Type inference and checking for POS-typed functional logic programs *

J.M. Almendros-Jiménez
Dpto. Informática y Automática. Facultad de CC. Matemáticas.
Universidad Complutense. E-28040 Madrid. Spain.
Email:jesusmal@eucmos.sim.ucm.es.

Abstract

In this paper we investigate the type inference and checking for a polymorphic order-sorted typed functional logic language. This language consists of a specification of types, a set of type declarations for data constructors and functions and a set of constructor-based conditional rewriting rules including data and type conditions as constraints in the rule applicability. We have interested in type conditions that can be satisfiable (i.e. the satisfiability depends on the program rules and therefore it is not decidable statically) and make the program rules well-typed. The well-typedness of a program rule establishes that expressions occurring in it must be well-typed w.r.t. the type declarations and its type conditions. We present an algorithm that infers a minimal set of type conditions making well-typed a rule and that checks satisfiability of the type information provided by the user. This algorithm is shown to be sound and complete.

1 Introduction

The usefulness of type systems has been widely accepted to detect programming errors, to obtain more readable programs and run-time optimizations.
Polymorphic order-sorted type systems include both parametric and inclusion polymorphism providing more expressive power. Parametric polymorphism parametrizes types by means of type variables which represent any type. Inclusion polymorphism allows subtype relations between types. Parametric polymorphism was introduced in functional languages in the language ML [DaMi82] and incorporated to logic programming in [MyOk84], [Han89]. These type systems were proved to be usually static type systems, that is, type information is not required at run-time. Inclusion polymorphism was studied in languages based on order-sorted equational logic in [GoMe92] and realized on the language OBJ-3. By contrast, type systems with inclusion polymorphism are dynamic type systems, that is, type information is required to be checked at run-time.

Type inference has been considered in functional languages in order to obtain the type of a function from its defining rules in such a way that the type does not need to be declared. This is the case, for example, of the parametric type system of ML. However, as it is observed in [Han90] the infered type could not be the type the user expects, and therefore type declarations can be adequate in some cases. When the type declarations are allowed, the type checking consists of checking the type declaration on every occurrence of a function. In some cases, it is also desirable to combine type checking and inference in order to infer certain additional type information from the type declarations. Previous works for polymorphic order-sorted logic programs have considered this combination, obtaining the type of the data variables in every program rule (cfr. [Smo89], [HiTo92] and [Bei95]). For these type systems, the problem of inference of the most general type has been treated for functional programming in [FuMi90] and [Smi94].

The integration of logic and functional programming has been studied in the last years (see [Han94] for a survey) considering the lazy narrowing as a sound and complete operational semantics [GHLR96]. This combination is adequate in order to include lazy evaluation allowing partial non-strict functions and infinite data. Parametric type systems for functional logic languages has been studied in [Han90].

*This research has been partially supported by the Spanish CICYT (project TIC 95-0433-C03-01 "CPD"). [Alm96] is an extended version of this paper.

In a previous work [AlGi96] a polymorphic order-sorted type system for a functional logic language was presented giving a sound and complete operational semantics w.r.t. a proof calculus combining lazy narrowing and type solving.

In this paper we consider the functional logic language presented in [AlGi96], for which a program consists of a specification of types, a set of type declarations for data constructors and functions and a set of constructor-based conditional rewriting rules. The rule conditions include data and type conditions working as constraints for the rule applicability. In particular, the type conditions must guarantee the well-typedness of the rules. Furthermore, we are interested in detecting unsatisfiable type conditions which can make non applicable a program rule. Our work consists of completing the type information when it is not enough for the well-typedness, checking the unsatisfiability of the type conditions (the satisfiability is not decidable statically since it depends on the program rules) and at the same time checking the type declarations. Our approach [AlGi96] shows that the type inference is also useful for practical reasons. An efficient execution must only solve those rule type conditions not being checkable statically, that is, at compile-time. Type inference completes the rules with minimal information to check at run-time.

We present an algorithm for type checking and inference as tool to check and complete the type conditions of the rule conditions. The type system that we present is based on the type system presented in [HiTo92] which considers subtype relations between sorts and type constructors of the same arity defining a quasi-lattice and satisfying the monotonicity property. The type information that we infer contains type assumptions for data expressions involving defined functions, which can not be checkable statically. In this sense, we extend the previous works for logic languages (cfr. [Smo89], [HiTo92], [Bei95]). Furthermore, the algorithms that were presented in such works are incomplete, as it is observed in [Bei95], whose algorithm is unfortunately also incomplete as we will show. The type inference algorithm that we present is sound and complete, in a sense that we will specify later, and it captures previous algorithms for logic programming.

Example 1 Typed Program

TYPES

$opnat, nat \leq int$
$negint, zero \leq opnat$
$posint, zero \leq nat$

$technician, artist \leq person$
$doctor, comp\_scientist,$
$architect \leq technician$
$sculptor, painter, musician$
$\leq artist$

$elist(\alpha), nelist(\alpha) \leq list(\alpha)$
$etree(\alpha), nebtree(\alpha) \leq btree(\alpha)$
$bool$

DATA CONSTRUCTORS

$0 : zero$
$pred : opnat \to negint$
$suc : nat \to posint$
$true, false : bool$

$nil : elist(\alpha)$
$empty : ebtree(\alpha)$
$[\_|\_] : \alpha \times list(\alpha) \to nelist(\alpha)$
$tree : btree(\alpha) \times \alpha \times btree(\alpha) \to nebtree(\alpha)$

$john, michael : doctor$
$thomas, margaret : architect$
$robert, nathalie : painter$
$frank, david : comp\_scientist$
$richard, marie : sculptor$
$michael : musician$

FUNCTIONS

$head : nelist(\alpha) \to \alpha$
$head([X|X_s]) := X \Leftarrow X : \alpha, X_s : list(\alpha)$

$tail : nelist(\alpha) \to list(\alpha)$
$tail([X|X_s]) := X_s \Leftarrow X : \alpha, X_s : list(\alpha)$

$second : nelist(\alpha) \to \alpha$
$second(X) := head(tail(X)) \Leftarrow$
$X : nelist(\alpha), tail(X) : nelist(\alpha)$

$unit : list(\alpha) \to bool$
$unit(X) := false \Leftarrow X : elist(\alpha)$
$unit(X) := true \Leftarrow$
$X : nelist(\alpha), tail(X) : elist(\alpha)$
$unit(X) := false \Leftarrow$
$X : nelist(\alpha), tail(X) : nelist(\alpha)$

$father : person \to person$
$father(david) := michael ...$
$mother : person \to person$
$mother(richard) := marie ...$

$fam\_tree : person \to nebtree(person)$
$fam\_tree(X) := tree(fam\_tree(father(X)),$
$X, fam\_tree(mother(X)))$
$\Leftarrow X : person$

$any\_anc\_artist? : person \to bool$
$any\_anc\_artist?(X) := true \Leftarrow X : person, sel\_artist(preorder$
$(fam\_tree(X))) : nelist(person)$
$any\_anc\_artist?(X) := false \Leftarrow X : person, sel\_artist(preorder$
$(fam\_tree(X))) : elist(person)$

$sel\_artist : list(person) \to list(artist)$
$sel\_artist(nil) := nil$
$sel\_artist([X|L]) := [X|sel\_artist(L)] \Leftarrow X : artist, L : list(person)$
$sel\_artist([X|L]) := sel\_artist(L) \Leftarrow X : technician, L : list(person)$

2 Motivation

We show the expressivity of the language by means of a typed program example. In it we define basic types as integers, naturals, positive integers, etc. and polymorphic constructors for lists and trees. We can use the language to deal with databases, for instance, the family tree of a person (describing the profession of every ancestor). Furthermore, polymorphic functions as head, tail or second are defined.

The well-typedness property is crucial for the operational semantics (see [AlGi96] for more details). We can define a function as follows

$$second(X) := head(tail(X))$$

However, this rule is not applicable if X is empty, because *second* is not defined. This situation is specified by the type declaration $second : nelist(\alpha) \to \alpha$. Therefore, we include a type condition as $X : nelist(\alpha)$ to the previous rule, obtaining

$$second(X) := head(tail(X)) \Leftarrow X : nelist(\alpha)$$

But it is not enough because $head : nelist(\alpha) \to \alpha$ requires a non empty list as argument. Therefore, we include a type condition as

$$second(X) := head(tail(X)) \Leftarrow X : nelist(\alpha), tail(X) : nelist(\alpha)$$

The satisfiability of the type condition $tail(X) : nelist(\alpha)$ depends on the value of X, and therefore it must be checked at run-time, that is, dynamically.

In this way, the rule agrees with the type declarations. In this sense, we say that a program rule is well-typed. The well-typedness property for program rules depends on the type declarations and establishes that type conditions guarantee the well-typedness for the expressions involved in the rule (head, body and data conditions).

Furthermore, type conditions offer different possibilities to users. For instance, the user can include type conditions for making a case distinction based on subtypes. For example

$$\begin{array}{l}
musician? : person \to bool \\
musician?(X) := true \Leftarrow X : musician \\
musician?(X) := false \Leftarrow X : technician \\
musician?(X) := false \Leftarrow X : painter \\
musician?(X) := false \Leftarrow X : sculptor
\end{array}$$

Another possibility is using type conditions to restrict the application of the rule. Such is the case of

$$\begin{array}{l}
any_anc_artist? : person \to bool \\
any_anc_artist?(X) := true \\
\quad \Leftarrow X : person, sel_artist(preorder(fam_tree(X))) : nelist(person) \\
any_anc_artist?(X) := false \\
\quad \Leftarrow X : person, sel_artist(preorder(fam_tree(X))) : elist(person)
\end{array}$$

Finally, type conditions provide several alternatives to write a rule. For instance, the rule

$$unit(nil) := false$$

could also also written as

$$unit(X) := false \Leftarrow X : elist(\alpha)$$

using type conditions as constraints.

However, the type information can be not enough for concluding the well-typedness of the rule. The type inference algorithm that we present completes the type information provided by the user. In particular, if the user does not include any type information in the rule, we infer the type conditions in such a way that the rule becomes well-typed.

Furthermore, certain compatibility for type conditions is required. For instance: $tail(X) : nat$ or $head(0) : nat$ are unsatisfiable. However, $tail(X) : nelist(nat)$ or $head([X, Y]) : int$ are compatible.

The parametric polymorphism in the type declarations means that the rules can not refer to instances of the declaration (as in [HiTo92], [Smo89]) since type variables are universally quantified in the type declaration (in some approaches cfr. [Han89], [Han90], [Han91] this is not required with the aim of including higher order programming). This means that, for instance, a rule for *head* as $head([0]) := 0$ is not allowed. We also assume that extra type variables (not occurring in the type declaration) that the user introduces in the rule condition are also universally quantified.

On the other hand, an efficient execution of typed programs must only solve at run-time those type conditions that could not be checked at compile-time. The notion of minimal set of type conditions considers this idea, not including information that can be deduced syntactically, that is, information obtained from the type declarations. For example, type inference considers that the program rule

$$second(X) := head(tail(X))$$

is well-typed if type conditions are

$$second(X) := head(tail(X)) \Leftarrow X : nelist(\alpha), head(tail(X)) : \alpha$$

From the type declaration for *head* and $tail(X) : nelist(\alpha)$, is deduced that $head(tail(X)) : \alpha$. Furthermore, both type conditions are semantically equivalent. In such case, we will say that $head(tail(X)) : \alpha$ is safe from $tail(X) : nelist(\alpha)$. In consequence, a simpler set of type conditions including enough information for this rule could be

$$second(X) := head(tail(X)) \Leftarrow X : nelist(\alpha), tail(X) : nelist(\alpha)$$

Type inference statically deduces type information from the program rules and the type declarations without adding extra assumptions for making a well-typed rule. For example, if we consider the equation

$$\Leftarrow head(X) == Y$$

there exists several type conditions making it well-typed. For example,

$$\Leftarrow head(X) == Y \,\Box X : nelist(nat), Y : nat$$

and

$$\Leftarrow head(X) == Y \,\Box X : nelist(\alpha), Y : \alpha$$

The first case constraints the solutions of the equation. By contrast, the second case does not add new constraints. An important property of the type inference algorithm is that it keeps the semantic solutions.

It would also be desirable to consider type conditions containing at most one type assumption for each expression, that is, not containing for example $X : nat$ and $X : int$ or $tail(X) : nelist(\alpha)$ and $tail(X) : list(int)$. Furthermore, we want to eliminate redundant type conditions as $tail(X) : nelist(\alpha)$ and $head(tail(X)) : \alpha$.

The algorithm that we present infers not redundant, not safe, compatible type conditions containing at most one condition for each expression and not containing additional constraints. Furthermore, it keeps the semantic solutions.

3 The Language

3.1 Polymorphic Signature

The type system that we study was presented for logic programming in [HiTo92] and it is strong enough for supporting practical applications.

Definition 1 (Type Specification) *A type specification is a pair* $(S, \sqsubseteq_S)$ *where* S *is a type signature, i.e. a finite set of constants -sorts-, and functions -type constructors-, each one with an associated arity, and* $\sqsubseteq_S$ *is an partial ordering only defined between elements of the same arity, and being a lower quasi-lattice (i.e. every lower bounded subset of* S *has infimum). We use* K*,* L*, etc. for elements of* S*.*

Let $\mathcal{X}$ be a countable set of *type variables* α, β, γ,... Let $T_S(\mathcal{X})$ be the set of types τ, σ,... composed of terms over S and $\mathcal{X}$. We denote by $tvar(\tau)$ the set of type variables of τ and by $\sqsubseteq_T$ the lower quasi-lattice over $T_S(\mathcal{X})$ induced by $\sqsubseteq_S$ (i.e. $\alpha \sqsubseteq_T \alpha$, if $\alpha \in \mathcal{X}$, and $K(\tau_1, .., \tau_n) \sqsubseteq_T L(\sigma_1, .., \sigma_n)$ if $K \sqsubseteq_S L$, $ar(K)$=$ar(L)$=n and $\tau_i \sqsubseteq_T \sigma_i$, $i = 1, .., n$, $n \geq 0$). We say that a type τ is *maximal* if $\tau = \alpha$ or $\tau = K(\tau_1, .., \tau_n)$ $(n \geq 0)$, K is a maximal element in S and τ_i $(i = 1, .., n)$ are maximal types.

We denote by $TSUST(S, \mathcal{X})$ the set of type substitutions over S and $\mathcal{X}$ defined as usual and consider the partial order $\leq_{Dom}$ over $TSUST(S, \mathcal{X})$ defined as $\rho \leq_{Dom} \rho'$ if $|Dom(\rho)| \leq |Dom(\rho')|$.

Definition 2 (Polymorphic Signature) *A polymorphic signature* Σ *is a tuple* $(S, \sqsubseteq_S, CONS, FUN)$ *where:*

- $(S, \sqsubseteq_S)$ *is a type specification*
- $CONS$ *is a finite set of type declarations of data constructors of the form* $c : \tau_1..\tau_n \to \tau_0$[1]*, where* $\tau_i \in T_S(\mathcal{X})$*,* $i = 0,..,n$ *(n is called arity of c) satisfying the so-called transparency property, i.e.* $\cup_{i=1}^{n} tvar(\tau_i) \subseteq tvar(\tau_0)$[2]
- FUN *is a finite set of function type declarations of the form* $f : \tau_1..\tau_n \to \tau_0$*, where* $\tau_i \in T_S(\mathcal{X})$*,* $i = 0,..,n$ *(n is called arity of f).*

We admit a unique type declaration for every data constructor and function, excluding overloading [GoMe92], [Smi94]. We will write $\varphi \in CONS \cup FUN$ if $\varphi : \sigma \in CONS \cup FUN$. The set $[\varphi]$ defined by $\{\tau_1\rho..\tau_n\rho \to \tau_0\rho \mid \rho \in TSUST(S, \mathcal{X})\}$ denotes the set of *instances of the type declaration* of $\varphi : \tau_1..\tau_n \to \tau_0 \in CONS \cup FUN$.

In the following we suppose a fixed polymorphic signature Σ. Let $\mathcal{V}$ be a countable set of *data variables* X, Y, Z, etc. By $EXP_\Sigma(\mathcal{V})$ (resp. $TERM_\Sigma(\mathcal{V})$) we denote the set of *expressions* (resp. *terms*) over symbols of $CONS \cup FUN$ (resp. of $CONS$) and variables of $\mathcal{V}$.

In parametric type systems environments usually refer to set of type assumptions for data variables. We generalize this notion in the following definition.

Definition 3 (Sets of Type Conditions and Environments) *A set of type conditions is a finite set of the form* $\{e_i : \tau_i \mid e_i \in EXP_\Sigma(\mathcal{V}), \tau_i \in T_S(\mathcal{X})\}$ *where we make type assumptions about expressions. When only data variables are considered and there is at most one type assumption for each data variable, the set is called environment.*

In the sequel set of type conditions (resp. environments) will be represented by C_T (resp. V). We denote by $C_T\rho$ (resp. $V\rho$) the application of ρ to the types of C_T (resp. V). Next, we define when an expression e has type σ w.r.t. C_T, only using the syntactic type information given by the type declarations of the signature.

Definition 4 (Syntactic Type of an Expression) *Given* $e \in EXP_\Sigma(\mathcal{V})$ *and* C_T*, we say that e has syntactic type* σ *in* C_T *and* Σ *if and only if* $(\Sigma, C_T) \vdash_{WTC} e : \sigma$ *by the following inference rules :*

(TAE)	**Type Assumptions of Expressions**	$\dfrac{}{(\Sigma, C_T) \vdash_{WTC} e : \tau}$	% $e : \sigma \in C_T$ % $\sigma \sqsubseteq_T \tau$
(TD)	**Type Declaration**	$\dfrac{.., (\Sigma, C_T) \vdash_{WTC} e_i : \sigma_i, ..}{(\Sigma, C_T) \vdash_{WTC} \varphi(e_1, .., e_n) : \tau}$	% $\sigma_1..\sigma_n \to \sigma_0 \in [\varphi]$ % $\sigma_0 \sqsubseteq_T \tau$ % $\varphi \in CONS \cup FUN$

The well-typedness of a rule will consist of proving syntactically from the type conditions, that the expressions involved in the rule have the type declared. Note that this definition is relative to a set of type conditions because environments are not enough to decide syntactically the type of an expression, as the example of $head(tail(X)) : \alpha$ shows. Note that an expression e has trivially type σ relative to the set of type conditions $\{e : \sigma\}$ but we are interested in a set of type conditions that contains minimal information to conclude syntactically the type.

Definition 5 *A set of type conditions* C'_T *is derived from* C_T *if* $(\Sigma, C_T) \vdash_{WTC} e : \sigma$ *for all* $e : \sigma \in C'_T$*. In this case, we will write* $C_T \vdash_{WTC} C'_T$*.*

---

[1] We require a natural condition in a constructor-based language: the type declarations of data constructors satisfy that $\tau_0 = K(\bar{\alpha})$ where $\bar{\alpha}$ are different type variables and K is minimal (in a similar way to [Smo89] and [Bei95]).

[2] This is a natural restriction for data constructors, but not for functions. We complete the type declaration as in [Han91] considering $c : \alpha_1..\alpha_m\tau_1..\tau_n \to \tau_0$ with fictitious arguments where $\{\alpha_1, .., \alpha_m\} = tvar(\tau_0) \setminus \cup_{i=1}^{n} tvar(\tau_i)$. We suppose in the rest of the paper that the signature has been completed in this way.

3.2 Typed Programs

Typed programs in our language consist of a set of conditional constructor-based rewriting rules in the line of BABEL [MoRo92] and BABLOG [AGL94].

Definition 6 (Program Rule) *A program rule is of the form:* $f(t_1,..,t_n) := r \Leftarrow C_D \Box C_T$ *where* $f(t_1,..,t_n)$ *is called head,* r *is called body and* $C_D \Box C_T$ *is called rule condition (*C_D *is called data condition and* C_T *type condition) satisfying the following requirements:*

- $f \in FUN$ *of arity* n*,* $t_i \in TERM_\Sigma(\mathcal{V})$*,* $i = 1,..,n$ *and* $r \in EXP_\Sigma(\mathcal{V})$
- C_D *is of the form* $l_1 == r_1,..,l_m == r_m$ *where* $l_j,\ r_j \in EXP_\Sigma(\mathcal{V})$*,* $j = 1,..,m$
- C_T *is a relevant set of type conditions of the form* $e_1 : \tau_1,..,e_s : \tau_s$ [3] *where* $e_k \in EXP_\Sigma(\mathcal{V})$ *and* $\tau_k \in T_S(\mathcal{X})$*,* $k = 1,..,s$
- $(t_1,..,t_n)$ *is a linear tuple of terms and* $dvar(r) \subseteq dvar(f(t_1,..,t_n))$[4]

In the rule condition can occur the so-called *extra* data variables which do not occur in the head and therefore in the body neither. We call *extra* type variables to the set of type variables in the rule that do not occur in the type declaration of f. Operationally, data conditions (strict equalities) $l_j == r_j$ will be solved by narrowing l_j and r_j into unifiable terms, and type conditions $e_k : \tau_k$, by narrowing e_k to an expression of type τ_k.

A *well-typed program rule* is a program rule satisfying that if $f : \tau_1..\tau_n \to \tau_0 \in FUN$ then for all $i = 1,..,n$, $(\Sigma, C_T) \vdash_{WTC} t_i : \tau_i$, $(\Sigma, C_T) \vdash_{WTC} r : \tau_0$ and there exist $\sigma_j \in T_S(\mathcal{X})$ such that $(\Sigma, C_T) \vdash_{WTC} l_j : \sigma_j$, $(\Sigma, C_T) \vdash_{WTC} r_j : \sigma_j$ for all $l_j == r_j \in C_D$.

Definition 7 (Typed Programs and Goals) *A typed program* P *consists of a polymorphic signature* Σ *and a finite set of well-typed program rules satisfying a non overlapping property to ensure the functionality of the definitions (see [AGG96] for more details). A (well-typed) goal for a program has the same structure than the condition of a (well-typed) program rule.*

Example 2 *A goal for the program of the example 1 is*
$? - unit(X){==}Y.$
The type inference algorithm will complete the goal with $X{:}list(\alpha)$ *and* $Y{:}bool$*.*
The expected answers are
$> Y{=}false, X{:}elist(\alpha)$
$> Y{=}true, X{=}[Z|Z_s], X{:}nelist(\alpha), Z{:}\alpha, Z_s{:}elist(\alpha)$
$> Y{=}false, X{=}[Z|Z_s], Z{:}\alpha, Z_s{:}nelist(\alpha).$
A query of the database can be
$? - P{==}father(X), M{==}mother(X) \Box\ n\_th(N, preorder(fam\_tree(X))){:}musician,$
$N{:}posint, X{:}person$
The expected answer is
$> X{=}richard, P{=}david, M{=}marie, N{=}suc(suc(0)) \Box\ X{:}sculptor, P{:}comp\_scientist, M{:}sculptor,$
$N{:}posint$

In [AlGi96] we presented a semantic calculus to prove validity of formulas in a typed program P w.r.t. an environment V. In this calculus we defined the notion of semantic type of an expression $e : \sigma$ as follows. If $e \equiv X$ is a data variable then we consult the environment V. If e is $c(\bar{e})$ then we consult the type declaration of c. Otherwise, $e \equiv f(\bar{e})$ and then we can consult the type declaration of f or reduce $f(\bar{e})$ to r applying a program rule such that $r : \sigma$. A formula $l == r$ is true if we find a total term t reducing both sides. In the same work, we defined the notion of solution of a goal as tuples (V, θ, ρ) where V is an environment containing type assumptions for every data variable and θ, ρ are data and type substitutions respectively such that when we apply θ and ρ, the formulas we obtain become derivable in the semantic calculus. In the rest of the paper, we denote by $SSol(G)$ the set of semantic solutions of a goal G. Given $\mathcal{X}' \subseteq \mathcal{X}$, we will say that $SSol(C) = SSol(C')[\mathcal{X}']$ if both inclusions holds when we consider the tuples (V, θ, ρ) restricted to the type variables of $\mathcal{X}'$.

[3] C_T is relevant if every e_k $(k = 1,..,s)$ has any subexpression included in $f(\bar{t})$, r or C_D.
[4] These two requirements are needed in order to get confluence.

3.3 Minimal Set of Type Conditions of a Rule

The notion of safety of an expression w.r.t. its arguments takes into account derivability w.r.t. the $\vdash_{WTC}$-calculus and semantical equivalence. We give examples of this.

Example 3 *The type condition* $head(tail(X)) : \alpha$ *is safe from* $tail(X){:}nelist(\alpha)$ *because* $\{tail(X) : nelist(\alpha)\} \vdash_{WTC} head(tail(X)) : \alpha$ *and* $SSol(head(tail(X)){:}\alpha) = SSol(tail(X){:}nelist(\alpha))$*. On the other hand, the type condition* $head(tail(X)) : int$ *is also safe from* $tail(X) : nelist(int)$*. In this case, there exists a substitution that assigns maximal types* $\{\alpha \leftarrow int\}$ *such that the range of the type declaration of head matches with int and therefore it is guaranteed that the arguments are not restricted because the type. These are not the only cases, for example,* $head([suc(X)]) : posint$ *is safe from* $X : nat$ *because there exists a matching (but not assigning maximal types) for the range of the type declaration of head, that is,* $\{\alpha \leftarrow posint\}$ *such that* $[suc(X)] : nelist(posint)$ *is safe from* $X : nat$*. Note that the substitution must be minimal w.r.t.* $\leq_{Dom}$*, for example, if we consider* $proy_1 : \alpha \times \beta \rightarrow \alpha$ *and the type condition* $proy_1(X, Y) : int$*, since int is maximal then* $proy_1(X, Y) : int$ *is safe from* $X : int, Y : \beta$*. However, the substitution can not constraint* Y *in order to keep solutions. Note that the type condition* $head(tail(X)){:}zero$ *is not safe from* $tail(X){:}nelist(zero)$*, since there exist solutions of the former (for instance,* $[0, 0, suc(0)]$*) that are not solutions of the latter, due to the argument has been restricted.*

Definition 8 (Safety) *A* C_T *is not safe if it does not contain any type condition safe from its arguments. A type condition* $e : \sigma$ *is safe from its arguments if one of the following conditions holds:*

- *(1)* $e \equiv \varphi(\bar{e})$*,* $\varphi : \tau_1..\tau_n \rightarrow \tau_0 \in CONS \cup FUN$ *and there exists* $\rho \in TSUST(S, \mathcal{X})$ *minimal w.r.t.* $\leq_{Dom}$ *such that assigns maximal types,* $\tau_0\rho \sqsubseteq_T \sigma$ *and the type conditions* $e_i : \tau_i\rho$ $(i = 1, .., n)$ *are compatible.*
- *(2)* $e \equiv \varphi(\bar{e})$*,* $\varphi : \tau_1..\tau_n \rightarrow \tau_0 \in CONS \cup FUN$ *and there exists* $\rho \in TSUST(S, \mathcal{X})$ *minimal w.r.t.* $\leq_{Dom}$*, such that* $\tau_0\rho \sqsubseteq_T \sigma$ *and the type conditions* $e_i : \tau_i\rho$ $(i = 1, .., n)$ *are safe.*

We say that an expression is safe if it is safe from its arguments and its arguments are safe.

A set of type conditions is called compatible if it is not trivially unsatisfiable because the type declarations. The satisfiability is not decidable statically because type conditions involving function symbols may require evaluation.

Example 4 *The type condition* $tail(X) : nelist(int)$ *is compatible because there exists an infimum for* $list(\alpha)\{\alpha \leftarrow int\}$ *and* $nelist(int)$*. However* $tail(X) : nat$ *is not compatible.*

Definition 9 (Compatibility) *A* C_T *is compatible if it only contains compatible type conditions. A type condition* $e : \sigma$ *is compatible if it is safe or*

- $e \equiv X$*,* $X \in \mathcal{V}$*.*
- $e \equiv c(\bar{e})$*,* $c : \tau_1..\tau_n \rightarrow \tau_0 \in CONS$ *and there exists* $\rho \in TSUST(S, \mathcal{X})$ *such that* $\tau_0\rho \sqsubseteq_T \sigma$ *and the type conditions* $e_i : \tau_i\rho$ $(i = 1, .., n)$ *are compatible.*
- $e \equiv f(\bar{e})$*,* $f : \tau_1..\tau_n \rightarrow \tau_0 \in FUN$ *and there exists* $\rho \in TSUST(S, \mathcal{X})$ *such that* σ*,* $\tau_0\rho$ *have an infimum and* $e_i : \tau_i\rho$ $(i = 1, .., n)$ *are compatible.*

A set of type conditions contains implicitally more information than we can derive by the calculus $\vdash_{WTC}$ as the following example shows. Given the set $C_T = \{head(tail(X)) : zero\}$ we can deduce from C_T that $head(tail(X)) : zero, tail(X) : nelist(int)$ and $X : nelist(int)$. We make explicit the implicit information in the completion of a C_T. But it is not correct to deduce $X : nelist(zero)$ because there exist solutions of the former that are not solutions of the latter.

Definition 10 (Completion) *Given a* C_T *we define the completion, that we denote by* $\overline{C}_T$*, as the set of type conditions obtained by iterating the following process:*
If $(\Sigma, C_T) \vdash_{WTC} \varphi(\bar{e}) : \tau$*,* $\varphi \in CONS \cup FUN$ *by only applying* (TAE) *where* $\varphi(\bar{e}) : \tau$ *is safe for* ρ *from* $\bar{e}$ *and it was not previously considered then the process continues for* $C'_T = C_T \backslash \{\varphi(\bar{e}) : \tau\} \cup \{.., e_i : \tau_i\rho, ..\}$*.*

Lemma 1 *The process of completion terminates and satisfies that* $\overline{C}_T \vdash_{WTC} C_T$*. Furthermore, if* C_T *is not safe, compatible and relevant for* R*, then* $\overline{C}_T$ *is not safe, compatible and relevant for* R *and* $SSol(\overline{C}_T) = SSol(C_T)[tvar(C_T)]$[5]*.*

---

[5]Note that in $\overline{C}_T$ can occur new variables that do not occur in C_T.

Definition 11 (Redundancy and Repetitions) *A C_T is not redundant if it does not contain any type condition $e : \sigma$ such that $(\Sigma, \overline{C}_T \backslash \{e : \sigma\}) \vdash_{WTC} e : \sigma$.*
A C_T does not contain repetitions if it contains at most a type condition for every expression.

Lemma 2 (Semantic Solutions) *We have the following semantic results about type conditions:*
- *(1) If $\varphi(\bar{e}) : \sigma$ is safe for ρ from $\bar{e}$ then $SSol(\varphi(\bar{e}) : \sigma) = SSol(.., e_i : \tau_i\rho, ..)[tvar(\sigma)]$.*
- *(2) If $e : \sigma$ is not compatible then $SSol(e : \sigma) = \emptyset$.*
- *(3) If $e : \sigma$ is redundant in C_T then $SSol(C_T) = SSol(C_T \backslash \{e : \sigma\})$.*
- *(4) If C_T contains $e : \tau$ and $e : \tau'$ and there not exists $\rho \in TSUST(S, \mathcal{X})$ such that $\tau\rho$ and $\tau'\rho$ have infimum then $SSol(C_T) = \emptyset$.*
- *(5) If $C_T \vdash_{WTC} C'_T$ then $SSol(C_T) \subseteq SSol(C'_T)$.*

Definition 12 (Admissible and Minimal Set for a Rule) *A C_T is admissible for R if it is relevant for R, not safe, compatible, not redundant and does not contain repetitions.A C_T is minimal for a rule $R \equiv f(\bar{t}) := r \Leftarrow C_D \Box C_U$ if C_T is admissible, makes the rule well-typed and $C_T \vdash_{WTC} C_U$.*

4 The Type Inference Algorithm

The algorithm proceeds as follows: it takes as input a set of type conditions making the rule well-typed and containing the information provided by the user, and by means of a process of debugging, it deletes redundant and safe type information, detecting simultaneously incompatible type conditions. If no incompatibility is detected then the output of the algorithm is a minimal set of type conditions and failure otherwise.

In the following we suppose fixed a rule $R \equiv f(t_1, .., t_n) := r \Leftarrow C_D \Box C_U$, $C_D \equiv l_1 == r_1, .., l_m == r_m$ where C_U is the type information provided by the user and $f : \tau_1..\tau_n \rightarrow \tau_0 \in FUN$.

The input of the algorithm is a set of type conditions that makes the rule R well-typed. This set can be done easily, by considering $C_0 = C_U \cup \{t_1 : \tau_1, .., t_n : \tau_n, r : \tau_0, l_1 : \alpha_1, r_1 : \alpha_1, .., l_m : \alpha_m, r_m : \alpha_m\}$ where α_j, $j = 1, .., m$ are new variables. We distinguish two kinds of type variables. The so-called *frozen* variables (which occur in the type declaration or in C_U), which can not be instantiated because they are universally quantified, and the so-called *existential* variables (the rest of type variables) which can be instantiated since we search values for them. The universal quantification of the type variables in the type declaration is due to the parametric polymorphism (rules being instances are not allowed). Therefore, any attempt of binding them must be considered as an inconsistent situation that rises a failure. Frozen variables will be considered as sorts (constants) in a unitary quasi-lattice and then they will be trivially maximal elements.

In the algorithm we can distinguish three process:
(1) safety and compatibility checking,
(2) solving subtype conditions (which appear during the process),
(3) redundancy checking and collecting of type assumptions.

The process (1) checks type conditions of the form $\varphi(\bar{e}) : \tau$. To this end, the algorithm considers the type declaration of $\varphi : \tau_1..\tau_n \rightarrow \tau_0$ and distinguishes three cases:
- If φ is a data constructor then it is checked its compatibility.
- If φ is a function, it is checked if the type condition is safe from its arguments. The safety can be checked obtaining a maximal matching for τ_0 or checking the safety of the arguments for a matching of τ_0.
- If φ is a function and it is not safe from its arguments then it is checked its compatibility and if no failure is detected, it must be kept to be checked at run-time.

The process (2) solves subtype conditions. It consists of the decomposition of subtype conditions and the matching of type variables to types. The process (3) deletes redundant type conditions and collects type assumptions for the same expression. We suppose that a previous checking detects not relevant type conditions for the rule. Combining the process (1) and (2) we obtain a not safe and compatible set of type conditions. The process (3) cleans the set of type conditions of redundancies and repetitions.

This algorithm is illustrated by a calculus $\vdash_{TI}$ that derives assertions of the form $C_0 \cup C_T \Box \rho$ where C_0 and C_T are sets of type constraints and ρ is a type substitution. The set C_0 represents constraints not yet analysed and subtype conditions that occur in the process. C_T and ρ represent the computed minimal set and type substitution respectively. Initially, the set C_0 contains a set of type conditions that makes the rule well-typed, C_T is the empty set and ρ is the identity substitution. A successful derivation is a derivation concluding $C_0 = \emptyset$ and C_T, a non redundant set of type conditions that do not contain repetitions. The type inference algorithm fails if there not exist successful derivations.

The set C_0 includes two kinds of constraints: $e : \sigma$ where $e \in EXP_\Sigma(\mathcal{V})$, $\sigma \in T_S(\mathcal{X})$ (*type conditions*) and $\tau \sqsubseteq \tau'$ where $\tau, \tau' \in T_S(\mathcal{X})$ (*subtype conditions*). In C_0 we distinguish: type conditions that are checked as it has been explained ($e : \sigma$), type conditions that must be safe ($e : !\sigma$) and type conditions that must be compatible ($e : ?\sigma$). We eliminate compatible and safe type conditions when they have been checked, in order to eliminate redundant type conditions. We will write $\hat{\tau}$ to represent that the type variables of τ must be bound to maximal types. Otherwise, a failure rises. For simplicity of notation, we do not distinguish these variables in the calculus.

Next, we present the inference algorithm. The rules are presented in the order they should be applied by the algorithm (in order to obtain the required properties). We denote by $\varphi(\bar{e})$ to $\varphi(e_1, .., e_n)$, $K(\bar{\tau})$ to $K(\tau_1, .., \tau_n)$. We denote by $[\varphi]_\mathcal{X}$ the variants of the type declaration of φ. We use an indexed variable α_e to collect type assumptions for the expression e.

4.1 Safety and Compatibility Checking

(DV) **Data Variable**

$$\frac{X : \tau, C_0 \cup C_T \Box \rho}{C_0 \cup X : \tau, C_T \Box \rho}$$

(DC) **Decomposition of data constructor**

$$\frac{c(\bar{e}) : \tau, C_0 \cup C_T \Box \rho}{.., e_i : \tau_i, .., \tau_0 \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [c]_\mathcal{X}$ and $n \geq 0$.

(FI) **Decomposition of function I**

$$\frac{f(\bar{e}) : \tau, C_0 \cup C_T \Box \rho}{.., e_i : \tau_i, .., \hat{\tau}_0 \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [f]_\mathcal{X}$ and $n \geq 0$ (in τ_i $(i = 1, .., n)$ the type variables of τ_0 must be bound to maximal types).

(FII) **Decomposition of function II**

$$\frac{f(\bar{e}) : \tau, C_0 \cup C_T \Box \rho}{.., e_i : !\tau_i, .., \tau_0 \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [f]_\mathcal{X}$ and $n \geq 0$.

(NSF) **Not safe function**

$$\frac{f(\bar{e}) : \tau, C_0 \cup C_T \Box \rho}{e_i : ?\tau_i, \alpha \sqsubseteq \tau, \alpha \sqsubseteq \tau_0, C_0 \cup f(\bar{e}) : \alpha, C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [f]_\mathcal{X}$, α is a new type variable and $n \geq 0$.

(SDCFI) **Decomposition of safe data constructor or function I**

$$\frac{\varphi(\bar{e}) : !\tau, C_0 \cup C_T \Box \rho}{.., e_i : \tau_i, .., \hat{\tau}_0 \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [\varphi]_\mathcal{X}$ and $n \geq 0$ (in τ_i $(i = 1, .., n)$ the type variables of τ_0 must be bound to maximal types).

(SDCFII) **Decomposition of safe data constructor or function II**

$$\frac{\varphi(\bar{e}) : !\tau, C_0 \cup C_T \Box \rho}{.., e_i : !\tau_i, .., \tau_0 \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [\varphi]_\mathcal{X}$, $\varphi \in CONS \cup FUN$ and $n \geq 0$.

(CDV) **Compatible data variable**

$$\frac{X : ?\tau, C_0 \cup C_T \Box \rho}{\alpha_X \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

(CDC) **Compatible data constructor**

$$\frac{c(\bar{e}) : ?\tau, C_0 \cup C_T \Box \rho}{.., e_i : ?\tau_i, .., \tau_0 \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [c]_\mathcal{X}$ and $n \geq 0$.

(CF) **Compatible function**

$$\frac{f(\bar{e}) : ?\tau, C_0 \cup C_T \Box \rho}{e_i : ?\tau_i, \alpha \sqsubseteq \tau, \alpha \sqsubseteq \tau_0, C_0 \cup C_T \Box \rho}$$

where $\tau_1..\tau_n \to \tau_0 \in [f]_\mathcal{X}$, α is a new type variable and $n \geq 0$.

4.2 Solving Subtype Conditions

This process begins by decomposing every subtype condition having sorts or type constructors in both sides. The matching of type variables can generate new subtype conditions of this kind and therefore the process must be iterated. The last rule is applicable when only subtype conditions over type variables remain.

(MN) **Monotonicity**

$$\frac{K(\bar{\tau}) \sqsubseteq L(\bar{\sigma}), C_0 \cup C_T \Box \rho}{.., \tau_i \sqsubseteq \sigma_i, .., C_0 \cup C_T \Box \rho}$$

if $K \sqsubseteq_S L$ where $n \geq 0$.

(TCI) **Transitive Closure I**

$$\frac{\alpha \sqsubseteq \beta, \beta \sqsubseteq \tau, C_0 \cup C_T \Box \rho}{\alpha \sqsubseteq \beta, \beta \sqsubseteq \tau, \alpha \sqsubseteq \tau, C_0 \cup C_T \Box \rho}$$

if $\alpha \sqsubseteq \tau \notin C_0$.

(TCII) **Transitive Closure II**

$$\frac{\tau \sqsubseteq \beta, \beta \sqsubseteq \alpha, C_0 \cup C_T \Box \rho}{\tau \sqsubseteq \beta, \beta \sqsubseteq \alpha, \tau \sqsubseteq \alpha, C_0 \cup C_T \Box \rho}$$

if $\tau \sqsubseteq \alpha \notin C_0$.

(IN) **Infimum**

$$\frac{\alpha \sqsubseteq K(\bar{\sigma}), \alpha \sqsubseteq L(\bar{\tau}), C_0 \cup C_T \Box \rho}{\alpha \sqsubseteq I(\bar{\alpha}), .., \alpha_i \sqsubseteq \sigma_i, \alpha_i \sqsubseteq \tau_i, .., C_0 \cup C_T \Box \rho}$$

if $I = infimum(K, L)$ where α_i, $i = 1, .., n$ are new type variables and $n \geq 0$.

(SP) **Supremum**

$$\frac{K(\bar{\tau}) \sqsubseteq \alpha, L(\bar{\sigma}) \sqsubseteq \alpha, C_0 \cup C_T \Box \rho}{S(\bar{\alpha}) \sqsubseteq \alpha, .., \tau_i \sqsubseteq \alpha_i, \sigma_i \sqsubseteq \alpha_i, .., C_0 \cup C_T \Box \rho}$$

if $S = supremum(K, L)$ where α_i, $i = 1, .., n$ are new type variables and $n \geq 0$.

(LM) **Lower Matching**

$$\frac{\alpha \sqsubseteq L(\bar{\sigma}), C_0 \cup C_T \Box \rho}{(.., \alpha_i \sqsubseteq \sigma_i, .., C_0 \cup C_T \Box \rho)\mu}$$

where $K \sqsubseteq_S L$, $\mu = \{\alpha \leftarrow K(\bar{\alpha})\}$, α_i, $i = 1, .., n$ are new variables and $n \geq 0$.

(UM) **Upper Matching**

$$\frac{K(\bar{\tau}) \sqsubseteq \alpha, C_0 \cup C_T \Box \rho}{(.., \tau_i \sqsubseteq \alpha_i, .., C_0 \cup C_T \Box \rho)\mu}$$

where $K \sqsubseteq_S L$, $\mu = \{\alpha \leftarrow L(\bar{\alpha})\}$, α_i, $i = 1, .., n$ are new variables and $n \geq 0$.

(TR) **Trivial**

$$\frac{\alpha \sqsubseteq \alpha, C_0 \cup C_T \Box \rho}{C_0 \cup C_T \Box \rho}$$

(MV) **Matching Variables**

$$\frac{\alpha \sqsubseteq \beta, C_0 \cup C_T \Box \rho}{(C_0 \cup C_T \Box \rho)\mu}$$

where $\mu = \{\alpha \leftarrow \beta\}$.

Rules of matching ((LM) and (UM)) try non deterministically to match type variables to sorts and type constructors. This matching can include several successful choices. The strategy suggests to try elements as great as possible in order to capture type conditions that represent a bigger set of solutions. The rule (LM) (resp. (UM)) is not applicable if α can only be bound to a maximal type and K (resp. L) is not maximal. In its turn, the type variables α_i, must be bound to a maximal type. Rules for infimum and supremum ((IN) and (SP)) and transitive closure ((TCI) and (TCII)) are considered in order to prune the search tree.

4.3 Redundance Checking and Collection of Type Assumptions

(CDV) **Collection of data variables**

$$\frac{C_0 \cup X : \tau, C_T \Box \rho}{\alpha_X \sqsubseteq \tau, C_0 \cup [X : \alpha_X], C_T \Box \rho}$$

$\alpha_X \not\equiv \tau$.

(CF) **Collection of functional expressions**

$$\frac{C_0 \cup f(\bar{e}) : \tau, C_T \Box \rho}{\alpha_{f(\bar{e})} \sqsubseteq \tau, C_0 \cup [f(\bar{e}) : \alpha_{f(\bar{e})}], C_T \Box \rho}$$

$\alpha_{f(\bar{e})} \not\equiv \tau$.

(RD) **Redundance checking**

$$\frac{C_0 \cup f(\bar{e}) : \tau, C_T \Box \rho}{C_0 \cup C_T \Box \rho}$$

if $(\Sigma, \overline{C}_T) \vdash_{WTC} f(\bar{e}) : \tau$.

The rules are presented in the order they should be applied by the algorithm. Type conditions between brackets are introduced only if they do not occur yet in C_T. In practice, rule (RD) is expensive and it could not be implemented if the operational semantics considers sharing [Che93].

5 Soundness and Completeness

Previous algorithms of type inference for logic programming (cfr. [HiTo92], [Smo89]) are incomplete as it is observed in [Bei95]. This latter work presents too strong results of completeness for its algorithm as the following example shows.

Example 5 *Let S be a type signature containing $c \sqsubseteq_S a$ and $c \sqsubseteq_S b$ and let p be a predicate defined by the rule $p(X,Z) : -p(Y,Z)$ where $p : \alpha \times \alpha$. The Beierle's type inference obtains type assumptions for data variables of the form $X : \gamma, Y : \gamma, Z : \gamma$ where γ is a type variable. Since instances of the type declaration for the program rules are allowed, the prefix $X : a, Y : b, Z : c$ makes the rule well-typed. But the type assumptions obtained from the type inference algorithm do not capture this prefix (see strong completeness theorem [Bei95]).*

We have analogous situations as the following example shows. Let $\Leftarrow X == Z, Y == Z$ be a condition rule (X, Y and Z are extra variables) and the signature S of the previous example. The type inference algorithm that we have presented obtain the type conditions $X : \gamma, Y : \gamma, Z : \gamma$. As we have seen, the type conditions $X : a, Y : b, Z : c$ contain additional constraints and therefore they do not keep the solutions. The type inference algorithm does not make additional assumptions in order to obtain the well-typedness of the rule. Hence, in order to restrict the applicability of a rule, the user must introduce in C_U additional information.

The following example based on the signature S of previous examples shows the indeterminism of the type inference. Given a constant $constc : c$, we consider the condition rule $X == constc$ (X is an extra variable). The type inference algorithm will compute the two following minimal type conditions $X : a$ and $X : b$. This type conditions are not comparable. In the practice, we can use the symbol $\cup$ by considering $X : a \cup b$. This type condition expresses syntactically in the rule that X can be bound to a data of sort a or sort b. In any case, we consider more adequated to demand additional information to the user.

Soundness theorem concludes that every successful derivation obtains a C_T that makes the rule well-typed and derives the information given by the user.

Theorem 1 (Soundness) *Given $R \equiv f(t_1, .., t_n) := r \Leftarrow C_D \Box C_U$ a program rule (where C_U is a relevant set of type conditions for R), C_0 a relevant set of type conditions for R such that $C_0 \vdash_{WTC} C_U$ and $f(t_1, .., t_n) := r \Leftarrow C_D \Box C_0$ is a well-typed program rule. If $C_0 \cup \emptyset \Box [] \vdash^*_{TI} \emptyset \cup C_T \Box \rho$ then:*
- *$C_T \vdash_{WTC} C_0\rho$ and therefore $f(t_1, .., t_n) := r \Leftarrow C_D \Box C_T$ is a well-typed program rule*
- *$C_T \vdash_{WTC} C_U$.*

The type inference obtains a minimal set as the Completeness theorem shows. Completeness theorem does not fail as in [Bei95] because we are only interested in capturing type conditions that keep solutions.

Theorem 2 (Completeness) *Given $R \equiv f(t_1, .., t_n) := r \Leftarrow C_D \Box C_U$ a program rule (where C_U is relevant for R) and C_0 relevant for R such that $C_0 \vdash_{WTC} C_U$ and $f(t_1, .., t_n) := r \Leftarrow C_D \Box C_0$ is a well-typed program rule. If C_T is a minimal set for R and $\mu \in TSUST(S, \mathcal{X})$ (for the non frozen type variables) are such that $SSol(C_T) = SSol(C_0\mu)$ $[tvar(C_0\mu)]$ and $C_T \vdash_{WTC} C_0\mu$. Then there exists a minimal set C'_T for R and $\mu' \in TSUST(S, \mathcal{X})$ such that:*

- $C_0 \cup \emptyset \Box [] \vdash^*_{TI} \emptyset \cup C'_T \Box \mu'$
- $C'_T \vdash_{WTC} C_0\mu$
- $SSol(C'_T) = SSol(C_T)[tvar(C_0\mu)]$
- $\mu = \mu'[tvar(C_0)]$

Acknowledgements

I would like to thank to Ana Gil Luezas and Antonio Gavilanes Franco for useful comments, contributions and corrections to this work.

References

[Alm96] J.M.ALMENDROS JIMÉNEZ. *Type Inference and Checking for Polymorphic Order-Sorted Type Functional Logic Programs*, Technical Report DIA 31/96, Universidad Complutense, 1996.

[AGG96] J.M.ALMENDROS-JIMÉNEZ, A. GAVILANES-FRANCO, A. GIL-LUEZAS. *Algebraic Semantics for Functional Logic Programs with Polymorphic Order-Sorted Types*, Technical Report DIA 32/96, Universidad Complutense, 1996.

[AlGi96] J.M.ALMENDROS-JIMÉNEZ, A. GIL-LUEZAS. *Lazy Narrowing with Polymorphic Order-Sorted Types*, Technical Report DIA 30/96, Universidad Complutense, 1996.

[AGL94] P. ARENAS-SÁNCHEZ, A.GIL-LUEZAS, F.LÓPEZ-FRAGUAS. *Combining Lazy Narrowing with Disequality Constraints*, Procs. PLILP'94, Springer LNCS 844, pp. 385-399, 1994.

[Bei95] Ch. Beierle. *Type Inferencing for Polymorphic Order-Sorted Logic Programs*, Procs. of the 12th International Conference on Logic Programming, pp. 765-779, The MIT Press, 1995.

[Che93] P.H. Cheong. L. Fribourg. *Implementation of Narrowing: The Prolog-Based Approach*, In K.R. Apt, J.W. Bakker, J.J.M.M. Rutten, Eds. Logic Programming languages: Constraints, Functions and Objects. pp. 1-20. The MIT Press. 1993.

[DaMi82] L. Damas, R. Milner. *Principal Type-Schemes for Functional Programs*, Procs. 9th. Annual Symposium on Principles of Programming Languages, pp. 207-212, 1982.

[FuMi90] Y. Fuh, P. Mishra. *Type Inference with Subtypes*, Theoretical Computer Science, 73, pp. 155-175, 1990.

[GoMe92] J.A. Goguen, J. Meseguer. *Order Sorted Algebra I: Equational deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations* , Theoretical Computer Science, 105, pp. 217-273, 1992.

[GHLR96] J.C. González - Moreno, T. Hortalá - González, F. López - Fraguas, M. Rodríguez - Artalejo. *A Rewriting Logic for Declarative Programming*, Procs. ESOP'96. Springer LNCS 1058, pp. 156-172, 1996.

[Han89] M. Hanus. *Horn Clauses with Polymorphic Types: Semantics and Resolution*, Procs. TAPSOFT'89, Springer LNCS 352, Vol 2. pp. 225-240, 1989.

[Han90] M. Hanus. *A Functional and Logic Language with Polymorphic Types*, Procs. Int. Symp. on Design and Implementation of Symbolic Computation Systems, Springer LNCS 429, pp. 215-224, 1990.

[Han91] M. Hanus. *Parametric Order-Sorted Types in Logic Programming*, Procs. TAPSOFT'91, Springer LNCS 494, pp. 181-200, 1991.

[Han94] M. Hanus. *The Integration of Functions into Logic Programming: A Survey,* Journal of Logic Programming (19,20), Special issue "Ten Years of Logic Programming", pp. 583-628, 1994.

[HiTo92] P.M. Hill, R.W. Topor. *A Semantics for Typed Logic Programming*, Chapter 1, pp. 1-58, Types in Logic Programming, Logic Programming Series, Frank Pfenning Editor, The MIT Press, 1992.

[MoRo92] J.J. Moreno-Navarro, M. Rodríguez-Artalejo. *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming, 12, pp. 191-223, 1992.

[MyOk84] A. Mycroft, R.A. O'Keefe. *A Polymorphic Type System for Prolog*, Artificial Intelligence, 23, pp. 295-307, 1984.

[Smi94] G.S. Smith. *Principal Type Schemes for Functional Programs with Overloading and Subtyping*, Science of Computer Programming, 23, pp. 197-226, 1994.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*, PhD thesis, Universität Kaiserslaütern, Germany, 1989.