Function *pt*, used in rule (VAR), uses function *lcg* (for *least common generalisation*) to give constrained types for overloaded symbols. In the construction of such types, *pt* renames term variables to fresh term variables, which are used only internally in the type system (i.e. they do not appear in programs). As an example where this renaming is needed, consider the types of e and e' in $e\,e'$, where $e \equiv (\texttt{let}\ x = e_1\ \texttt{in}\ \texttt{let}\ x = e_2\ \texttt{in}\ e_3)$ and $e' \equiv (\texttt{let}\ x = e'_1\ \texttt{in}\ \texttt{let}\ x = e'_2\ \texttt{in}\ e'_3)$. Without renaming, constraints on these types would both include typings for x. For simplicity, in examples where it is not necessary, renaming of let-bound variables in constraints is not used. $pt(x, \Gamma)$ is given as follows:

```
$pt(x, \Gamma) =$
    **if** $\Gamma(x) = \emptyset$ **then** fail **else**
    **if** $\Gamma(x) = \{\forall\alpha_j.\,\kappa.\,\tau\}$ **then** $(\kappa.\,\tau, \Gamma)$
     **else** $(\{x' : \tau\}.\,\tau, \Gamma[x'/x])$, where $\Gamma(x) = \{\forall(\alpha_j)_i.\,\kappa_i.\,\tau_i\}^{i=1..n}, n > 1$,
       $x'$ is a fresh term variable and $\tau = lcg(\{\tau_i\})$
```

$lcg(\{\tau_i\})$ gives the least common generalisation of $\{\tau_i\}$. It takes into account the fact that, for example, $lcg(\{\texttt{Int} \to \texttt{Int}, \texttt{Bool} \to \texttt{Bool}\})$ is $\alpha \to \alpha$, for some α, and not $\alpha \to \alpha'$, for some other type variable $\alpha' \not\equiv \alpha$. Other simple examples: $lcg(\{\texttt{Tree Int}, \texttt{List Int}\}) = \alpha\ \texttt{Int}$ and $lcg(\{\texttt{Tree}\ \beta, \texttt{List}\ \beta\}) = \alpha\ \alpha'$. Due to reasons of space, we do not present the definition of *lcg*.

Rule (LET) uses function *close*, to quantify types over type variables that are not free in a typing context: $close(\Delta, \Gamma) = \forall\alpha_j.\,\Delta$, where $\{\alpha_j\} = tv(\Delta) - tv(\Gamma)$.

Rule (APPL) has a premise with a type that is not in a functional form, and uses a fresh type variable, instead of having the usual premise with a functional type. This occurs because of overloading, as illustrated by the following simple example. Let $\Gamma = \{x : \texttt{Int}, x : \texttt{Int} \to \texttt{Int}, \texttt{1} : \texttt{Int}\}$. In this context the type derived for x is not a functional type; we derive instead $\Gamma \vdash x : \{x : \alpha\}.\,\alpha$. Then $\Gamma \vdash x\ 1 : \texttt{Int}$ is derivable, only with rule (APPL) as presented.

Overloading resolution in rule (APPL) is based on two tests. Consider an application $e\,e'$. The first test guarantees that all constraints occurring in types of both e and e', after unification, are *satisfiable*. This is based on function *sat*, also used in rule (LET) to control overloading resolution.

$sat\,(\kappa, \Gamma)$ returns a set of substitutions that unifies types of overloaded symbols in κ with the corresponding types for these symbols in Γ. It is defined inductively as follows, where we assume that quantified type variables (of types in Γ) are distinct from those in $tv(\kappa, \Gamma)$:

$$\begin{array}{l}
sat\,(\emptyset, \Gamma) = \{id\} \\
sat\,(\{o : \tau\}, \Gamma) = \bigcup_{\sigma_i \in \Gamma(o)} \{S \mid \; S = Unify(\{\tau = \tau_i\}, V) \text{ and } sat\,(S\kappa_i, S\Gamma) \neq \emptyset\} \\
\quad \text{where } V = tv(\tau_i) - \big(\{(\alpha_j)_i\} \cup tv(\tau)\big) \text{ and } \sigma_i = \forall (\alpha_j)_i.\, \kappa_i.\, \tau_i \\
sat\,(\{o : \tau\} \cup \kappa, \Gamma) = \bigcup_{S_i \in sat\,(\{o:\tau\}, \Gamma)} \bigcup_{S_{ij} \in sat\,(S_i\kappa, S_i\Gamma)} \{S_{ij} \circ S_i\}
\end{array}$$

To elucidate the meaning of *sat*, let $\Gamma = \{\texttt{f} : \texttt{Int} \rightarrow \texttt{Float}, \texttt{f} : \texttt{Float} \rightarrow \texttt{Int}, \texttt{one} : \texttt{Int}, \texttt{one} : \texttt{Float}\}$. Then $sat\,(\{\texttt{f} : \alpha \rightarrow \beta, \texttt{one} : \alpha\}, \Gamma)$ is a set with two substitutions, say $\{S_1, S_2\}$, where $S_1(\alpha) = \texttt{Int}$ and $S_1(\beta) = \texttt{Float}$), and $S_2(\alpha) = \texttt{Float}$ and $S_2(\beta) = \texttt{Int}$).

After the first test, any free type variable in constraints of types of e or e' that is mapped (by substitutions given by *sat*) to a single type σ, is replaced by σ, by application of S_Δ. Applying S_Δ thus eliminates constraints when overloading has been resolved.

Function *unresolved* is used to include unresolved constraints, if type σ above is itself the type of an overloaded symbol. For example, if `member` is overloaded for testing membership in lists and trees, having type $\{\texttt{member} : \alpha \rightarrow \beta\,\alpha \rightarrow \texttt{Bool}\}.\, \alpha \rightarrow \beta\,\alpha \rightarrow \texttt{Bool}$ in a context with typings $\texttt{member} : \forall \alpha\,\beta.\, \{= : \alpha \rightarrow \alpha \rightarrow \texttt{Bool}\}.\, \alpha \rightarrow [\alpha] \rightarrow \texttt{Bool}$ and $\texttt{member} : \forall \alpha\,\beta.\, \{= : \alpha \rightarrow \alpha \rightarrow \texttt{Bool}\}.\, \alpha \rightarrow \texttt{Tree}\,\alpha \rightarrow \texttt{Bool}$, then, in a context as above that has also typings $l : [\alpha'], x : \alpha'$, $\texttt{member}\,x\,l$ has type $\{= : \alpha' \rightarrow \alpha' \rightarrow \texttt{Bool}\}.\texttt{Bool}$. Function *unresolved* is defined as follows:

$$\begin{array}{l}
unresolved\,(\emptyset, \Gamma) = \emptyset \\
unresolved\,(\{o : \tau\} \cup \kappa, \Gamma) = \kappa' \cup unresolved(\kappa, \Gamma) \\
\quad \text{where } \kappa' = \textbf{if } sat(\{o : \tau\}, \Gamma) = \{S\}, \text{ for some } S, \\
\qquad\qquad\quad \textbf{then } unresolved(S\kappa', \Gamma), \text{ where } \forall \alpha_j.\, \kappa'.\, \tau' \in \Gamma(o),\ S\tau = S\tau' \\
\qquad\qquad\quad \textbf{else } \{o : \tau\}
\end{array}$$

The second test detects ambiguity by looking at the resulting constraint set κ. This includes all constraints for both e and e', after unification and (possibly) constraint elimination. Predicate *ambiguous* issues an error if and only if overloading can no longer be resolved. It is defined as follows:

$$\begin{array}{l}
ambiguous(V_1, \kappa, V) = V' \neq \emptyset \text{ and } V' \subseteq V_1 \\
\quad \text{where } V' = tv(\kappa) - tv(\kappa|^{*}_{V})
\end{array}$$

Type variables in a constraint of a constrained type $\kappa.\, \tau$ either occur free in τ or in the relevant typing context, or occur in another constraint with this property. Consider, for example, $\Gamma_{\texttt{f}} \vdash \texttt{f one} : \{\texttt{f} : \alpha \rightarrow \beta, \texttt{one} : \alpha\}.\, \beta$ (cf. section 2). Type variable α does not occur in $tv(\tau, \Gamma) = \{\beta\}$, but constraint $\{\texttt{one} : \alpha\}$ is captured by operation $\kappa|^*_{tv(\tau,\Gamma)}$ (where $\kappa = \{\texttt{f} : \alpha \to \beta, \texttt{one} : \alpha\}$). In general, a type variable may occur in $tv(\kappa) - tv(\kappa|^*_{tv(\tau,\Gamma)})$ if and only if the type variable does not appear in a constraint of the type of the function being applied (i.e. it does not appear in $S_\Delta S\kappa_1$). It may appear though in a constraint of the argument's type. For example, $\Gamma_o \vdash \texttt{fst (True,one)} : \texttt{Bool}$, where $\kappa = \{\texttt{one} : \beta\}$ and $\kappa|^*_{tv(\texttt{Bool},\Gamma_o)} = \emptyset$ and $\mathit{ambiguous}(\emptyset, \{\texttt{one} : \beta\}, \emptyset)$ is false.

Example $\Gamma_{\texttt{g}} \not\vdash \texttt{g one} : \{\texttt{g} : \alpha \to \texttt{Int}, \texttt{one} : \alpha\}. \texttt{Int}$ (see section 2) illustrates detection of a type error by predicate *ambiguous*. We have that $\kappa = \{\texttt{g} : \alpha \to \texttt{Int}, \texttt{one} : \alpha\} \neq \kappa|^*_\emptyset$. There is a constraint in the type of the applied function which should have been resolved (it would never be resolved in a later context).

4 Type inference

Figure 3 presents the type inference algorithm, PP_c, which gives principal pairs (type and context) for a given term. It allows local overloading only of symbols with closed types.[4] It is proved sound with respect to the version of the type system that uses this overloading policy (i.e. ρ_c below, instead of ρ), and to compute principal typings. The overloading policy is given by:

$$\begin{array}{l} \rho_c(o : \forall\alpha_j.\, \kappa.\, \tau, \mathcal{T}) = (\mathcal{T} = \emptyset) \text{ or} \\ \quad tv(\forall\alpha_j.\, \kappa.\, \tau) = \emptyset \text{ and for each } \sigma = \forall(\beta_k).\, \kappa'.\, \tau' \in \mathcal{T} \\ \quad\quad \mathit{unify}(\{\tau = \tau'\}) \text{ fails and } tv(\sigma) = \emptyset \end{array}$$

ρ_c requires closed types only for overloaded symbols ($\mathcal{T} = \emptyset$ means no typings for o in the relevant typing context, and in this case $\forall\alpha_j.\kappa.\tau$ may have free type variables). Any core-ML expression is a well-formed expression in System CT.

For simplicity, α-conversions are not considered, following the assumption that if a variable is let-bound, then it is not lambda-bound.

PP_c uses *typing environments* A, which are sets of elements $x : (\sigma, \Gamma)$. Pair (σ, Γ) is called an entry for x in A. We write $A(x)$ for the set of entries of x in A, and $A^t(x)$ for the set of first elements (types) in these entries.

Fresh type variables are assumed to be chosen so as to be different from any other type variable, including any type variable that occurs free in a typing context (in the type system, this assumption is not necessary).

$pt\epsilon(x, A)$ gives both type and context for x in A, as $pt(x, \Gamma)$, but requiring fresh type variables to be introduced for let-bound variables, as defined below:

```
$pt\epsilon(x, A) =$
    **if** $A(x) = \emptyset$ **then** $(\alpha, \{x : \alpha\})$, where $\alpha$ is a fresh type variable **else**
    **if** $A(x) = \{(\forall\alpha_j.\, \kappa.\, \tau, \Gamma)\}$ **then** $(\kappa.\, \tau, \Gamma)$,
      with quantified type variables $\{\alpha_j\}$ renamed as fresh type variables
     **else** $\big(\{x' : lcg(\{\tau_i\})\}.\, lcg(\{\tau_i\}), \bigcup \Gamma_i[x'/x]\big)$,
      where $A(x) = \{(\forall(\alpha_j)_i.\, \kappa_i.\, \tau_i, \Gamma_i)\}$ and $x'$ is a fresh term variable
```

[4] Provably sound and complete type inference for local overloading of symbols with non-closed types is, as far as we know, an open problem.

```
$PP_c(x, A) = pte(x, A)$
$PP_c(\lambda u.e, A) =$
    let $(\kappa.\, \tau, \Gamma) = PP_c(e, A)$
     in if $u : \tau' \in \Gamma$, for some $\tau'$  then $(\kappa.\, \tau' \to \tau,\ \Gamma - \{u : \tau'\})$
        else $(\kappa.\, \alpha \to \tau,\ \Gamma)$, where $\alpha$ is a fresh type variable
$PP_c(e_1\, e_2, A) =$
    let $(\kappa_1.\, \tau_1, \Gamma_1) = PP_c(e_1, A)$,   $(\kappa_2.\, \tau_2, \Gamma_2) = PP_c(e_2, A)$
        $S = unify(\{\tau_u = \tau'_u \mid u : \tau_u \in \Gamma_1$ and $u : \tau'_u \in \Gamma_2\} \cup \{\tau_1 = \tau_2 \to \alpha\})$
                                 where $\alpha$ is a fresh type variable
        $\Gamma' = S\Gamma_1 \cup S\Gamma_2$,   $\mathbb{S} = sat(S\kappa_1 \cup S\kappa_2, \Gamma')$
     in if $\mathbb{S} = \emptyset$  then fail
        else let $S_\Delta = \bigcap \mathbb{S}$,  $\Gamma = S_\Delta \Gamma'$,  $\tau = S_\Delta S\alpha$,
                 $V = tv(\tau, \Gamma)$,  $\kappa = unresolved(S_\Delta S(\kappa_1 \cup \kappa_2), \Gamma)$
              in if $ambiguous(tv(S_\Delta S\kappa_1), \kappa, V)$  then fail  else $\big(\kappa|^*_V.\, \tau, \Gamma\big)$
$PP_c($let $o = e_1$ in $e_2, A) =$
    let $(\kappa_1.\, \tau_1, \Gamma_1) = PP_c(e_1, A)$,   $\sigma = close(\kappa_1.\, \tau_1, \Gamma_1)$
     in if $\rho_c\big(\sigma, A^t(o)\big)$ does not hold  then fail
        else let $A' = A \cup \big\{o : (\sigma, \Gamma_1)\big\}$, $(\kappa_2.\, \tau_2, \Gamma_2) = PP_c(e_2, A')$
                 $S = unify(\{\tau_u = \tau'_u \mid\ u : \tau_u \in \Gamma_1, u : \tau'_u \in \Gamma_2\})$
                 $\Gamma' = S\Gamma_1 \cup S\Gamma_2$,   $\mathbb{S} = sat(S\kappa_1 \cup S\kappa_2, \Gamma')$
              in if $\mathbb{S} = \emptyset$  then fail
                 else let $S_\Delta = \bigcap \mathbb{S}$,   $\Gamma = S_\Delta \Gamma'$,  $\tau = S_\Delta S\tau_2$
                          $V = tv(\tau, \Gamma)$,  $\kappa = unresolved(S_\Delta S(\kappa_1 \cup \kappa_2), \Gamma)$
                      in $\big(\kappa|^*_V.\, \tau, \Gamma\big)$
```

Figure 3: Type inference for System CT

PP_c substitutes type variables that occur free in a typing context, reflecting the contexts in which corresponding free term variables are used. sat_c used in PP_c is simplified (with repect to its counterpart sat) due to allowing local overloading only of symbols with closed types:

$$sat_c\,(\emptyset, \Gamma) = \{id\}$$
$$sat_c\,(\{o : \tau\}, \Gamma) = \bigcup_{\sigma_i \in \Gamma(o)} \{S \mid\ S = unify(\{\tau = \tau_i\}) \text{ and } sat\,(S\kappa_i, \Gamma) \neq \emptyset\}$$
$$\quad \text{where } \sigma_i = \forall(\alpha_j)_i.\, \kappa_i.\, \tau_i$$
$$sat_c\,(\{o : \tau\} \cup \kappa, \Gamma) = \bigcup_{S_i \in sat\,(\{o:\tau\}, \Gamma)} \bigcup_{S_{ij} \in sat\,(S_i\kappa, \Gamma)} \{S_{ij} \circ S_i\}$$

4.1 Soundness and Completeness

In this section we prove soundness and completeness of PP_c with respect to the version of System CT that uses the overloading policy given by ρ_c.

Definition 1 A typing context Γ is *valid* if i) $u : \sigma \in \Gamma$ implies that σ is a simple type, ii) $u_1 : \sigma_1 \in \Gamma$, $u_2 : \sigma_2 \in \Gamma$, $\sigma_1 \not\equiv \sigma_2$, implies that $u_1 \not\equiv u_2$, and iii) $o : \sigma \in \Gamma$ implies that $\rho(\sigma, \Gamma(o) - \{\sigma\})$ holds.

Definition 2 An *o-closed* typing context is a valid typing context with the property that $o : \sigma_1, o : \sigma_2 \in \Gamma$, $\sigma_1 \not\equiv \sigma_2$ implies that $tv(\sigma_1, \sigma_2) = \emptyset$.

Definition 3 A typing environment A is *valid* if, for all $x : (\forall\alpha_j.\kappa.\,\tau, \Gamma) \in A$, Γ is o-closed, $\bigcap sat(\kappa, \Gamma) = id$ and $unresolved(\kappa, \Gamma) = \kappa$.

Definition 4 Γ' extends (or is an extension of) Γ if whenever X is the set of all typings for x in Γ, then X is also the set of all typings for x in Γ'.

Definition 5 A_Γ stands for $\{o : (\sigma, \Gamma) \mid o : \sigma \in \Gamma, o \in \mathcal{O}\}$, U_Γ stands for $\{u : \tau \mid u : \tau \in \Gamma, u \in \mathcal{U}\}$ and $O_A = \{o : \sigma \mid o \in \mathcal{O}$ and $o : (\sigma, \Gamma) \in A$ for some $\Gamma\}$.

Definition 6 $\kappa.\,\tau \preceq_S \kappa'.\,\tau'$ if $S(\kappa.\,\tau) = \kappa'.\,\tau'$, and $\Gamma \preceq_S \Gamma'$ if Γ' extends $S\Gamma$.

Lemma 1 If $\Gamma \vdash e :\ (\sigma, \Gamma_0)$ is provable and $\Gamma \cup \Gamma'$ is an extension of Γ, then $\Gamma \cup \Gamma' \vdash e :\ (\sigma, \Gamma_0')$ is provable, where Γ_0' is an extension of Γ_0.

Lemma 2 If $PP_c(e, A) = (\kappa.\,\tau, \Gamma)$, where A is valid, then Γ is valid, $tv(\Gamma) = tv(close(\kappa.\,\tau, \Gamma))$, $\bigcap sat(\kappa, \Gamma) = id$ and $unresolved(\kappa, \Gamma) = \kappa$.

Lemma 3 (Substitution Lemma) If $\Gamma \vdash e : (\kappa.\,\tau, \Gamma')$ is provable, where Γ is *o-closed*, and S, S_Δ are such that $dom(S) \subseteq tv(\Gamma)$, $sat_c(S\kappa, S\Gamma') \neq \emptyset$ and $S_\Delta = \bigcap sat(S\kappa, S\Gamma')$, then $S_\Delta S\Gamma \vdash_\varepsilon e : (\kappa'.\,\tau', S_\Delta S\Gamma')$ is provable, where $\kappa'.\tau' = S_\Delta|_{tv(\Gamma)} S(\kappa.\tau)$ or $\kappa'.\tau' = (S_\Delta S\kappa)|_V^*.S_\Delta S\tau$, where $V = tv(S_\Delta S\tau, S_\Delta S\Gamma)$.

Theorem 1 (Soundness of PP_c) If $PP_c(e, A) = (\kappa.\,\tau, \Gamma)$, where A is valid, then $O_A \cup U_\Gamma \vdash e : (\kappa.\,\tau, \Gamma')$ is provable, where $\Gamma' = O_A \cup \Gamma$.

Theorem 2 (Completeness) Let Γ_0 be *o-closed*. If $\Gamma_0 \vdash e : (\kappa_0.\,\tau_0, \Gamma')$ is provable then either: i) $PP_c(e, A_{\Gamma_0}) = (\kappa.\,\tau, \Gamma)$ and there is S such that $O_A \cup U_\Gamma \preceq_S \Gamma_0$ and $\kappa.\tau \preceq_S \kappa_0.\,\tau_0$, or ii) $PP_c(e, A_{\Gamma_0})$ fails and e has a subexpression of the form `let` $o = e_1$ `in` e_2 such that $\Gamma_1 \vdash e_1 : (\kappa_1.\,\tau_1, \Gamma_1')$ is provable in the derivation of $\Gamma_0 \vdash e : (\kappa_0.\tau_0, \Gamma')$, for some Γ_1, Γ_1', $\kappa_1.\tau_1$, but $\Gamma_1 \vdash e_1 : (\kappa_1'.\tau_1', \Gamma_1'')$ is also provable, for some Γ_1'', $\kappa_1'.\tau_1'$ such that $\kappa_1'.\,\tau_1' \preceq_{S_1} \kappa_1.\,\tau_1$, for some S_1, and $\rho_c(close(\kappa_1'.\,\tau_1', \Gamma_1), \Gamma_1(o))$ does not hold.

An example of item (b) is `let` $f = \lambda x.\,x$ `in let` $f = \lambda y.\,y$ `in`... . Types `Int` $\rightarrow$ `Int` and `Float` $\rightarrow$ `Float` can be derived for $\lambda x.x$ and $\lambda y.y$, respectively, and overloading of `f` with these two definitions is allowed. PP_c, however, infers the most general type for each of these expressions, namely $\forall\alpha.\alpha \rightarrow \alpha$, and thus does not allow overloading with these two definitions.

Principal Typings

Definition 7 A *typing problem* is a pair (e, Γ_0), where e is an expression and Γ_0 is a valid typing context. A *typing solution* for this typing problem is a pair $(\kappa.\,\tau, \Gamma)$ such that $\Gamma \preceq_S \Gamma_0$, for some S, and $\Gamma \vdash e :\ (\kappa.\,\tau, \Gamma')$ is provable, for some Γ'. It is the *most general* if, for every other typing solution $(\kappa'.\,\tau', \Gamma')$ to this problem, there exists S such that $\Gamma \preceq_S \Gamma'$ and $\kappa.\,\tau \preceq_S \kappa'.\,\tau'$.

Theorem 3 (Principal Typings) If $PP_c(e, A_{\Gamma_0}) = (\kappa.\ \tau, \Gamma)$, where Γ_0 is *o-closed*, then $(\kappa.\ \tau, O_A \cup U_\Gamma)$ is the most general solution for (e, Γ_0), and if $PP_c(e, A_{\Gamma_0})$ fails, then (e, Γ_0) does not have a most general solution.

5 Conclusion and further work

The type inference algorithm presented in this paper computes most general typings and supports overloading and polymorphism without the need for declarations or annotations, and with the only restriction that types of local overloaded definitions must be closed types. No language construct is introduced for the purpose of coping with overloading. The context-dependent overloading policy, controlled by a single predicate used in let bindings, allows overloading of constants and other similar forms of overloadings. Information provided by the context in which an expression appears is used to detect ambiguity on the use of overloaded symbols. When there is ambiguity, a type error is detected, and when it is possible for overloading to be resolved further on in a later context, this is reflected in the type of the expression.

Our work is a continuation of previous works directed towards a comprehensive support for overloading, together with polymorphism and type inference. The mechanism of type classes used in Haskell represented a big step forward in this direction. In an upcoming article, a formal semantics of the language of System CT will be presented, together with a type soundness theorem. Further work involves incorporating the type inference algorithm in a modern functional language, and studying support for subtyping and separate compilation. We have a prototype implementation, and are developping it with the aim of exploring further the idea of using overloading to support writing reusable code.

Though we have not discussed the efficiency and time complexity of the type inference algorithm in this article, upon a preliminary evaluation we think that its efficiency should be approximately the same as that of the algorithm used for example for ML. We do not have yet a result on the complexity of the type inference algorithm but we plan to study the subject in detail and perform extensive experimentations with our prototype. We note that the cost of computing *sat*, as given in this paper, is proportional to m and n, where m is the number of typings on a given constraint set and n is the (average) number of overloadings of a given symbol. Usually, both m and n are small. Moreover, for any given typing $o : \tau$ occurring in a given constraint set k, the number of substitutions obtained as a result of unifying τ with typings for o in a given typing context is “even smaller”. If this number is zero, this can eliminate the need to compute all other unifications for the remaining constraints in κ. Other optimizations may be worthwhile. We expect a practical implementation of our type inference algorithm in a language such as Haskell or SML to be efficient.

Acknowledgements

We acknowledge the financial support of FAPEMIG under Project Tec 1243/97.

References

[DCO96] Dominic Duggan, Gordon Cormack, and John Ophel. Kinded type inference for parametric overloading. *Acta Informatica*, 33(1):21–68, 1996.

[DM82] Luís Damas and Robin Milner. Principal type schemes for functional programs. *POPL'82*, pages 207–212, 1982.

[HHJW96] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM TOPLAS*, 18(2):109–138, March 1996.

[J^+98] Simon Peyton Jones et al. GHC – The Glasgow Haskell Compiler. Available at http://www.dcs.gla.ac.uk/fp/software/ghc/, 1998.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. *ACM SIGPLAN Haskell Workshop*, 1997.

[Jon94] Mark Jones. *Qualified Types.* Cambridge University Press, 1994.

[Jon95] Mark Jones. A system of constructor classes: overloading and higher-order polymorphism. *Journal of Functional Programming*, 5:1–36, 1995.

[Jon96] Simon P. Jones. Bulk types with class. In Phil Trinder, ed., *Eletronic Proc. 1996 Glasgow Workshop on Functional Programming*, October 1996. http://ftp.dcs.glasgow.ac.uk/fp/workshops/fpw96/PeytonJones.ps.gz.

[Jon98a] M. Jones. Hugs: The Haskell User's Gofer System. haskell.org/hugs, 1998.

[Jon98b] Simon Peyton Jones. Multi-parameter type classes in GHC. Available at http://www.dcs.gla.ac.uk/~simonpj/multi-param.html, 1998.

[Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, ed., *ESOP*, LNCS 300, pages 131–144, 1988.

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

[Mit88] John Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.

[Mit96] John Mitchell. *Foundations for programming languages.* MIT Press, 1996.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1989.

[NP93] Tobias Nipkow and Christian Prehofer. Type Reconstruction for Type Classes. *Journal of Functional Programming*, 1(1):1–100, 1993.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A Second Look at Overloading. In *ACM Conf. Funct. Prog. Comp. Arch.*, pages 135–146, 1995.

[Pau96] L. Paulson. *ML for the Working Programmer, Cambridge Univ. Press, 1996.*

[Pe97] J. Peterson and K. Hammond (eds.). Rep. on the prog. lang. Haskell, a non-strict, purely funct. lang. (v1.4). Techn. rep. , Haskell committee, 1997.

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

[Smi91] Geoffrey Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, 1991.

[Tho96] S. Thompson. *Haskell: The Craft of Funct. Prog.*, Addison-Wesley, 1996.

[Tur85] D. Turner. A non-strict funct. lang. with polymorphic types. In *2nd Int. Conf. on Funct. Prog. and Comp. Arch.*, IEEE Comp. Soc. Press, 1985.

[Wad89] P. Wadler. How to make ad-hoc polymorphism less ad hoc. *POPL'89*, 1989.