Type Inference for Programming Languages: A Constraint Logic Programming Approach

Sandra Alves and Mário Florido

Technical Report Series: DCC-2004-5

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6078830 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Type Inference for Programming Languages: A Constraint Logic Programming Approach

Sandra Alves and Mário Florido
DCC-FC & LIACC
University of Porto, Portugal
{sandra,amf}@ncc.up.pt

Abstract

In this paper we present an application of Constraint Logic Programming to the design and implementation of type inference algorithms for programming languages. We present implementations in Prolog and Constraint Handling Rules (CHR) of several algorithms which belong to the state of the art of type inference for programming languages: the Damas-Milner type system, the Ohory system for labeled records and the Rank-2 Intersection Type system. In our implementation the differences between the general aspects of the type inference algorithms and the constraint resolution modules become more clear, when compared to other implementations of the same systems, usually made in a functional programming language. In the constraint modules, solving equality constraints, here implemented by Prolog unification, is completely separated from constraint simplification, which is made by solvers implemented in CHR for each system. Constraint Logic Programming revealed to be a highly declarative specification and implementation language for type inference algorithms.

1 Introduction

The advantages of strong static typing, where types are inferred by the compiler at compile time, are now generally recognized. As a matter of fact languages such as Haskell, ML, Mercury or Java all rely on strong typing. In order to gain programming flexibility avoiding rejecting perfectly safe programs, the type inference algorithm should support *type polymorphism*, i.e. it should allow programs to possess more than one type. Two main options for polymorphism are universal types, where types are parametrized (this concept is known as *parametric polymorphism*), and intersection types which introduce an operator of intersection over types (their duals are the existential types and union types).

The most popular type inference algorithm is the Damas-Milner algorithm [10] which supports parametric polymorphism with universal types. In the area of type systems for programming languages there was a long search for systems more expressive than the Damas-Milner type system and with decidable typability. In this search there have been a few positive results in two distinct directions: extensions of the Damas-Milner approach and systems based on intersection types [7, 18].

Type inference in all these systems rely on some notion of constraint resolution. The type inference algorithm of the Damas-Milner system (used as the basis of the type systems of functional programming languages such as ML and Haskell), uses unification to solve constraints in the type language. 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 [23] type inference dealing with overloading [35, 21, 22] and subtyping [1, 12, 24]. Type inference for decidable fragments of intersection types also rely on constraint resolution methods [16, 18].

Sulzmann and Wehr defined the HM($\mathcal{X}$) framework, [32], as a general framework for extensions of the Damas-Milner 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 [15]). 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 was 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.

The first part of this paper is a revised version of a previous work of the authors reported in [2], where it is presented an implementation the general type inference algorithm of HM($\mathcal{X}$) using Prolog with Constraint Handling Rules (CHR) [13]. 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. Dealing only with equality constraints we get an implementation of the paradigmatic polymorphic type system of *pure ML*. We also present an extension of the general algorithm to deal with records. In this implementation the constraint normalization process has a very high level declarative implementation using CHR.

In the second part of the paper we present an implementation in Prolog and CHR of the most used decidable fragment of intersection types, the Rank-2 Intersection Type System [16, 34]. Intersection type systems provide what is generally called discrete polymorphism [19] and type more programs than the Damas-Milner type system. They provide more flexibility to the type system avoiding, in many cases, the contortions that programmers sometimes have to do to convince the compiler that their programs are well-typed. The price to pay for this flexibility is that intersection type systems are rather complex, with definitions that can be quite difficult to understand. Also for the intersection type inference implementation presented in this paper, logic programming and CHR revealed to be a quite powerful framework leading to a high declarative and understandable implementation. As far as we know, this is the first work which uses a Constraint Programming Language in the implementation of type inference algorithms for intersection type systems.

Let us present an example clarifying the role of constraint resolution in the type inference process.

Example 1 *Consider a simple declaration like*

$$compose\ f\ g\ =\ \lambda\, x.f(g(x))$$

which defines function composition. Considering that the types of f*,* g *and* x *are, respectively,* α_1*,* α_2 *and* γ*, typing the expression*

$$f(g(x))$$

generates the constraints:

$$\{\alpha_2 = \gamma \to \alpha_3, \alpha_1 = \alpha_3 \to \alpha_4\}$$

being α_3*,* α_4 *respectively the types for* $g(x)$ *and* $f(g(x))$*. Solving those constraints using Robinson unification [26] algorithm as the constraint solver one gets the following type for compose:*

$$compose\ ::\ (\alpha_3\ \to\ \alpha_4)\ \to\ (\gamma\ \to\ \alpha_3)\ \to\ \gamma\ \to\ \alpha_4$$

In this work we do not present yet another type inference system. The main contribution is to show that constraint logic programming provides a powerful declarative way to specify and implement type inference algorithms for programming languages. This work sustains our thesis that type inference becomes more clear when implemented in a constraint logic programming language.

Let us now present the related work: the use of CHR for checking the satisfiability of subtype inequalities was reported in [9]. CHR for type inference with Haskell as host language was presented in [28]. The study and implementation of systems dealing with overloading employing CHR was done in [14] and in [29]. The CHR specifications presented in these works can be easily integrated in our implementation as new CHR modules, allowing to make type inference in the presence of overloading.

Both approaches, using Haskell as a host language and ours which uses Prolog, show the ease of use and declarative character of axiomatic based constraint simplification mechanisms (such as CHR) for the specification of extensions of unification in type inference algorithms. The main differences between the two approaches are essentially related to differences in the two paradigms used as host languages. The approaches based on functional languages implemented a standard substitution-based

definition of unification. Note that Prolog unification (based on a graph representation of terms) is highly optimized. This makes Prolog implementations of type inference using built-in unification quite efficient and at the same time more declarative, having propagation of substitutions for free. Finally, note that an elegant way to specify type inference algorithms is by using a logical system consisting of a set of rules defined by induction on the program structure. As it is shown in this paper these rules can be directly coded as Horn clauses where the head defines the type of the expression and the body the types of its subexpressions.

We assume that the reader is familiar with the logic programming paradigm [20]. A good survey about type inference for programming languages can be found in [5].

The rest of the paper is structured as follows: The next section presents some preliminary notions. In section 3 we talk about parametric polymorphism and present a brief description of the Damas-Milner type system. In section 4 we describe succinctly the HM($\mathcal{X}$) framework and define the general type inference algorithm for the HM($\mathcal{X}$) framework, as a logic program with a constraint simplification module. As case studies, we show how type inference for ML is trivially done as a special case of the algorithm, and present the CHR rules which implement type simplification in a record calculus. In section 5 we present the Rank 2 Intersection Types System, and implement an inference algorithm for this system also using logic programming and CHR. Finally we conclude.

2 Preliminaries

2.1 Constraint Handling Rules (CHR)

Constraint Handling Rules [13] are a high-level language designed to write constraint solvers. It allows us to add executable specifications of a constraint theory (the *user-defined* constraints) to a given host language, such as Prolog, Java or Haskell. Those *user-defined* constraints are handled by a *user-defined* set of CHR rules.

Definition 2 *A CHR program is a finite set of guarded rules. The two most important rules (the only rules used in our application) are:*

$$\begin{array}{llcl} (\textit{simplification}) & H_1, \ldots, H_i & \Longleftrightarrow & G_1, \ldots, G_j \mid B_1, \ldots, B_k \\ (\textit{propagation}) & H_1, \ldots, H_i & \Longrightarrow & G_1, \ldots, G_j \mid B_1, \ldots, B_k \end{array}$$

with $i > 0, j \geq 0, k \geq 0$, and where $H_1, \ldots, H_i$ is a nonempty sequence of CHR (user-defined) *constraints, the guard $G_1, \ldots, G_j$ is a sequence of built-in* (predefined) *constraints, and $B_1, \ldots, B_k$ is a sequence of built-in and CHR constraints.*

Basically, *simplification* rewrites constraints preserving logical equivalence, and *propagation* adds new constraints that may cause further simplifications.

Empty sequences are represented by the built-in constraint *true*. The empty guard, *true*, can be omitted.

2.1.1 Operational Semantics

The *operational semantics* of CHR programs is given by a transition system. Through computational steps, one can proceed from one state to another. A state is a tuple

$$\langle F, E, D \rangle$$

where F is a conjunction of CHR and built-in constraints, E is a conjunction of CHR constraints and D is a conjunction of built-in constraints. In a state $\langle F, E, D \rangle$, F are the constraints that remain to be solved, and D and E are the constraints that have been accumulated and simplified so far.

Definition 3 *Let P be a* CHR *program for the CHR constraints and CT a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for* CHR *is defined thus:*

$$\begin{array}{ll}
(\textit{solve}) & \langle C \wedge F, E, D\rangle \longmapsto \langle F, E, D'\rangle \\
& \textit{if } C \textit{ is a built-in constraint and } CT \models (C \wedge D) \leftrightarrow D' \\
(\textit{introduce}) & \langle H \wedge F, E, D\rangle \longmapsto \langle F, H \wedge E, D\rangle \\
& \textit{if } H \textit{ is a CHR constraint} \\
(\textit{simplify}) & \langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle \\
& \textit{if } (H \Leftrightarrow G \mid B) \textit{ in } P \textit{ and } CT \models D \rightarrow \exists \bar{x}\ (H = H' \wedge G) \\
(\textit{propagate}) & \langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, H' \wedge E, H = H' \wedge D\rangle \\
& \textit{if } (H \Rightarrow G \mid B) \textit{ in } P \textit{ and } CT \models D \rightarrow \exists \bar{x}\ (H = H' \wedge G)
\end{array}$$

Definition 4 *A computation of a conjunction of constraints G, is a sequence of states $S_0, S_1, \ldots,$ with $S_i \mapsto S_{i+1}$, beginning with the initial state $S_0 = \langle G, \text{true}, \text{true}\rangle$, and ending with a final state or diverging. A finite computation is* successful *if it ends with a final state of the form $\langle \text{true}, E, D\rangle$, and it is* failed *if it ends with a final state of the form $\langle F, E, \text{false}\rangle$.*

Let us now present an example:

Example 5 *For the following CHR program, specifying the relation* `=<`*:*

```
reflexivity  @ X =< Y <=> X = Y | true.
antisymmetry @ X =< Y, Y =< X <=> X = Y.
transitivity @ X =< Y, Y =< Z ==> X =< Z.
```

the computation of $A \leq B \wedge C \leq A \wedge B \leq C$ is:

$$\begin{array}{ll}
 & \langle A \leq B \wedge C \leq A \wedge B \leq C, \text{true}, \text{true}\rangle \\
\longmapsto_{\textit{introduce}} & \langle \text{true}, A \leq B \wedge C \leq A \wedge B \leq C, \text{true}\rangle \\
\longmapsto_{\textit{propagate}} \textbf{Transitivity} & \langle C \leq B, A \leq B \wedge C \leq A \wedge B \leq C, \text{true}\rangle \\
\longmapsto_{\textit{introduce}} & \langle \text{true}, A \leq B \wedge C \leq A \wedge B \leq C \wedge C \leq B, \text{true}\rangle \\
\longmapsto_{\textit{simplify}} \textbf{Antisymmetry} & \langle B = C, A \leq B \wedge C \leq A, \text{true}\rangle \\
\longmapsto_{\textit{solve}} & \langle \text{true}, A \leq B \wedge C \leq A, B = C\rangle \\
\longmapsto_{\textit{simplify}} \textbf{Antisymmetry} & \langle A = B, \text{true}, B = C\rangle \\
\longmapsto_{\textit{solve}} & \langle \text{true}, \text{true}, A = B \wedge B = C\rangle
\end{array}$$

2.2 The Lambda Calculus

In 1932, Alonzo Church [6], defined the the λ-calculus as a model of the computable functions. Lambda calculus has been used as the basis of intermediate languages in functional programming languages compilers (see [4] about the use of λ-calculus in computer science).

In this section we will briefly present some basic concepts concerning λ-calculus which are needed for the comprehension of this work (see [3] for a survey on the subject).

Definition 6 *Let $\mathcal{V}$ be an infinite set of variables. The set of λ-terms, Λ is constructed inductively from $\mathcal{V}$ in the following way:*

$$\begin{array}{lrcl}
(\textit{Variable}) & x \in \mathcal{V} & \Rightarrow & x \in \Lambda \\
(\textit{Application}) & M, N \in \Lambda & \Rightarrow & (MN) \in \Lambda \\
(\textit{Abstraction}) & M \in \Lambda,\ x \in \mathcal{V} & \Rightarrow & (\lambda x.M) \in \Lambda
\end{array}$$

Considering application to be left associative, and abstraction to be right associative, we use the following abbreviations to simplify notation:

- $(M_1 M_2 \ldots M_n)$ for $(\ldots(M_1 M_2)\ldots M_n)$
- $(\lambda x_1 x_2 \ldots x_n.M)$ for $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots)))$

Definition 7 *Given $M \in \Lambda$, the set $FV(M)$ of all free variables in M is defined inductively by:*

$$\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(MN) & = & F(M) \cup FV(N) \\
FV(\lambda x.M) & = & FV(M) \setminus \{x\}
\end{array}$$

A term is closed iff $FV(M) = \emptyset$. A variable occurrence which is not free is called a bound occurrence.

The result of replacing all the free occurrences of a variable x by a term N in a term M is denoted by $M[N/x]$. That substitution $M[N/x]$ is allowed, in which case we say that x is replaceable by N in M, if x does not occur free in any subterm of M of the form $\lambda y.P$ and $y \in FV(N)$. In the rest of the paper we assume that the sets of free and bound variables of a term are disjoint, thus every substitution $M[N/x]$ is allowed.

Example 8 *The following functional program*

$$f\ x\ y\ =\ g\ (x\ y)\ z$$

is represented by the λ-term

$$(\lambda xy.g(xy)z)$$

3 Parametric Polymorphism

In programming languages where functions or procedures have a unique type, one cannot define functions that might be applied to different data types, making it necessary to define distinct functions for different types. Such languages, like Pascal and C, are called *monomorphic*. For example, let $(\lambda x.x)$ be the term representing the identity function. In a monomorphic type system we can consider the following types for this term:

$$\vdash \lambda x.x : int \to int \qquad\qquad \vdash \lambda x.x : char \to char$$

For functions like the identity, one is interested in using the same function for a set of types instead of defining a different one for each different type. This concept is known as *type polymorphism*.

There are several kinds of polymorphism. We are interested in the kind of polymorphism provided by the ML language, the *parametric polymorphism*, in which polymorphic functions are used uniformly in a set of types.

In parametric polymorphism, one describes sets of types using schemes with parameters that can be instantiated, allowing a representation of the set of types for a certain term. For the identity function $(\lambda x.x)$, the set of types that can be derived is represented by $(\forall \alpha.\alpha \to \alpha)$.

We now describe the Damas-Milner type system [10], which is the base of type systems for polymorphic languages.

3.1 The term language

The term language is just the λ-calculus with local definitions.

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

The term $let\ x = M\ in\ M'$, although semantically equivalent to $(\lambda x.M')M$, allows more terms to be typable, because, in $let\ x = M\ in\ M'$, x can be used polymorphically in M', even if the term $(\lambda x.M')$ is not typable. We shall see an example concerning the *let* term when we present the system.

3.2 Type schemes

We define the notion of type scheme so that, given a λ-term M, one can represent the set of types that are allowed for that term.

We first define the set of simple types (or Curry types).

Definition 9 *Let α range over an infinite set of type variables, $\mathcal{V}$. The set of simple types (here denoted by τ, τ' and τ'') is defined inductively by:*

$$\tau ::= \alpha \mid \tau' \to \tau''$$

Definition 10 *We say that* σ *is a type scheme if* σ *is a simple type* τ *or is of the form* $\forall \alpha_1, \ldots, \alpha_n.\tau$, *where* $\alpha_1, \ldots, \alpha_n$ *are type variables called generic.*

Type schemes represent the set of types resulting from every possible substitution of the generic variables by simple types.

Definition 11 *Let* τ *be a type and* σ *a type scheme. We say that* τ *is a generic instance of* σ *iff* $\sigma = \tau$ *or* $\sigma = \forall \alpha_1, \ldots, \alpha_n.\tau'$ *and* $\exists \tau_1, \ldots, \tau_n$ *such that* $\tau = [\tau_i/\alpha_i]\tau'$.

3.3 Types

The set of types of the Damas-Milner type system is defined by:

$$\sigma \quad ::= \quad \tau \mid \forall \alpha.\sigma'$$

where τ is a simple type, and σ, σ' are type schemes.

3.4 The type system

Let Γ be one set of type declarations where for each term variable there is at most one type declaration. Let σ range over type schemes and τ, τ' range over simple types. $\Gamma \vdash M : \sigma$, meaning that M has type σ with the type declarations Γ, is defined by the following rules:

$$(\mathit{Var}) \qquad \Gamma \vdash x : \sigma, \text{ if } (x : \sigma) \in \Gamma$$

$$(\mathit{Gen}) \qquad \frac{\Gamma \vdash M : \sigma, \text{ if } \alpha \notin \mathit{fv}(\Gamma)}{\Gamma \vdash M : \forall \alpha.\sigma}$$

$$(\mathit{Inst}) \qquad \frac{\Gamma \vdash M : \forall \alpha.\sigma}{\Gamma \vdash M : \sigma[\tau/\alpha]}$$

$$(\mathit{App}) \qquad \frac{\Gamma \vdash M : (\tau' \to \tau), \Gamma \vdash M' : \tau'}{\Gamma \vdash (MM') : \tau}$$

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

$$(\mathit{Let}) \qquad \frac{\Gamma \vdash M : \sigma, \Gamma \cup \{x : \sigma\} \vdash M' : \tau}{\Gamma \vdash \mathit{let}\ x = M\ \mathit{in}\ M' : \tau}$$

The function $\mathit{fv}(\Gamma)$, returns the set of free type variables in Γ.

Example 12 *Consider the term let* $y = (\lambda x.x)$ *in* yy, *and the type derivation for this term given by figure 1.*

Note that the term $(\lambda y.yy)(\lambda x.x)$, *semantically equivalent to let* $y = (\lambda x.x)$ *in* yy, *is not typable, since it is not possible to type the subterm* $(\lambda y.yy)$ *because* y *cannot be used polymorphically in* $\lambda y.yy$.

4 Constraint based Type Inference

4.1 The HM($\mathcal{X}$) framework

Sulzmann, Odersky and Wehr defined the HM($\mathcal{X}$) [32] as a general framework for extensions of the Damas-Milner type system with constraints. The $\mathcal{X}$ in the HM($\mathcal{X}$) framework stands for a constraint system along the lines defined in [27]. 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;

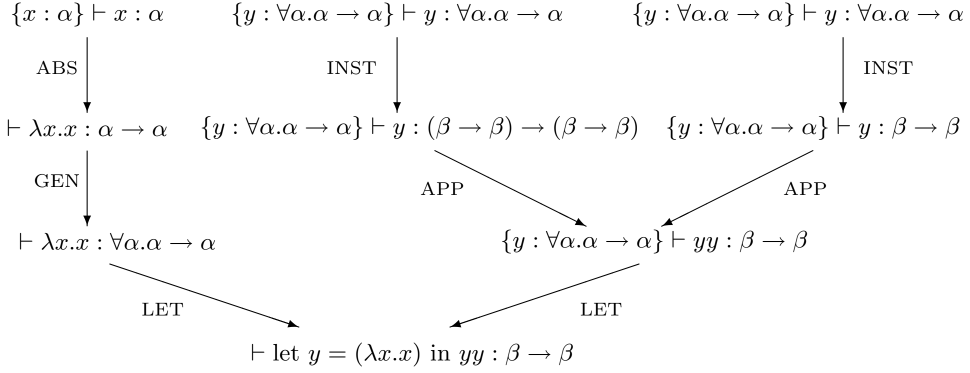

Figure 1: Type derivation

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 ($\exists$);

4. a set of predicates relating types. This set must contain an equality predicate.

Concerning type inference, an instance of the general type inference framework HM($\mathcal{X}$) is defined 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, and Γ_0 is the set with the initial type declarations.

4.1.1 The term language

The term language is the same presented in last section for the Damas-Milner type system:

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

Additional language constructors are expressed as predefined variables and their types are declared in the initial environment Γ_0.

4.1.2 The type language

The definition of types is more general than the one for the Damas-Milner system. The type language is now defined by $\mathcal{T}$, and might contain other type constructors besides $\to$. 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:

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

T stands for other type constructors defined in $\mathcal{T}$. Those constructors depend on the particular instance of HM($\mathcal{X}$). Type schemes $\forall\alpha.C \Rightarrow \sigma$, include a set of constraints C, which restricts the types that can be instantiated to α. In this framework $\{C_1, \ldots, C_n\}$ is equivalent to $C_1 \wedge \cdots \wedge C_n$.

4.1.3 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, where Γ_x denotes the result of excluding from Γ any assumptions about x:

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

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

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

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

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

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

The type $\forall \bar{\alpha}.D \Rightarrow \tau$ is a different notation for $\forall \alpha_1.\text{true} \Rightarrow \cdots \forall \alpha_n.D \Rightarrow \tau$, 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.

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.

In some situations we might want to restrict the set of constraints appearing in type schemes and type derivations. For that purpose a set of constraints in *solved form* is defined and the constraints in type schemes and type derivations are restricted to constraints in that set. 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.

4.1.4 Normalization

Type inference in HM($\mathcal{X}$) combines constraint generation and constraint normalization. The general idea is to transform the problem of inferring a type for a term into a constraint satisfaction problem, such that the term is typable if the constraint satisfaction problem has a solution. The solution of the constraint problem is a constraint in solved form.

Here we present the concept of constraint normalization, along the lines presented in [31].

In [31] 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 13 *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 14 *Let C, C', D, ϕ and ϕ' be constraints in $\mathcal{X}$, then $C \wedge \phi$, is a 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 $U = fv(C \wedge C' \wedge \phi \wedge \phi') \backslash fv(D)$, with fv being the function that returns the set of free variables of a constraint/type.*

The principal normal form represents the most general solution of a constraint problem.

Example 15 *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 16 *The function* normalize *from constraints to normal forms is defined as:*

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

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.

Example 17 *Consider a constraint system, with primitive constraints of the form* $\alpha :: \tau$*, and the following rule:*

$$(\alpha :: \tau_1) \wedge (\alpha :: \tau_2) \models \tau_1 = \tau_2$$

Then for $C = (\alpha :: (\beta \to \beta) \to \beta) \wedge (\alpha :: \gamma \to \beta)$*,*

$$\textit{normalize}(C) = (\alpha :: (\beta \to \beta) \to \beta) \wedge [(\beta \to \beta)/\gamma]$$

Definition 18 *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.*

4.2 The Type Inference Algorithm

In [32] a general algorithm for infering types in the HM($\mathcal{X}$) framework is defined. 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 19 *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:*

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

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

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

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

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

The predicate *normalize* is defined as follows:

$$normalize([\,], C) \leftarrow findall_constraints(_, CId),\\ removeId(CId, C).$$

$$normalize([X|R1], C) \leftarrow call(X),\\ normalize(R1, C).$$

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). That is, D is a new instance of D_1 and τ a new instance of τ_1, which results from replacing the variables in $\bar{\alpha}$ by fresh variables.

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

In this section we present instances of the HM($\mathcal{X}$) framework, by defining the parameters ($\mathcal{X}$, $\mathcal{T}$, $\mathcal{S}$, Γ_0). The systems presented are the Damas-Milner type system presented in section 3, and the Ohori type system for extensible records reported in [23]. Those systems were used in [30] 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.

4.4 The Damas-Milner Type System

Type inference for pure ML, also known as Damas-Milner type system [10], is obtained by defining $\mathcal{X}$, $\mathcal{T}$, $\mathcal{S}$, Γ_0 in the following way.

4.4.1 The constraint system $\mathcal{DM}$

The constraint system $\mathcal{DM}$ is the Herbrand constraint system (see example 15), with primitive constraints of the form $\tau_1 = \tau_2$, which are solved by unification.

4.4.2 The type language $\mathcal{T}$

We define the type language $\mathcal{T}$ by the following grammar:

$$\tau ::= \alpha \mid \tau \rightarrow \tau'$$

That is, the only type constructor is $\rightarrow$.

4.4.3 The constraints in solved form $\mathcal{S}$

The set $\mathcal{S}$ of constraints in solved form consists of only of constraints of the form:

$$C ::= true$$

where *true* is represented by the empty set. That is, the only possible type schemes are of the form $\forall\alpha.\{\} \Rightarrow \sigma$, which correspond to $\forall\alpha.\sigma$ in the Damas-Milner type system.

4.4.4 The initial environment Γ_0

The term language is the one defined for HM($\mathcal{X}$). Additional language constructors are declared in the initial environment Γ_0. That is, if one wants to extend the language to deal with comparing functions $(eq, <, >, \leq, \geq)$, one should consider those functions as variables declared in the initial environment.

Example 20 *Consider the program:*

$$
\begin{array}{lll}
f & = & \textbf{let} \\
 & & \quad g\ x\ =\ x \\
 & & \textbf{in} \\
 & & \quad g\ g
\end{array}
$$

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

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

Thus,

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

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

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

Thus,

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

After generalizing the variables which do not occur free in the basis, the types inferred for g and f with initial set of type declarations $\Gamma_0 = \emptyset$, are:

$$\begin{array}{l} f :: \forall \alpha_2 . \{\} \Rightarrow \alpha_2 \rightarrow \alpha_2 \\ g :: \forall \alpha . \{\} \Rightarrow \alpha \rightarrow \alpha \end{array}$$

4.5 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 Ohori type system extends the ML type system with polymorphic records.

4.5.1 The term language

The term language for this system is an extension to the Damas-Milner term language, with terms for records creation, field access and field modification:

$$\begin{array}{rcl} M & ::= & x \mid MM \mid \lambda x.M \mid let\ x = M\ in\ M \mid \\ & & \{l = M, \ldots, l = M\} \mid M.l \mid modify(M, l, M) \end{array}$$

Example 21 *As an example of an expression in this language consider:*

$\lambda\, x\, y$. **let**
$name = \lambda\, z.\, z.Name$
in
$name\, \{Name = x,\ Age = y\}$

4.5.2 Types and Kinds

The set of types for the Ohori system is given by:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \rightarrow \tau' \mid \{l : \tau_1, \ldots, l : \tau_n\} \\ \sigma & ::= & \tau \mid \forall \alpha :: k.\sigma \end{array}$$

where τ is a monomorphic type and σ is a polymorphic type.

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

Example 22 *The following term for record creation*

$$\{Name = x, Age = y\}$$

has type

$$\{Name : \alpha, Age : \beta\}$$

where α and β are the types of x and y respectively.

The type variables in type schemes are constrained to a set of types which is called *kind*. The set of *kinds* is given by:

$$\kappa ::= U \mid \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$$

U is the set of all types. A kind $\{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$ represents the records which contain, at least, the fields $l_1, \ldots, l_n$ with types $\tau_1, \ldots, \tau_n$ respectively.

Example 23 *If a type variable α has kind U, then α can represent any type. If α has kind*

$$\{\!\{Name : string, Age : int\}\!\}$$

then α can represent the type

$$\{Name : string, Age : int, Phone : int\}$$
$$or$$
$$\{Name : string, Age : int, Address : string\}$$

but not,

$$\{Name : string, Phone : int, Address : string\}$$

Next we present the parameters $(\mathcal{X}, \mathcal{T}, \mathcal{S}, \Gamma_0)$ that define the HM($\mathcal{R}$) instance.

4.5.3 The type language $\mathcal{T}$

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

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

That is, the type language has an additional type constructor for record types.

4.5.4 The Constraint System $\mathcal{R}$

Kinded quantification is modeled in the following way:

- A *kind* κ is defined as $\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$, and defines the records which contain at least the fields $l_1, \ldots, l_n$.
- The primitive constraints of this system are of the form $(\tau :: \kappa)$, where τ is a type and κ is a *kind*.

In HM($\mathcal{R}$) there are no constraints of the form $\tau :: U$, because we consider that, in a type scheme $\forall \bar{\alpha}.C \Rightarrow \tau$, if a type variable $\alpha \in \bar{\alpha}$ does not appear in C then α can represent any type.

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* $\langle L : \tau_i \rangle$. Let $fv(K, V)$ be a predicate where V is the set of free variables of K. The normalization rules for $\mathcal{R}$ are the following:

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

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

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

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

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. Rule 4 simplifies constraints existentialy quantified by generalization, preserving logical equivalence.

As defined in [32], constraints in solved form are the satisfiable constraints of the form

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

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

$$\mathit{modify}_l : \forall\alpha, \beta.(\alpha :: \langle l : \beta\rangle) \Rightarrow \alpha \to \beta \to \alpha$$

for field modification.

Example 24 *Consider the following program:*

$f\ x\ =\ \mathbf{let}$
$\qquad g\ x\ y\ =\ \{l_1\ =\ x,\ l_2\ =\ y\}$
$\quad \mathbf{in}$
$\quad \mathit{modify}(g\ 1\ 2, l_1, x)$

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:

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

Example 25 *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:

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

5 Intersection Types

Intersection Types were introduced in [7] to allow typings where different occurrences of the same variable are used with different types. This ability makes it possible to type terms, not typable in ML, such as $\lambda x.xx$.

In Intersection Type Systems (ITS), functional types of the form

$$\sigma_1 \cap \cdots \cap \sigma_n \to \sigma$$

are introduced to model the type of a function which returns an object of type σ when given an argument of, simultaneously, types $\sigma_1, \ldots, \sigma_n$.

Type inference for Intersection Type Systems is undecidable [7, 25]. However there are decidable restrictions which are useful for type assignment in programming languages [33, 11, 8, 34, 17, 18]. Particularly, the rank 2 Intersection Type System was studied in detail in [33, 16, 17, 34].

There are many different formulations of intersection type systems (see [33] for a survey).

Definition 26 *Let $\mathcal{V}$ be an infinite set of type variables. The set of Intersection Types is defined inductively by:*

1. *If $\alpha \in \mathcal{V}$ then α is a type;*
2. *If $\sigma_1, \ldots, \sigma_n$ and σ are types, then $\sigma_1 \cap \cdots \cap \sigma_n \to \sigma$ is a type.*

5.1 Rank 2 Intersection Types

Rank 2 Intersection Types are the most studied decidable restrictions of Intersection Type Systems. Rank 2 Intersection types are of the form $\tau_1 \cap \cdots \cap \tau_n \to \sigma$, where $\tau_1, \ldots, \tau_n$ are simple types. Here we present a basic definition of the Rank 2 Intersection Type System along the lines presented in [16].

Definition 27 *Let $\mathcal{T}_C$ be the set of simple types. We then consider the following sets of types:*

1. *The set of Rank 1 Types, $\mathcal{T}_1$ is defined by: If $\tau_1, \ldots, \tau_n \in \mathcal{T}_C$ $(n \geq 1)$, then $\overline{\tau} = \tau_1 \cap \cdots \cap \tau_n \in \mathcal{T}_1$.*
2. *The set of Rank 2 Intersection Types, $\mathcal{T}_2$ is inductively defined by:*
   - *(a) If $\tau \in \mathcal{T}_C$, then $\tau \in \mathcal{T}_2$.*
   - *(b) If $\overline{\tau} \in \mathcal{T}_1$ and $\sigma \in \mathcal{T}_2$, then $\overline{\tau} \to \sigma \in \mathcal{T}_2$.*

We assume that the intersection operator is associative, commutative and idempotent.

Definition 28 *A Rank 2 basis is a set of pairs of the form $x : \sigma$ where x (the subject) is a term-variable and σ (the predicate) $\in \mathcal{T}_1$. Alternatively one can define a basis as a partial function from term-variables to types in $\mathcal{T}_1$.*

The set $\mathcal{T}_1$ consists of nonempty, finite intersections of simple types. The set $\mathcal{T}_2$ is the set of types containing intersections of simple types but only to the left of an arrow.

Definition 29 *Let $\tau, \tau_1, \ldots, \tau_n \in \mathcal{T}_C$ and $\sigma, \sigma' \in \mathcal{T}_2$. The Rank 2 Type System is defined by:*

$$(Var) \qquad A \cup \{x : \tau_1 \cap \cdots \cap \tau_n\} \vdash_2 x : \tau_i \quad (i \in \{1, \ldots, n\})$$

$$(Abs) \qquad \frac{A_x \cup \{x : \tau_1 \cap \cdots \cap \tau_n\} \vdash_2 M : \sigma}{A \vdash_2 \lambda x.M : \tau_1 \cap \cdots \cap \tau_n \to \sigma}$$

$$(App) \qquad \frac{A \vdash_2 M : \tau_1 \cap \cdots \cap \tau_n \to \sigma \quad A \vdash_2 N : \tau_1 \ldots A \vdash_2 N : \tau_n}{A \vdash_2 MN : \sigma}$$

Example 30 *Consider the term $(\lambda x.xx)$. We have the following type derivation:*

$$\{x : (\alpha \to \beta) \cap \alpha\} \vdash_2 x : \alpha \to \beta \qquad \{x : (\alpha \to \beta) \cap \alpha\} \vdash_2 x : \alpha$$

APP $\qquad$ APP

$$\{x : (\alpha \to \beta) \cap \alpha\} \vdash_2 xx : \beta$$

ABS

$$\vdash_2 \lambda x.xx : ((\alpha \to \beta) \cap \alpha) \to \beta$$

Note that the term $(\lambda x.xx)$ is not typable in ML.

5.2 Type Inference

The type inference algorithm that we implement in Prolog and CHR in this section was defined by van Bakel and Trevor Jim in [33] and [16]. We will follow the presentation of [16] since it is a constraint based formulation of the algorithm.

In [16], to avoid unification of types involving intersections (which have algebraic properties that would complicate the unification process) the type inference algorithm generates a set of subtyping constraints which respect semantic inclusion between types. In the same paper, the author proved that these subtype constraints could be transformed into a set of equality constraints which can be solved by syntactic unification, defined a constraint system for these transformation, and proved

that the resulting algorithm is sound with respect to the type system. In our implementation the transformation of subtyping into equality constraints becomes quite clear as a set of CHR rules. As for the previous systems, equality constraints are solved by Prolog unification with an occur-check. Let us now present our implementation of the type inference algorithm defined in [16] for the Rank-2 Intersection Type System.

Definition 31 *Given a term M, the algorithm gives as output the type τ for M, and a Rank 2 basis A. The type inference algorithm is defined by the following set of Horn clauses:*

$$type([(x,\alpha)], x, \alpha).$$

$$\begin{aligned} type(A_x, \lambda x.N, \tau' \to \tau) \quad \leftarrow \quad & type(A, N, \tau), \\ & member(x, A, \tau'), \\ & remove(x, A, A_x). \end{aligned}$$

$$\begin{aligned} type(A, \lambda x.N, \alpha \to \tau) \quad \leftarrow \quad & not\ member(x, A, _), \\ & type(A, N, \tau). \end{aligned}$$

$$\begin{aligned} type(A, M_1M_2, \alpha_2) \quad \leftarrow \quad & type(A_1, M_1, \alpha), \\ & var(\alpha), \\ & type(A_2, M_2, \tau_2), \\ & unify_with_occurs_check(\alpha, \alpha_1 \to \alpha_2), \\ & normalize([\tau_2 \leq \alpha_1], []), \\ & plus(A_1, A_2, A). \end{aligned}$$

$$\begin{aligned} type(A, M_1M_2, \sigma_1) \quad \leftarrow \quad & type(A_1, M_1, \overline{\tau} \to \sigma_1), \\ & itlength(\overline{\tau}, N), \\ & ntype(N, A_I, M_2, \overline{\tau}, Clist), \\ & normalize(Clist, []), \\ & plus(A_1, A_I, A). \end{aligned}$$

We represent a Rank 1 type $\tau_1 \cap \cdots \cap \tau_n$ as a list of Curry types $[\tau_1, \ldots, \tau_n]$.

The predicate *normalize* is the one defined in section 4.2. However, in this system, type inference is only possible if the inequality constraints are transformed into equality constraints, which are solved by unification, thus we expect the result of normalize to be an empty set of constraints.

The predicate *plus* used in the rules for the application implements the following operation on basis:

$$(A_1 + A_2)(x) = \begin{cases} A_1(x) & \textit{if } x \notin dom(A_2), \\ A_2(x) & \textit{if } x \notin dom(A_1), \\ A_1(x) \wedge A_2(x) & \textit{otherwise.} \end{cases}$$

for any $x \in dom(A_1) \cup dom(A_2)$.

The predicate $itlength(\overline{\tau}, n)$ in the second rule for the application gives the number n of types in $\overline{\tau}$ and, for that n, the predicate $ntype(n, A_I, M_2, \overline{\tau}, Clist)$, calls n times the predicate *type* for M_2 and builds a list of constraints *Clist* of the form $[\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n]$, and a Rank 2 basis $A_I = A_1 + \cdots + A_n$, where each pair (σ_i, A_i) ($i \in \{1, \ldots, n\}$) is such that $type(A_i, M_2, \sigma_i)$. The implementation of *ntype* is the following:

$$ntype(0, A, M, \sigma, [\tau \leq \sigma]) \quad \leftarrow \quad type(A, M, \tau).$$

$$ntype(1, A, M, [\sigma], [\tau \leq \sigma]) \quad \leftarrow \quad type(A, M, \tau).$$

$$ntype(N, A, M, [\sigma|R], [\tau \leq \sigma|RC]) \quad \leftarrow \quad type(A_1, M, \tau),\; N_1 \;is\; N-1,\; ntype(N_1, A_2, M, R, RC),\; plus(A_1, A_2, A).$$

For simple types the call for *itlength* returns 0 and the first rule of *ntype* is used.

Let α, α_1 and α_2 be type variables. Subtype normalization is defined by the following set of CHR rules:

$$twone1 \;@\; \sigma_1 \to \sigma_2 \leq \alpha \Longleftrightarrow var(\alpha) \mid \alpha_1 \leq \sigma_1, \sigma_2 \leq \alpha_2, unify_with_occurs_check(\alpha, \alpha_1 \to \alpha_2).$$

$$twone2 \;@\; \sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2 \Longleftrightarrow \tau_1 \leq \sigma_1, \sigma_2 \leq \tau_2.$$

$$twone3 \;@\; \sigma \leq [\tau_1] \Longleftrightarrow \sigma \leq \tau_1.$$

$$twone4 \;@\; \sigma \leq [\tau_1 \mid \tau_2] \Longleftrightarrow \sigma \leq \tau_1, \sigma \leq \tau_2.$$

$$twone5 \;@\; \alpha \leq \tau \Longleftrightarrow var(\alpha), simple(\tau) \mid unify_with_occurs_check(\alpha, \tau).$$

The first rule simplifies a subtype constraint decomposing it into two simpler constraints. Rule 2 implements contra-variance. The next two rules transform a constraint of the form $\sigma \leq \tau_1 \cap \cdots \cap \tau_n$ into a set of constrains of the form $\sigma \leq \tau_1 \wedge \cdots \wedge \sigma \leq \tau_n$. The last rule transforms an inequality constraint $\alpha \leq \tau$, into a equality constraint $\alpha = \tau$, and solves it by unification, if α is a variable and τ is a simple type.

Example 32 *Consider the following function:*

$$f = (\lambda\, x.xx)I$$

being I the identity function $(\lambda y.y)$. Applying the type inference algorithm to $(\lambda x.xx)$ we have:

$type(A_x, (\lambda x.xx), \beta_1 \to \beta_2)$
$\quad type(A, xx, \beta_2)$
$\quad\quad type([(x, \alpha_1)], x, \alpha_1)$
$\quad\quad var(\alpha_1)$
$\quad\quad type([(x, \alpha_2)], x, \alpha_2)$
$\quad\quad unify_with_occurs_check(\alpha_1, \alpha_3 \to \beta_2)$ $\qquad \alpha_1 = \alpha_3 \to \beta_2$
$\quad\quad \boldsymbol{normalize}([\alpha_2 \leq \alpha_3], [\,])$
$\quad\quad\quad unify_with_occurs_check(\alpha_2, \alpha_3)$ $\qquad \alpha_2 = \alpha_3$
$\quad\quad plus([(x, \alpha_3 \to \beta_2)], [(x, \alpha_3)], [(x, [\alpha_3 \to \beta_2, \alpha_3])])$
$\quad type([(x, [\alpha_3 \to \beta_2, \alpha_3])], xx, \beta_2)$
$\quad member(x, [(x, [\alpha_3 \to \beta_2, \alpha_3])], [\alpha_3 \to \beta_2, \alpha_3])$
$\quad remove(x, [(x, [\alpha_3 \to \beta_2, \alpha_3])], [\,])$

Thus,
$\quad type([\,], (\lambda x.xx), [\alpha_3 \to \beta_2, \alpha_3] \to \beta_2)$

Applying the type inference algorithm to I we have:

$type(A_y, (\lambda y.y), \beta_3 \to \beta_4)$
$\quad type([(y, \alpha_4)], y, \alpha_4)$
$\quad member(y, [(y, \alpha_4)], \alpha_4)$
$\quad remove(y, [(y, \alpha_4)], [\,])$

Thus,

$type([\,], (\lambda y.y), \alpha_4 \rightarrow \alpha_4)$

If we apply the type inference algorithm to $(\lambda x.xx)I$*:*

$type(A, (\lambda x.xx)I, \beta_2)$

$type([\,], (\lambda x.xx), [\alpha_3 \rightarrow \beta_2, \alpha_3] \rightarrow \beta_2)$

$itlength([\alpha_3 \rightarrow \beta_2, \alpha_3], 2)$

$ntype(2, [], (\lambda y.y), [\alpha_3 \rightarrow \beta_2, \alpha_3], [\alpha_4 \rightarrow \alpha_4 \leq \alpha_3 \rightarrow \beta_2, \alpha_5 \rightarrow \alpha_5 \leq \alpha_3])$

$\boldsymbol{normalize}([\alpha_4 \rightarrow \alpha_4 \leq \alpha_3 \rightarrow \beta_2, \alpha_5 \rightarrow \alpha_5 \leq \alpha_3], [\,])$, $\alpha_3 = \alpha_5 \rightarrow \alpha_5,$
$\beta_2 = \alpha_4 = \alpha_3 = \alpha_5 \rightarrow \alpha_5$

$plus([\,], [\,], [\,])$

Thus,

$type([\,], (\lambda x.xx)I, \alpha_5 \rightarrow \alpha_5)$

Example 33 *Consider the following function:*

$$f\ x\ y\ z\ =\ x\ z\ (y\ z)$$

Applying the inference algorithm obtains the following type for f*:*

$$f\ ::\ (\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3) \rightarrow (\alpha_4 \rightarrow \alpha_2) \rightarrow (\alpha_1 \cap \alpha_4) \rightarrow \alpha_3$$

6 Conclusion

Type inference for modern programming languages is based on logical systems and constraint resolution. In this paper we implement three type inference algorithms, which belong to the state of the art of the area, using constraint logic programming. The implementations presented are clear and highly declarative and show one particularly interesting area of application of logic programming, which is the design and implementation of type inference algorithms for programming languages.

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] A. Aiken and E.L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architechture*, 1993.

[2] S. Alves and M. Florido. Type Inference using Constraint Handling Rules. In *Eletronic Notes in Theoretical Computer Science (Proceedings of the 10th International Workshop on Functional and Logic Programming)*, volume 64. Elsevier Science, 2001.

[3] H. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[4] Henk Barendregt. The impact of the Lambda calculus on logic and Computer Science. *Bulletin of Symbolic Logic*, 3(3):181–215, 1997.

[5] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[6] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932.

[7] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[8] M. Coppo and P. Giannini. Principal types and unification for simple intersection type systems. *Information and Computation*, 122(1), 1995.

[9] E. Coquery and F. Fages. From Typing Constraints to Typed Constraint Systems in CHR. In *Third Workshop on Rule-Based Constraint Reasoning and Programming*, Cyprus, December 2001.

[10] 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.

[11] F. Damiani and P. Giannini. A decidable intersection type system based on relevance. In *Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science. Springer Verlag, 1994.

[12] 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.

[13] T. Fruhwirth. Theory and practice of Constraint Handling Rules. *The Journal of Logic Programming*, 37, 1998.

[14] K. Glynn, P. Stuckey, and Sulzmann M. Type Classes and Constraint Handling Rules. In *Workshop on Rule-Based Constraint Reasoning and Programming*, 2000.

[15] 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.

[16] T. Jim. Rank 2 type systems and recursive definitions. Technical report, Massachusetts Institute of Technology, 1995.

[17] T. Jim. What are principal typings and are they good for? In *ACM Symp. Principles of Programming Languages*, 1996.

[18] A. J. Kfoury and J. B. Wells. Principality and Decidable Type Inference for Finite-Rank Intersection Types. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 161–174, New York, NY, 1999.

[19] D. Leivant. Discrete polymorpism. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* , 1990.

[20] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[21] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages.* ACM Press, 1993.

[22] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming & Computer Architecture*, San Diego, California, June 1995. ACM Press.

[23] Ohori. A Polymorphic Record Calculus and its Compilation. *ACM TOPLAS*, 6, 1995.

[24] F. Pottier. Simplifying subtyping constraints. In *International Conference on Functional Programming* , 1996.

[25] G. Pottinger. A type assignement for the strongly normalizable terms. In J.R. Hindley and J.P. Seldin, editor, *To H.B. Curry, Essays in Combinatory Logic, Lambda-calculus and Formalism*, pages 561–577. Academic Press, 1980.

[26] J. A. Robinson. A Machine-Oriented logic based on the Resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[27] Vijay A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[28] P. Stuckey and M. Sulzmann. A systematic approach in type system design based on Constraint Handling Rules. In *Third Workshop on Rule-Based Constraint Reasoning and Programming*, Cyprus, December 2001.

[29] Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. In *International Conference on Functional Programming*, October 2002.

[30] Martin Sulzmann. *A general framework for Hindley/Milner type systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[31] 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.

[32] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[33] S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Department of Computer Science, University of Nijmegen, 1993.

[34] S. van Bakel. Rank 2 intersection type assignment in term rewriting systems. *Fundamenta Informaticae*, 2(26):141–166, 1996.

[35] 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.