Type Inference using Constraint Handling Rules

Sandra Alves, Mário Florido

University of Porto
Department of Computer Science & LIACC
R. do Campo Alegre 823, 4150-180 Porto, Portugal
e-mail: {sandra,amf}@ncc.up.pt

Abstract. In this paper we present an implementation of the general system for type inference algorithms HM(X) using Prolog and Constraint Handling Rules. In our implementation the difference between the general aspects of the type inference algorithms and the constraint resolution module becomes more clear when compared to other implementations of the same systems, usually made in a functional programming language. In the constraint module solving equality constraints, here implemented by Prolog unification, is completely separated from constraint simplification which is made by a solver implemented in CHR for each system. CHR rules become a clear and natural way of specifying the simplification mechanism.

1 Introduction

Constraint based type inference becomes more clear when implemented in a constraint programming language. Type inference algorithms for functional programming languages are based in the notion of constraint resolution. The type inference algorithm of ML [5] uses unification for solving constraints in the type language and several extensions of this algorithm are based on extensions of the unification algorithms with some kind of constraint resolution mechanism. Some examples include record systems [15] and type inference dealing with overloading [20, 12, 14]. The same happens for object oriented languages where the previous systems are extended with the notion of subtyping [2, 6, 7]. Subtyping is modeled by solving subtype constraints in the type inference process.

In this context Odersky, Sulzmann and Wehr defined the HM($\mathcal{X}$) framework, [19], as a general framework for type systems with constraints. Instantiating the parameter $\mathcal{X}$ to a specific constraint system defines a specific type system (note the relation with the CLP($\mathcal{X}$) framework for constraint logic programming [10]). They showed that the instantiations of $\mathcal{X}$ are sound under an untyped semantics and they presented a generic type inference algorithm which guarantees standard properties of type inference, such as decidability and the existence of principal types, under certain conditions on $\mathcal{X}$. The HM($\mathcal{X}$) type inference algorithm is presented in a syntax closely related with an implementation in a functional programming language, as it is usually done in the definition of type inference algorithms for functional languages.

In this paper we redefine the general algorithm using Prolog with Constraint Handling Rules (CHR), (see [8] for a survey on CHR). In this formulation of the algorithm equality constraints are specified by Prolog unification and any extensions become clear as a set of simplification rules with a declarative specification using CHR.

In this work we do not present yet another type inference system. The main contribution is to show that logic programming , as a specification language for type inference algorithms, and Constraint Handling Rules as a way of specifying explicitly the constraint solving algorithms implicitly defined by type systems, are the right way to specify and implement type inference algorithms for programming languages.

We assume that the reader is familiar with CHR and type inference for functional programming languages (see [4] for a good survey on the subject). It is also assumed that the reader is familiar with the logic programming paradigm, [11].

The rest of the paper is structured as follows: The next section presents the related work. Then we present a brief description of the HM($\mathcal{X}$) framework. In the next section we define the general type inference algorithm for the HM($\mathcal{X}$) framework as a logic program with a constraint simplification module and show how type inference for ML is trivially done as a special case of the algorithm. Then, as case studies, we present the CHR rules which implement type simplification in a record calculus, and a system dealing with overloading. Finally we outline some future work.

2 Related Work

As far as we know the only previous work which related CHR to type inference for functional programming was reported in [9]. In this paper Glynn, Stuckey and Sulzmann applied CHR to the study and implementation of several extensions of type classes. Their thesis was that *" Constraint handling rules are the right way to understand type classes constraints, and extensions to type classes"*. In our work we apply CHR to the specification of any type inference system based in extensions to unification within the HM($\mathcal{X}$) framework. The specification of type classes in [9] can be viewed as a CHR module for the simplification of type classes which can be trivially integrated in our implementation allowing to make type inference in the presence of overloading using the type classes formalism.

3 The HM($\mathcal{X}$) Framework

Here we give a brief overview of the HM($\mathcal{X}$) framework defined in [19]. Originally HM($\mathcal{X}$) has en extra rule to deal with subtyping. In our work we do not deal with subtyping thus the system is presented here without the subtyping rule. The $\mathcal{X}$ in the HM($\mathcal{X}$) framework stands for a constraint system along the lines defined in [16]. We will not present a formal definition of constraint system here. We just note that $\mathcal{X}$ is a parameter defining:

1. a set of primitive constraints;
2. an entailment relation between constraints ($\models$);
3. a projection of a constraint c onto variables $\bar{x}$. As usual, projection is denoted by the existential quantifier;
4. a set of predicates relating types. This set must contain an equality predicate.

Concerning type inference we define an instance of the general type inference framework HM($\mathcal{X}$) by $(\mathcal{X}, \mathcal{T}, \mathcal{S}, \Gamma_0)$. $\mathcal{T}$ defines the type language and $\mathcal{X}$ defines the constraint system. The set $\mathcal{S}$ defines the set of valid constraints used in the type schemes and in type derivations. Γ_0 is the set with the initial type declarations.

3.1 The term language

The term language is just the λ-calculus ([3]) with the additional term *let*

$$M ::= x \mid \lambda x.M \mid MM' \mid \text{let } x = M \text{ in } M'$$

3.2 The type language

The type language is defined by $\mathcal{T}$. Let α range over an infinite set of type variables, τ and τ' denote types and σ denote type schemes. Then the type language is defined as follows:

$$\begin{aligned} \tau &::= \alpha \mid \tau \to \tau' \mid T\bar{\tau} \\ \sigma &::= \tau \mid \forall \alpha.C \Rightarrow \sigma \end{aligned}$$

T stands for other type constructors defined in $\mathcal{T}$. Those constructors depend on the particular instance of HM($\mathcal{X}$). Type schemes are of the form $\forall \alpha.C \Rightarrow \sigma$, where C is a set of constraints. In this framework $\{C_1, \ldots, C_n\}$ is equivalent to $C_1 \wedge \cdots \wedge C_n$.

3.3 The type system

Let C be satisfiable in $\mathcal{X}$, Γ one set of type declarations where for each term variable there is at most one type declaration, and σ range over type schemes. We define $C * \Gamma \vdash M : \sigma$ by the following rules:

$$\textit{(Var)} \qquad C * \Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$$

$$\textit{(Abs)} \qquad \frac{C * \Gamma \cup \{x : \tau\} \vdash M : \tau'}{C * \Gamma \vdash \lambda x.M : \tau \to \tau'}$$

$$\textit{(App)} \qquad \frac{C * \Gamma \vdash M : \tau_1 \to \tau_2 \quad C * \Gamma \vdash M' : \tau_1}{C * \Gamma \vdash M M' : \tau_2}$$

$$\textit{(Let)} \qquad \frac{C * \Gamma \vdash M : \sigma \quad C * \Gamma \cup \{x : \sigma\} \vdash M' : \tau'}{C * \Gamma \vdash \text{let } x = M \text{ in } M' : \tau'}$$

$$\textit{(}\forall\textit{ Intro)} \qquad \frac{C \cup D * \Gamma \vdash M : \tau \quad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \cup \{\exists \bar{\alpha}.D\} * \Gamma \vdash M : \forall \bar{\alpha}.D \Rightarrow \tau}$$

$$\textit{(}\forall\textit{ Elim)} \qquad \frac{C * \Gamma \vdash M : \forall \bar{\alpha}.D \Rightarrow \tau' \quad C \models [\bar{\tau}/\bar{\alpha}]D}{C * \Gamma \vdash M : [\bar{\tau}/\bar{\alpha}]\tau'}$$

The type $\forall \bar{\alpha}.D \Rightarrow \tau$ is a different notation for $\forall \alpha_1.\text{true} \Rightarrow \ldots \forall \alpha_n.D \Rightarrow \tau$, and $\exists \bar{\alpha}.D$ for $\exists \alpha_1. \ldots \exists \alpha_n.D$. A type derivation $C * \Gamma \vdash M : \sigma$ is valid if C is satisfiable.

We can restrict the constraints in type schemes and type derivations to constraints in *solved form*. The set $\mathcal{S}$, of constraints in *solved form*, is a subset of the set of satisfiable constraints in $\mathcal{X}$.

The set $\mathcal{S}$ determines the valid typings in HM($\mathcal{X}$). For example, if $\mathcal{S}$ is the empty set then we cannot type any term.

4 Normalization

Here we present the concept of constraint normalization, along the lines presented in [18].

In [18] substitution $\phi = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$, is viewed as the constraint set $\{\alpha_1 = \tau_1, \ldots, \alpha_n = \tau_n\}$, and the application of substitution ϕ to a constraint C is viewed as the constraint $\exists \alpha_1 \ldots \alpha_n.(C \wedge (\alpha_1 = \tau_1) \wedge \cdots \wedge (\alpha_n = \tau_n))$.

Definition 1 *Let C, D and ϕ be constraints in $\mathcal{X}$, then $C \wedge \phi$ is a normal form of D iff $C \in \mathcal{S}$ (being $\mathcal{S}$ the set of constraints in solved form), and $C \wedge \phi \models D$.*

Definition 2 *Let C, C', D, ϕ and ϕ' be constraints in $\mathcal{X}$, then $C \wedge \phi$, is the principal normal form of D, if for all normal forms $C' \wedge \phi'$ of D, $\exists U.(C' \wedge \phi') \models \exists U.(C \wedge \phi)$ where $U = fv(C \wedge C' \wedge \phi \wedge \phi') \setminus fv(D)$.*

Example 1 *Let Herbrand be the constraint system where primitive constraints are of the form $\tau_1 = \tau_2$ where τ_1, τ_2 are types. The equality predicate is syntactic equality, and entailment is checked by a matching algorithm. For example, $\beta \to \gamma = \alpha \to (\alpha \to \beta)$ entails $\beta = \alpha$ and $\gamma = \alpha \to \beta$. Let $\mathcal{S} = true$ here represented by $\emptyset$. Then normal forms correspond to unifiers and principal normal forms to most general unifiers.*

Definition 3 *The function normalize from constraints to normal forms is defined as:*

$$\begin{aligned} normalize(D) &= C \wedge \phi \text{ if } C \wedge \phi \text{ is the principal normal form of } D \\ &= fail \quad \text{if no normal form exists} \end{aligned}$$

Normalization can be viewed as an extension of constraint solving and unification. In our implementation unification is done by the builtin unification of Prolog, and normalization corresponds to constraint solving in CHR.

Definition 4 *A constraint system $\mathcal{X}$ has the principal constraint property, if for every constraint C in $\mathcal{X}$, either C has a principal normal form, or C does not have a normal form.*

5 The Type Inference Algorithm

In [19] it is defined a general algorithm for inferring types in the HM($\mathcal{X}$) framework. This algorithm is proved to be sound with respect to an untyped semantics and to derive principal types if $\mathcal{X}$ has the principal constraint property.

Here we present our specification of the algorithm using Prolog and CHR. Equality constraints are solved using Prolog unification with an occur-check. Each different instance will correspond to a different module in CHR to normalize constraints which were not solved by unification. Thus to each type system corresponds a different set of CHR rules, and those rules are the only differences between different type systems within HM($\mathcal{X}$) (except for possible differences in the type and term languages).

In the definition of the type inference algorithm and in the rest of the paper, by an abuse of notation, we use an abstract syntax for types ($\tau, \forall\bar{\alpha}.D \Rightarrow \tau, \ldots$) instead of the proper syntax for Prolog terms.

Definition 5 *Given a set of type declarations Γ, and a term M, the algorithm gives as an output the type τ for M, and the set of constraints in solved form C. The type inference algorithm is defined by the following set of Horn clauses:*

$$\begin{aligned} type(C, \Gamma, x, \tau) \leftarrow\ & member(x, \Gamma, \forall\bar{\alpha}.D \Rightarrow \tau_1), \\ & new\_instance(\bar{\alpha}, \tau_1, \tau, D_1, D), \\ & normalize(D_1, C). \end{aligned}$$

$$type(C, \Gamma, \lambda x.M, \alpha \to \tau) \leftarrow type(C, [(x, \forall\emptyset.\mathrm{true} \Rightarrow \alpha) \mid \Gamma], M, \tau).$$

$$\begin{aligned} type(D, \Gamma, M_1 M_2, \tau) \leftarrow\ & type(C_1, \Gamma, M_1, \tau_1), \\ & type(C_2, \Gamma, M_2, \tau_2), \\ & append(C_1, C_2, C), \\ & unify\_with\_occurs\_check(\tau_1, \tau_2 \to \tau), \\ & normalize(C, D). \end{aligned}$$

$$
\begin{aligned}
type(C, \Gamma, \text{let } x = M_1 \text{ in } M_2, \tau) \leftarrow\ & type(C_1, \Gamma, M_1, \tau_1), \\
& gen(C_1, \Gamma, \tau_1, C_2, \sigma), \\
& type(C_3, [(x, \sigma) \mid \Gamma], M_2, \tau), \\
& append(C_2, C_3, D), \\
& normalize(D, C).
\end{aligned}
$$

We assume the notation $\forall\emptyset.true => \tau$ for simple types. This avoids having different clauses for simple types and type schemes.

The predicate *normalize* is defined as follows:

$$
\begin{aligned}
normalize([], C) \leftarrow\ & !, findall\_constraints(\_, CId), \\
& removeId(CId, C).
\end{aligned}
$$

$$
\begin{aligned}
normalize([X|R1], C) \leftarrow\ & call(X), \\
& normalize(R1, C).
\end{aligned}
$$

The first argument of normalize is a set of constraints. These are the constraints which are going to be solved by CHR, and it is the only point where the type inference algorithm differs from one type system to another.

The CHR builtin function, *findall_constraints(Pattern,List)*, unifies *List* with a list of *Constraint # Id* pairs from the constraint store that match *Pattern*. When we call *findall_constraints(_,List)*, we just collect all constraints.

The *removeId(CId,C)* predicate scans the list *CId* of pairs of the form *Constraint # Id* and for each pair removes the second element.

Let us briefly explain the predicate *gen* used in the *Let* rule.

In $gen(C, \Gamma, \sigma, D \cup \{\exists\bar{\alpha}.C'\}, \forall\bar{\alpha}.C' \Rightarrow \sigma)$, $C \in \mathcal{S}$ is a set of constraints, Γ is a set of type declarations, σ is a type scheme and $fv(D) \cap \bar{\alpha} = \emptyset$. The set $\bar{\alpha}$ is a subset of $(fv(\sigma) \cup fv(C)) \setminus fv(\Gamma)$.

This predicate splits C in two sets, D and C'. The generalized variables appear only in C'.

Finally in the predicate $new\_instance(\bar{\alpha}, \tau_1, \tau, D_1, D)$, $\bar{\alpha}$ is a set of type variables, τ_1 a type, D_1 a set of constraints, τ is a type and D a set of constraints, such that $\tau = [\bar{\beta}/\bar{\alpha}]\tau_1$ and $D = [\bar{\beta}/\bar{\alpha}]D_1$ ($\bar{\beta}$ are new variables).

6 Some instances of HM($\mathcal{X}$)

In this section we present instances of the HM($\mathcal{X}$) framework by defining the CHR rules used in the normalization of type constraints. The systems presented are the Damas-Milner [5] type system, the Ohori type system for extensible records reported in [15], and the system O for dealing with overloading [14]. Those systems were used in [17] to illustrate the use of specific constraint systems for type inference. Here we show that when compared to other languages CHR gives a clear and direct implementation of the entailment relation defined in the constraint system as a set of CHR rules.

6.1 The Damas-Milner Type System

Type inference for the type system of the programming language ML, also known as Damas-Milner type system [5], is obtained just by defining type schemes as $\forall\alpha.\{\} \Rightarrow \sigma$, which correspond to $\forall\alpha.\sigma$ in the original system, and the satisfiable constraints equal to *true*, here represented by the empty set. In this type system every constraint is solved by unification.

Example 2 *Consider the program:*

$$
\begin{array}{l}
f \;=\; \mathbf{let} \\
\qquad\quad g\ x \;=\; x \\
\qquad \mathbf{in} \\
\qquad g\ g
\end{array}
$$

Applying the type inference algorithm to $g = (\lambda x.x)$ *we have:*

$$
\begin{array}{ll}
type(C, \Gamma_0, (\lambda x.x), \alpha \rightarrow \tau) & \\
\quad type(C, [(x, \alpha) \mid \Gamma_0], x, \tau) & \\
\quad\quad member(x, [(x, \alpha) \mid \Gamma_0], \beta) & \\
\quad\quad new\_instance(\{\}, \alpha, \tau, \{\}, D) & \tau = \alpha, D = \{\} \\
\quad\quad \boldsymbol{normalize}(\{\}, C) & C = \{\}
\end{array}
$$

Thus,

$$type(C, \Gamma_0, (\lambda x.x), \alpha \rightarrow \alpha)$$

If we apply the type inference algorithm to let $g = (\lambda x.x)$ *in gg*

$$
\begin{array}{ll}
type(C, \Gamma_0, let\ g = (\lambda x.x)\ in\ gg, \tau) & \\
\quad type(C_1, \Gamma_0, (\lambda x.x), \tau_1) & \tau_1 = \alpha \rightarrow \alpha, C_1 = \{\} \\
\quad gen(C_1, \Gamma_0, \alpha \rightarrow \alpha, C_2, \sigma) & C_2 = \{\}, \sigma = \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha \\
\quad type(C, [(g, \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], gg, \tau) & \\
\quad\quad type(C_3, [(g, \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], g, \tau_3) & \\
\quad\quad\quad member(g, [(g, \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) & \\
\quad\quad\quad new\_instance(\alpha, \alpha \rightarrow \alpha, \tau_3, \{\}, D_3) & \tau_3 = \alpha_1 \rightarrow \alpha_1, D_3 = \{\} \\
\quad\quad\quad \boldsymbol{normalize}(\{\}, C_3) & C_3 = \{\} \\
\quad\quad type(C_4, [(g, \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], g, \tau_4) & \\
\quad\quad\quad member(g, [(g, \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) \mid \Gamma_0], \forall\alpha.\{\} \Rightarrow \alpha \rightarrow \alpha) & \\
\quad\quad\quad new\_instance(\alpha, \alpha \rightarrow \alpha, \tau_4, \{\}, D_4) & \tau_4 = \alpha_2 \rightarrow \alpha_2, D_4 = \{\} \\
\quad\quad\quad \boldsymbol{normalize}(\{\}, C_4) & C_4 = \{\} \\
\quad\quad union(\{\}, \{\}, D) & D = \{\} \\
\quad\quad unify\_with\_occurs\_check(\alpha_1 \rightarrow \alpha_1, \alpha_2 \rightarrow \alpha_2 \rightarrow \alpha_3) & \alpha_1 = \alpha_2 \rightarrow \alpha_2, \\
 & \alpha_3 = \alpha_2 \rightarrow \alpha_2 \\
\quad\quad \boldsymbol{normalize}(\{\}, C) & C = \{\} \\
\quad\quad \tau = \alpha_3 & \tau = \alpha_2 \rightarrow \alpha_2
\end{array}
$$

Thus,

$$type(\{\}, \Gamma_0, let\ g = (\lambda x.x)\ in\ gg, \alpha_2 \rightarrow \alpha_2)$$

The types inferred for g and f with initial set of type declarations $\Gamma_0 = \emptyset$, are:

$$f :: \forall \alpha_2 . \{\} \Rightarrow \alpha_2 \rightarrow \alpha_2$$
$$g :: \forall \alpha . \{\} \Rightarrow \alpha \rightarrow \alpha$$

6.2 Records

Here we present an instance of the HM($\mathcal{X}$) framework by defining the CHR rules used in the normalization of type constraints for the Ohori type system.

The type language $\mathcal{T}$ The types in $\mathcal{T}$ are defined by:

$$\tau ::= \alpha \mid \tau \rightarrow \tau' \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

The type $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ defines records with label fields $l_1, \ldots, l_n$, which have types $\tau_1, \ldots, \tau_n$.

The Constraint System $\mathcal{R}$ In HM($\mathcal{R}$) type variables in type schemes are constrained by a *kind* k. A *kind* k is defined as $\langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle$ and defines the records which contain at least the fields $l_1, \ldots, l_n$.

Thus the primitive constraints are of the form $(\tau :: k)$, where τ is a type and k is a *kind*.

Let $rectype(R)$ ($R = [(l_1, \tau_1), \ldots, (l_n, \tau_n)]$) denote the type $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, and $labtype(L, \tau_i)$ represent the *kind* $\langle L : \tau_i\rangle$. The normalization rules for $\mathcal{R}$ are the following:

$$records1\ @\ rectype(R) :: labtype(L, \tau_i) \Longleftrightarrow member(L, R, \tau_i) \mid true.$$

$$records2\ @\ \tau :: labtype(L, \tau_1), \tau :: labtype(L, \tau_2) \Longrightarrow \tau_1 = \tau_2.$$

$$records3\ @\ rectype(R) :: labtype(L, \tau_2) \Longrightarrow member(L, R, \tau_1) \mid \tau_2 = \tau_1.$$

$$records4\ @\ exists(\alpha, (\alpha :: K)) \Longleftrightarrow fv(K, V), notin(\alpha, V) \mid true.$$

The first rule states that for any l_i, τ_i $i = 1 \ldots n$ we have $\{l_1 : \tau_1, \ldots, l_n : \tau_n\} :: \langle l_i : \tau_i\rangle$. Rules 2 and 3 avoid overloading for fields. One record cannot have different types for the same field.

As defined in [19] constraints in solved form are the satisfiable constraints of the form

$$C ::= \text{true} \mid (\alpha :: \langle l : \tau\rangle) \mid C \wedge C \mid \exists\bar{\alpha}.C$$

The initial set of declarations Γ_0 has the primitive constructors for the creation, selection and modification of records.

Let $l_1, \ldots, l_n$ be a sequence of fields, and $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ the type representing a record with those fields. We define the constructor $l_1\_\ldots\_l_n$ and add to Γ_0:

$$l_1\_\ldots\_l_n : \tau_1 \to \cdots \to \tau_n \to \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

$l_1\_\ldots\_l_n e_1 \ldots e_n$ creates a new record $\{l_1 = e_1, \ldots, l_n = e_n\}$.

For each field l we add to Γ_0 the following type declarations:

$$\_.l : \forall \alpha, \beta.(\alpha :: \langle l : \beta \rangle) \Rightarrow \alpha \to \beta$$

for field access and

$$modify_l : \forall \alpha, \beta.(\alpha :: \langle l : \beta \rangle) \Rightarrow \alpha \to \beta \to \alpha$$

for field modification.

Example 3 *Consider the following program:*

$$\begin{array}{l} f\ x\ =\ \textbf{let} \\ \qquad\qquad g\ x\ y\ =\ \{l_1\ =\ x,\ l_2\ =\ y\} \\ \qquad\quad \textbf{in} \\ \qquad\quad modify(g\ 1\ 2, l_1, x) \end{array}$$

The types inferred for f and g, with the initial type declaration $\{eq : \forall \alpha.\{\} \Rightarrow \alpha \to \alpha \to bool, 1 : int, 2 : int\}$, are respectively:

$$\begin{array}{l} f\ ::\ int\ \to\ \{l_1\ :\ int,\ l_2\ :\ int\} \\ g\ ::\ \forall\,\alpha,\ \beta\,.\{\}\ \Rightarrow\ \alpha\ \to\ \beta\ \to \{l_1\ :\ \alpha,\ l_2\ :\ \beta\} \end{array}$$

Example 4 *Consider the following function:*

$$f\ x\ y\ =\ eq\ x.l\ y.l$$

The type inferred for f, with the initial type declaration $\{eq : \forall \alpha.\{\} \Rightarrow \alpha \to \alpha \to bool\}$, is:

$$f\ :: \forall\,\alpha,\ \beta,\ \gamma\,.(\alpha\ ::\ \langle\ l\ :\ \beta\rangle\ \wedge\ \gamma\ ::\ \langle\ l\ :\ \beta\ \rangle\ )\ \Rightarrow \alpha\ \to\ \gamma\ \to\ bool$$

6.3 System O

Here we define a set of CHR rules for normalization of type constraints in system O [14], as instance of HM($\mathcal{X}$). System O deals with a restricted form of overloading.

In this system, each overloaded operator must have a type of the form $\alpha \to \tau$, where τ may itself involve α. That is, overloaded operators always have a functional type where overloading is restricted to the first argument.

With this simple restriction system O avoids ambiguity problems which may arise in the Haskell type class formalism [13].

The type language $\mathcal{T}$ The type language $\mathcal{T}$ is defined by:

$$\tau ::= \alpha \mid \tau \to \tau \mid T\tau_1 \ldots \tau_n$$

where T represents a type constructor of arity n. For instance, in this type language we consider basic types, such as Int, to be type constructors of arity 0.

The Constraint System $\mathcal{O}$ In $\mathcal{O}$ we have two different kinds of constraints. We have constraints of the form $(o :: \alpha \to \tau)$ that restrict the possible instances of a type variable α form a given overloaded identifier o; and constraints of the form $o :: \sigma_T$, used to describe instance declarations of overloaded identifiers. The type scheme σ_T must satisfy the following conditions:

$$\begin{array}{rll} \sigma_T ::= & T\bar{\alpha} \to \tau & fv(\tau) \subseteq \bar{\alpha} \\ \mid & \forall \alpha.\pi_\alpha \Rightarrow \sigma'_T & fv(\pi_\alpha) \subseteq fv(\sigma'_T) \end{array}$$

where the constraint π_α only contains predicates of the form $(o :: \alpha \to \tau)$. We assume that σ_T is a closed type scheme. The normalization rules for $\mathcal{O}$ are the following:

$$\begin{array}{l} over1 \ @ \ (o :: arrow(\alpha, \tau_1)), (o :: arrow(\alpha, \tau_2)) \Longrightarrow \\ \qquad unify\_with\_occurs\_check(\tau_1, \tau_2). \end{array}$$

$$\begin{array}{l} over2 \ @ \ (o :: arrow(T\bar{\tau_1}, \tau_1)), (o :: arrow(T\bar{\tau_2}, \tau_2)) \Longrightarrow \\ \qquad unify\_with\_occurs\_check(\bar{\tau_1}, \bar{\tau_2}), \\ \qquad unify\_with\_occurs\_check(\tau_1, \tau_2). \end{array}$$

$$\begin{array}{l} over3 \ @ \ (o :: forall(\alpha, D, \tau)), D_1 \Longrightarrow replace(D, [(\alpha, T)], D_1), \\ \qquad replace(\tau, [(\alpha, T)], \tau_1) \mid (o :: \tau_1). \end{array}$$

$$over4 \ @ \ exists(\alpha, (o :: arrow(\alpha, \tau))) \Longleftrightarrow true.$$

Rules 1 and 2 ensure the fact that the argument type uniquely determines the result type. Rule 3 implements the instance (or $\forall$ Elim) rule. Finally the last rule implements an *open world* approach, where one does not have to declare instances for every overloaded operator.

For each overloaded identifier o the following type declaration:

$$o : \forall \alpha, \beta.(o :: \alpha \to \beta) \Rightarrow \alpha \to \beta$$

is added to the initial environment Γ_0. These type declarations added to the initial environment, provide the restrictions imposed on overloaded operators in system O.

In HM($\mathcal{O}$) the term language is extended with instance declarations. The extended term language is as follows:

$$M ::= x \mid MM \mid \lambda x.M \mid let\ x = M\ in\ M$$

$$P\ ::= M \mid inst\ o : \sigma_T = M\ in\ P$$

In the instance declaration $inst\ o : \sigma_T = M\ in\ P$, the meaning of the identifier o is overloaded with the function given by M.

In [14] the type system is extended with an extra rule for instances. Assuming that the instance declarations are available before the type inference process we do a preprocessing stage where, for each instance declaration $inst\ o : \sigma_T = M\ in\ P$, we typecheck $M : \sigma_T$, and then add the constraint $(o :: \sigma_T)$ to the CHR constraint store.

Thus, each instance of an overloaded operator defines an axiom of the form $(o :: \sigma_T)$ where σ_T is the type declaration for the corresponding instance of o.

Example 5 *Consider the following program:*

$inst + \ :\ int \rightarrow int \rightarrow int\ =\ plus\_int$
$inst + \ :\ float \rightarrow float \rightarrow float\ =\ plus\_float$
$f\ =\ 3\ +\ toInt(2.4\ +\ 2.5)$

Where plus_int and plus_float are the sum for integers and for floats, respectively. The type inferred for f*, with the initial environment* $\Gamma_0 = \{toInt : float \rightarrow int, + : \forall\alpha,\beta.(+ :: \alpha \rightarrow \beta) \Rightarrow \alpha \rightarrow \beta, 3 : int, 2.4 : float, 2.5 : float\}$*, is:*

$f\ ::\ int$

7 Future Work

There are two topics which are left for future work:

1. There is a close connection between some properties of CHR programs, such as confluence and termination ([1]), and standard properties of type inference systems, such as decidability and existence of principal types. By showing directly some properties for the CHR program which specifies each type system we would get for free properties of the type system. That would be another gain in using CHR for the specification of type constraints simplification.
2. Applying CHR to the specification of subtyping. Previous work about subtyping constraints simplification are constraint-based and thus CHR seems to be a promising framework for this topic.

Acknowledgements We thank Martin Odersky and Martin Sulzmann for their comments on some aspects of HM($\mathcal{X}$). The work presented in this paper has been partially supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*.

References

1. Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Principles and Practice of Constraint Programming*, pages 252–266, 1997.
2. A. Aiken and E.L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architechture*, 1993.
3. H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
4. Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.
5. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 207–212, 1982.
6. J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to object oriented programming. *Eletronic Notes in Theoretical Computer Science*, 1, 1995.
7. Pottier F. Simplifying subtyping constraints. In *International Conference on Functional Programming*, 1996.
8. T. Fruhwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 37, 1998.
9. K. Glynn, P. Stuckey, and Sulzmann M. Type classes and constraint handling rules. In *Workshop on Rule-Based Constraint Reasoning and Programming*, 2000.
10. Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Conference record of the 14th annual ACM symposium on principles of programming languages*, pages 111 – 119, Munich, Germany, 1987.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
12. Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*. ACM Press, 1993.
13. Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.
14. Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming & Computer Architecture*, San Diego, California, June 1995. ACM Press.
15. Ohori. A polymorphic record calculus and its compilation. *ACM TOPLAS*, 6, 1995.
16. Vijay A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.
17. Martin Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.
18. Martin Sulzmann. A General Type Inference Framework for Hindley/Milner Style Systems. In *5th International Symposium on Functional and Logic Programming (FLOPS)*, Tokyo, Japan, March 2001.
19. Martin Sulzmann, Martin Odersky, and Martin Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
20. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16'th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1989.