Electronic Notes in Theoretical Computer Science **14** (1998)
URL: `http://www.elsevier.nl/locate/entcs/volume14.html` 10 pages

A Type System for Context-dependent Overloading

Carlos Camarão

Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, 31270-010 Belo Horizonte, Brasil

Lucília Figueiredo

Departamento de Computação, Universidade Federal de Ouro Preto, 35400-000 Ouro Preto, Brasil

Abstract

This article presents a type system for context-dependent overloading, based on the notion of constrained types. These are types constrained by the definition of functions or constants of given types. This notion supports both overloading and a form of subtyping, and is related to Haskell type classes [11,2], System O [7] and other systems with constrained types [9,8]. We study an extension of the Damas-Milner system[4,1] with constrained types. The inference system presented uses a context-dependent overloading policy, which is specified by means of a predicate used in a single inference rule. The idea simplifies the treatment of overloading, enables the simplification of inferred types (by means of class type annotations), and is adequate for use in a type system with higher-order types.

1 Introduction

In a previous work by the authors, presented at the *First Workshop on Formal Foundations of Software Systems* [1], a type system for context-independent overloading was presented, which removed some restrictions imposed by existing systems with support for polymorphism, type inference and overloading. The article also made preliminary comments on the idea of defining types parameterised on constrained polymorphic types. This article presents a more powerful type system, that adopts a context-dependent overloading policy. Using this policy, overloading is resolved when (and if) there is enough information provided by the relevant context. Consider the following example:

[1] Sponsored by the National Science Foundation (NSF) and *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq) and held in the *Pontifícia Universidade Católica do Rio de Janeiro* on 5-9 May 1997.

```
let one = 1 in
let one = 1.0 in ...
```

In our core system, called CT, the type assigned to one in a context after the let-bindings above is:

$$\{\texttt{one} : \alpha\}.\, \alpha$$

where α is a fresh type variable. That indicates, informally, a type for which there is a constant one of that type. In a context after the let-bindings, requiring a value of type Int, one will behave as an Int. In this context, Int (as well as Real) is an instance of such type.

In a typing context Γ that has also an overloaded symbol f, with typings $\texttt{f} : \texttt{Int} \to \texttt{Real}$, $\texttt{f} : \texttt{Real} \to \texttt{Int}$, System CT infers the following typings:

$$\texttt{f} : \{\texttt{f} : \alpha \to \beta\}.\, \alpha \to \beta$$

$$\texttt{f one} : \{\texttt{f} : \alpha \to \beta, \texttt{one} : \alpha\}.\, \beta$$

where α and β are fresh type variables. The typing for f one indicates that, in context Γ, this expression works like an overloaded constant; it can be used in a context requiring either a value of type Int or Real.

Consider now a typing context where we have: $\texttt{g} : \texttt{Int} \to \texttt{Int}$, $\texttt{g} : \texttt{Real} \to \texttt{Int}$, $\texttt{one} : \texttt{Int}$ and $\texttt{one} : \texttt{Real}$. We have the inferred type:

$$\texttt{g} : \{\texttt{g} : \alpha \to \texttt{Int}\}.\, \alpha \to \texttt{Int}$$

Expression g one generates a type error. No context containing this expression can "resolve the overloading". We will discuss this example further on Section 3.

In CT, as in System O, and in constrast to Haskell type classes, a program can be assigned a meaning independent of its types, and every typeable program has a single most general type.

The rest of the paper is organized as follows. Section 2 introduces the type rules of System CT. Section 3 presents some examples of type deduction. Section 4 presents the type inference algorithm. Section 5 concludes.

2 Type System

We use a kernel language that is similar to Core-ML [4,1,5] [6, Section 11.2]. We include value constructors ($k \in \mathcal{K}$) and type constructors ($C \in \mathcal{C}$) and

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

Fig. 1. Abstract Syntax of system CT

assume (for simplicity) that overloaded variables are distinct from value constructors and non-overloaded variables; in particular, all lambda-bound variables are non-overloaded.

Term variables ($x \in X$) are divided into three groups: overloaded ($o \in O$), non-overloaded ($u \in U$), and value constructors ($k \in \mathcal{K}$), the latter being considered as constants, having a value fixed in a global environment.

Meta-variables α and β are used for type variables. Meta-variable κ is used to denote a set of pairs $\{o_i : \tau_i\}$, which is called a set of constraints.

The notation $tv(\sigma)$ stands for the set of free type variables of σ. We assume systematically that index i in, say, x_i, indicates the sequence $x_1, \ldots, x_n$, and similarly for index j, ranging from 1 to m (where $m, n \geq 0$). For example $\{o_i : \tau_i\}$ and $\{\alpha_j\}$ are abbreviations for $\{o_1 : \tau_1, \ldots, o_n : \tau_n\}$ and $\{\alpha_1, \ldots, \alpha_m\}$, respectively.

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

Types are modified (with respect to the type system of Core-ML) to include constrained types. Quantified constrained types are restricted to types of let-bound variables in typing contexts.

Renaming of bound variables in quantified types yield syntactically equal types. Types $\forall \alpha_j \beta_k.\, \Delta$ and $\forall \alpha_j.\, \Delta$ are also syntactically equal if $tv(\{\beta_k\}) \cap tv(\Delta) = \emptyset$. A constrained type with a constraint that has no type variables is syntactically equal to one without this constraint. In particular, a constrained type $\kappa.\, \tau$ for which $\kappa = \emptyset$ is syntactically equal to τ.

The type rules are given in Figure 2.

A *typing context* Γ is a set of pairs, written as $x : \sigma$. In our system, a variable x can occur more than once in a typing context, if $x \in O$. A pair $x : \sigma$ is called a *typing for* x. The notation Γ_x indicates a typing context for which it is assumed that x does not appear (this does not cause any restrictions due to the possibility of renaming bound variables).

A *type substitution* (or simply substitution) is a function from type variables to types. If σ is a type and S is a substitution, then $S\sigma$ is used to denote the type obtained by replacing each free type variable α in σ with $S(\alpha)$. Similarly, for a typing context Γ, the notation $S\Gamma$ denotes $\{x : S\sigma \mid \ x : \sigma \in \Gamma\}$, and for a set of constraints κ, the notation $S\kappa$ denotes $\{o : S\tau \mid \ o : \tau \in \kappa\}$.

The overloading policy is based on unification of simple types. Function

$Unify(E,V)$ computes the most general unifying substitution for the set of equations E between type expressions, considering that type variables in V are not unifiable. We define (C is considered below to include the $\rightarrow$ type constructor):

$$unify(E) = Unify(E, \emptyset)$$

$$Unify(\emptyset, V) = \emptyset$$

$$\begin{array}{l} Unify(E \cup \{C\,\tau_1 \ldots \tau_n = C'\,\tau'_1 \ldots \tau'_m\}, V) = \\ \qquad \textbf{if } C \not\equiv C' \textbf{ then } \text{fail} \\ \qquad \textbf{else } Unify(E \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}, V) \quad (\text{where } m = n) \end{array}$$

$$\begin{array}{l} Unify(E \cup \{\alpha = \tau\}, V) = \\ \qquad \textbf{if } \alpha \equiv \tau \textbf{ then } Unify(E,V) \\ \qquad \textbf{else if } \alpha \text{ occurs in } \tau \textbf{ then } \text{fail} \\ \qquad\qquad \textbf{else if } \alpha \in V \textbf{ then} \\ \qquad\qquad\qquad \textbf{if } \tau \equiv \beta, \text{ where } \beta \notin V \\ \qquad\qquad\qquad\quad \textbf{then } Unify(E[\beta :\rightarrow \alpha], V) \circ (\beta \mapsto \alpha) \\ \qquad\qquad\qquad\quad \textbf{else } \text{fail} \\ \qquad\qquad\qquad \textbf{else } Unify(E[\alpha :\rightarrow \tau], V) \circ (\alpha \mapsto \tau) \end{array}$$

The overloading policy is controlled by predicate ρ, used in rule (LET). The value given by $\rho(\sigma_1, \sigma_2)$ means "σ_1 and σ_2 can be types of overloaded symbols". The evaluation of $\rho(\sigma_1, \sigma_2)$ basically tests if the simple types in σ_1 and σ_2 are not unifiable; it is defined by:

$$\rho(\sigma_1, \sigma_2) = \begin{cases} unify(\{\tau_1 = \tau_2\}) \text{ fails} & \text{if } \sigma_1 \equiv \forall \alpha_i.\, \kappa_1.\, \tau_1, \sigma_2 \equiv \forall \beta_j.\, \kappa_2.\, \tau_2, \\ & \quad tv(\tau_1) \subseteq \{\alpha_i\} \text{ and } tv(\tau_2) \subseteq \{\beta_j\} \\ \text{false} & \text{otherwise} \end{cases}$$

Quantified type variables in a given type are assumed above to be distinct from other type variables (possibly by renaming).

The notation $tv(\Gamma)$ stands for the set of free type variables of Γ.

The notation $\Gamma, x : \sigma$ stands for:

$$\Gamma, x : \sigma = \begin{cases} \Gamma_x \cup \{x : \sigma\} \text{ if } x \in U \\ \Gamma \cup \{x : \sigma\} \quad \text{if } x \in O \wedge \{x : \sigma'\} \in \Gamma \Rightarrow \rho(\sigma, \sigma') \end{cases}$$

A type of an overloaded symbol cannot contain a free type variable (in other words, it can only be a closed type). This forbids "local overloading", so that x cannot be overloaded, using `let` $x = e$ `in` e', if a free type variable occurs in the type of e.

Function *lcg* computes the type that is the *least common generalisation* for a set of types. *lcg* is defined by (where C is considered below to include the $\rightarrow$ type constructor):

$$lcg(\{\forall\alpha_j\{o_i : \tau_i\}.\,\tau\} \cup \mathcal{S}) = lcg(\{\tau\} \cup \mathcal{S})$$

$$lcg(\{\tau\}) = \tau$$

$$\begin{array}{l} lcg(\{C\,\tau_1 \ldots \tau_n,\ C'\,\tau'_1 \ldots \tau'_n\} \cup \mathcal{S}) = \\ \quad \textbf{if } C \not\equiv C' \ \textbf{ then } \alpha, \text{ where } \alpha \text{ is a fresh type variable} \\ \quad\ \textbf{else } lcg(\mathcal{S} \cup \{C\ lcg_1\ \ldots\ lcg_n\}) \\ \qquad \text{where } lcg_i = lcg(\{\tau_i, \tau'_i\}), \text{ for } i = 1, \ldots, n \text{ and type variables} \\ \qquad\qquad \text{are renamed so that } \alpha \equiv \alpha' \text{ whenever there exists} \\ \qquad\qquad \tau_a, \tau_b \text{ such that } lcg(\{\tau_a, \tau_b\}) = \alpha \text{ and } lcg(\{\tau_a, \tau_b\}) = \alpha' \end{array}$$

$$lcg(\{\alpha\} \cup \mathcal{S}) = \alpha', \text{ where } \alpha' \text{ is a fresh type variable}$$

Function *lcg* takes into account the fact that, for example, $lcg(\{\texttt{Int} \rightarrow \texttt{Int}, \texttt{Bool} \rightarrow \texttt{Bool}\})$ is $\alpha \rightarrow \alpha$, for some type variable α, and not $\alpha \rightarrow \alpha'$, for some other type variable $\alpha' \not\equiv \alpha$.

Function *pt*, used in rule (VAR), uses function *lcg* to give constrained types for overloaded symbols.

The value $pt(x,\Gamma)$ is given as follows. If $x \in U$, let τ_0 be the typing for x in Γ; otherwise ($x \in O$) let $\{x : \sigma_1, \ldots, x : \sigma_n\}$ be the set of all typings for x in Γ; we have:

$$pt(x,\Gamma) = \begin{cases} \tau_0 & \text{if } x \in U \\ lcg(\{\sigma_i\}) & \text{otherwise} \end{cases}$$

For any given typing context Γ, we define an instance relation $\leq_\Gamma$ for this context, between simple and class types, by:

- $\tau \leq_\Gamma \tau'$ if there exists S such that $\tau \equiv S\tau'$;
- $\tau \leq_\Gamma \forall\alpha_j.\,\kappa.\,\tau'$ if there exists S such that $\tau \equiv S\tau'$ and $sat(S\kappa,\Gamma) \neq \emptyset$.

Function $sat(\kappa,\Gamma)$ returns a set of substitutions that unifies types of overloaded symbols in κ with the corresponding types for these symbols in Γ. Function *sat* is used in the side-conditions of rules (APPL) and (INST) to control overloading resolution. It is defined by:

$$\begin{array}{rl} sat(\{o_i : \tau_i\},\Gamma) = \{S \mid & dom(S) \subseteq tv(\{\tau_i\}) \text{ and} \\ & S\tau_i \leq_\Gamma \sigma_i,\ \text{for some } o_i : \sigma_i \in \Gamma\} \end{array}$$

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

$$\frac{\Gamma \vdash e : \kappa.\,\tau \qquad \Gamma, o : close(\kappa.\,\tau, \Gamma) \vdash e' : \kappa'.\,\tau'}{\Gamma \vdash \texttt{let}\ o = e\ \texttt{in}\ e' : \kappa \cup \kappa'.\,\tau'} \quad sat(\kappa \cup \kappa', \Gamma) \neq \emptyset \qquad \text{(LET)}$$

$$\frac{\Gamma \vdash e : \kappa.\,\tau}{\Gamma \vdash e : S(\kappa.\,\tau)} \quad \{S\} = sat(\kappa, \Gamma) \qquad (\text{INST}\Delta)$$

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

$$\frac{\Gamma \vdash e : \kappa.\,\tau \qquad \Gamma \vdash e' : \kappa'.\,\tau'}{\Gamma \vdash e\,e' : S(\kappa \cup \kappa'.\,\alpha)} \quad \begin{array}{l} S = Unify(\{\tau = \tau' \rightarrow \alpha\}, tv(\Gamma)) \\ ss(S(\kappa \cup \kappa'.\,\alpha), \Gamma) \end{array} \qquad \text{(APPL)}$$

where α is a fresh type variable

Fig. 2. Type Rules of system CT

Predicate ss (for single substitution), used in rule (APPL), controls the instantiation of constrained types by application. The value $ss(\kappa.\,\tau, \Gamma)$ is defined by:

$$\begin{array}{l} sat(\kappa, \Gamma) \neq \emptyset, \text{ and} \\ tv(\tau, \Gamma) \cap tv(\kappa) = \emptyset \text{ implies that } sat(\kappa, \Gamma) \text{ is a singleton} \end{array}$$

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

3 Examples

In this section we present simple illustrative examples of type derivations in System CT.

3.1 Application to Overloaded Constant

Consider a typing context Γ with typings

$$\begin{aligned}&\mathtt{g} : \mathtt{Int} \to \mathtt{Int},\\ &\mathtt{g} : \mathtt{Real} \to \mathtt{Int},\\ &\mathtt{one} : \mathtt{Int},\ \text{ and}\\ &\mathtt{one} : \mathtt{Real}\end{aligned}$$

The following are derivable, from (VAR):

$$\begin{aligned}&\Gamma \vdash \mathtt{g} : \{\mathtt{g} : \alpha \to \mathtt{Int}\}.\, \alpha \to \mathtt{Int}\\ &\Gamma \vdash \mathtt{one} : \{\mathtt{one} : \alpha\}.\, \alpha\end{aligned}$$

can be inferred.

Now, $\mathtt{g}$ cannot be applied to $\mathtt{one}$, because $ss(\mathtt{Int}, \{\mathtt{g} : \alpha \to \mathtt{Int}, \mathtt{one} : \alpha\}.\, \mathtt{Int}, \Gamma)$ cannot be satisfied, since $sat(\{\mathtt{g} : \alpha \to \mathtt{Int}, \mathtt{one} : \alpha\}, \Gamma)$ is a set with two substitutions, namely, $(\alpha \mapsto \mathtt{Int})$ and $(\alpha \mapsto \mathtt{Real})$.

An application of $\mathtt{g}$ to a constant $\mathtt{c}$ of type $\mathtt{Int}$ (or $\mathtt{Real}$) would generate a correct typing $\mathtt{g}\ \mathtt{c} : \mathtt{Int}$.

An application to a constant of a different type, other that $\mathtt{Int}$ or $\mathtt{Real}$, would constitute a type error, since ss would be false due to sat being empty.

3.2 Overloaded Division

Consider a typing context Γ with typings

$$\begin{aligned}&(/) : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int},\\ &(/) : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Real},\\ &(/) : \mathtt{Real} \to \mathtt{Real} \to \mathtt{Real},\\ &(=) : \mathtt{Int} \to \mathtt{Int} \to \mathtt{Int},\ \text{ and}\\ &(=) : \mathtt{Real} \to \mathtt{Real} \to \mathtt{Real}\end{aligned}$$

Figure 3 presents a type derivation for

$$(4/2)/(5/2) = 1 : \mathtt{Bool}$$

in System CT. In this figure we use $\mathtt{I}$, $\mathtt{R}$ and $\mathtt{B}$ for $\mathtt{Int}$, $\mathtt{Real}$ and $\mathtt{Bool}$, respectively; sequents are abbreviated, by writing only terms and their types, since the typing context is always Γ.

$$
\dfrac{\dfrac{(=\mathtt{1}) : \mathtt{I} \to \mathtt{B} \qquad \dfrac{\dfrac{/ : \{/ : \alpha \to \alpha \to \alpha'\}.\, \alpha \to \alpha \to \alpha' \qquad \dfrac{\dfrac{\mathtt{4} : \mathtt{I} \quad / : \{/ : \alpha \to \alpha \to \beta\}.\, \alpha \to \alpha \to \beta}{(\mathtt{4}/) : \{/ : \mathtt{I} \to \mathtt{I} \to \beta\}.\, \mathtt{I} \to \beta} \qquad \mathtt{2} : \mathtt{I}}{\mathtt{4/2} : \{/ : \mathtt{I} \to \mathtt{I} \to \beta\}.\, \beta}}{((\mathtt{4/2})/) : \{/ : \mathtt{I} \to \mathtt{I} \to \beta, / : \beta \to \beta \to \alpha'\}.\, \beta \to \alpha'} \qquad \mathtt{5/2} : \{/ : \mathtt{I} \to \mathtt{I} \to \beta'\}.\, \beta'}{(\mathtt{4/2})/(\mathtt{5/2}) : \{/ : \mathtt{I} \to \mathtt{I} \to \beta, / : \beta \to \beta \to \alpha'\}.\, \alpha'}}{(\mathtt{4/2})/(\mathtt{5/2}) = \mathtt{1} : \{/ : \mathtt{I} \to \mathtt{I} \to \beta, / : \beta \to \beta \to \mathtt{I}\}.\, \mathtt{B}}\ (\text{APPL})}{(\mathtt{4/2})/(\mathtt{5/2}) = \mathtt{1} : \mathtt{B}}\ (\text{INST}\Delta)
$$

Fig. 3. Illustrative type derivation for System CT

From this typing derivation, it is easy to see that

$$(4/2)/(5/2) = 1.0 : \mathtt{Bool}$$

is not typable, since $sat(\{/ : \mathtt{Int} \to \mathtt{Int} \to \beta, / : \beta \to \beta \to \mathtt{Real}\}, \Gamma) = \{(\beta \mapsto \mathtt{Int}), (\beta \mapsto \mathtt{Real})\}$ (not a singleton).

4 Type inference

Figure 4 presents the type inference algorithm. Function PP computes principal pairs (type and context) for a given term.

For simplicity, we do not consider α-substitutions and assume that if a variable is let-bound, then it is not lambda-bound. We can see that this assumption is important in examples for which the assumptions do not hold: for $\mathtt{let}\ x = 10\ \mathtt{in}\ \lambda x.\, x\ \mathit{or}\ \mathit{true}$, PP would fail, and for $\mathtt{let}\ x = 10\ \mathtt{in}\ \lambda x.\, x$, PP would not give a principal typing.

Algorithm PP uses *typing environments* A, which are sets of pairs written in the form $x : (\sigma, \Gamma)$. We write $A(x)$ for the set of triples (σ_x, Γ_x) such that $x : (\sigma_x, \Gamma_x) \in A$.

5 Conclusion

In extensions of the Damas-Milner type system, constrained types and class types, as defined and used in System CT, provide a simplified treament of overloading.

In System CT the overloading policy is controlled by means of a single predicate (ρ), used in rule (LET).

Types can be inferred, without the need for any type annotations, and there is an algorithm for computing most general typings.

We are currently working on a denotational semantics for terms and types of System CT. We are also working on a proof that algorithm PP given in this paper indeed computes most general typings.

System CT has less restrictions than existing similar systems that extend ML-like polymorphic type systems with overloading, for example by allowing that types of overloaded functions have a type variable as the outermost type, and overloaded non-functional values. System CT also allows overloaded

$$
\begin{array}{l}
PP(u, A) = \\
\qquad (\alpha, \{u : \alpha\}),\ \text{where } \alpha \text{ is a fresh type variable} \\
PP(o, A) = \\
\qquad \textbf{if } A(o) = \{(\forall \alpha_j.\ \kappa\, \tau, \Gamma)\},\ \text{for some } \{\alpha_j\}, \kappa, \tau, \Gamma \\
\qquad\quad \textbf{then } (\kappa.\ \tau, \Gamma) \\
\qquad\quad \textbf{else if } A(x) = \{(\sigma_i, \Gamma_i)\},\ \text{for some } \{(\sigma_i, \Gamma_i)\} \\
\qquad\qquad\quad \textbf{then } \big(pt(x, \{x : \sigma_i\}), \bigcup \Gamma_i\big) \\
PP(\lambda u.e, A) = \\
\qquad \textbf{let } PP(e, A) = (\kappa.\ \tau, \Gamma)\ \ \textbf{in} \\
\qquad\quad \textbf{if } u : \tau' \in \Gamma,\ \text{for some } \tau' \\
\qquad\qquad \textbf{then } (\kappa.\ \tau' \to \tau,\ \Gamma - \{u : \tau'\}) \\
\qquad\qquad \textbf{else } (\kappa.\ \alpha \to \tau,\ \Gamma),\ \text{where } \alpha \text{ is a fresh type variable} \\
PP(e_1\, e_2, A) = \\
\qquad \textbf{let } PP(e_1, A) = (\kappa_1.\ \tau_1, \Gamma_1) \\
\qquad\quad PP(e_2, A) = (\kappa_2.\ \tau_2, \Gamma_2),\ \text{with type variables renamed} \\
\qquad\qquad\qquad \text{to be different from those in } (\tau_1, \kappa_1, \Gamma_1) \\
\qquad\quad S = unify(\{\tau_u = \tau'_u \mid\ u : \tau_u \in \Gamma_1 \text{ and } u : \tau'_u \in \Gamma_2\} \cup \{\tau_1 = \tau_2 \to \alpha\}), \\
\qquad\qquad\qquad \text{where } \alpha \text{ is a fresh type variable} \\
\qquad\quad \Gamma = S\Gamma_1 \cup S\Gamma_2 \\
\qquad \textbf{in if } ss(S(\kappa_1 \cup \kappa_2.\ \alpha),\ \Gamma)\ \ \textbf{then} \\
\qquad\qquad \textbf{if } sat(S(\kappa_1 \cup \kappa_2), \Gamma) = \{S_\Delta\},\ \text{for some } S_\Delta, \\
\qquad\qquad\quad \textbf{then } (S_\Delta\, S(\kappa_1 \cup \kappa_2.\ \alpha), \Gamma) \\
\qquad\qquad\quad \textbf{else } (S(\kappa_1 \cup \kappa_2.\ \alpha), \Gamma) \\
\qquad \textbf{else } \text{fail} \\
PP(\texttt{let } o = e_1 \texttt{ in } e_2, A) = \\
\qquad \textbf{let } PP(e_1, A) = (\kappa_1.\ \tau_1, \Gamma_1) \\
\qquad\quad \sigma = close(\kappa_1.\ \tau_1, \Gamma_1) \\
\qquad \textbf{in if } \rho_g\big(\sigma, A_t(o)\big)\ \ \textbf{then} \\
\qquad\qquad \textbf{let } A' = A \cup \big\{o : \big(\sigma, \Gamma \cup \{o : \sigma\}\big)\big\} \\
\qquad\qquad\quad PP(e_2, A') = (\kappa_2.\ \tau_2, \Gamma_2) \\
\qquad\qquad\quad S = unify(\{\tau_u = \tau'_u \mid\ u : \tau_u \in \Gamma_1 \text{ and } u : \tau'_u \in \Gamma_2\}, \\
\qquad\qquad\quad \Gamma = S\Gamma_1 \cup S\Gamma_2 \\
\qquad\qquad\ \textbf{in if } sat(S(\kappa_1 \cup \kappa_2), \Gamma) = \emptyset\ \ \textbf{then } \text{fail} \\
\qquad\qquad\qquad \textbf{else } \big(S(\kappa_1 \cup \kappa_2.\ \tau_2), \Gamma - \{o : S(\kappa_1.\ \tau_1)\}\big) \\
\qquad\quad \textbf{else } \text{fail}
\end{array}
$$

Fig. 4. Type inference for system CT

functions or constants to be used as arguments of other (possibly overloaded) functions.

We intend to explore the use of constrained types together with a concept of higher-order types, that is, types that are parameterised on other types, which can be constrained parameters. We think this enhances a functorial view of modules as parameterised types.

Further explorations of this system involve incorporating the concept of class types in a functional programming language, studying the implications of this concept with regards to the subtyping relation and program development, and studying its relation to intersection types [10,3].

References

[1] Damas, Luis and Robin Milner, *Principal type schemes for functional programs*, Proc. 9th ACM Symp. on Principles of Programming Languages (1982), 207–212.

[2] Hall, C., K. Hammond, S.P. Jones and P. Wadler, *Type Classes in Haskell*, Proc. 5th European Symposium on Programming (1994), 241–256, Springer LNCS 788.

[3] Jim, Trevor, *What are principal typings and what are they good for*, Conf. Record of POPL'96: the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1996, pp. 42–53.

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

[5] Mitchell, J.C., *Polymorphic type inference and containment*, Information and Computation **76** (1988), no. 2/3, 211–249.

[6] ______, *Foundations for programming languages*, MIT Press, 1996.

[7] Odersky, M., P. Wadler and M. Wehr, *A Second Look at Overloading*, Conference Record of Functional Programming and Computer Architecture (1995).

[8] Palsberg, Jens and Scott Smith, *Constrained Types and Their Expressiveness*, ACM TOPLAS **18** (1996), no. 5, 519–527.

[9] Smith, Geoffrey S., *Polymorphic Type Inference for Languages with Overloading and Subtyping*, Ph.D. thesis, Cornell University, 1991.

[10] van Bakel, S., *Essential Intersection Type Assignment*, Proc. 13th Conf. Foundations of Software Technology and Theoretical Computer Science (R.K. Shyamasunda, ed.), LNCS **761** (1993), pp. 13–23.

[11] Wadler, Philip, *How to make ad-hoc polymorphism less ad hoc*, Conf. Record of the 16th ACM Symposium on Principles of Programming Languages (1989).