[**To appear in:** L. Sterling (ed): *Proceedings of the Twelfth International Conference on Logic Programming - ICLP'95*, Tokyo, Japan. MIT Press, 1995.]

Type inferencing for polymorphic order-sorted logic programs

Christoph Beierle
FernUniversität Hagen
Fachbereich Informatik
Bahnhofstr. 48
D-58084 Hagen, Germany
christoph.beierle@fernuni-hagen.de

Abstract

The purpose of this paper is to study the problem of complete type inferencing for polymorphic order-sorted logic programs. We show that previous approaches are incomplete even if one does not employ the full power of the used type systems. We present a complete type inferencing algorithm that covers the polymorphic order-sorted types in PROTOS-L, a logic programming language that allows for polymorphism as in ML and for hierarchically structured monomorphic types.

1 Introduction

It has often been argued that the lack of types in logic programming is a disadvantage from a software engineering point of view. There are now many different approaches for introducing types in logic programming, for an overview see [11]. In this paper we consider polymorphic types combined with order-sorted types, i.e. types which are hierarchically ordered. Polymorphism for logic programming was first suggested in [10]; for extensions of this approach to order-sorted types see [4], [6], [13], [8], [3].

In all of these approaches, a precondition for the well-definedness of the semantics of a typed logic program is that the program is well-typed. Thus, (static) type checking of programs is a precondition for the semantics to work properly, and usually type checking comes with the type inferencing done automatically to at least some degree. The purpose of this paper is to study the problem of complete type inferencing for polymorphic order-sorted logic programs. However, among the different approaches to type inferencing in logic programming (e.g. [9, 15, 14, 13, 8]) there are only a few dealing with type inferencing in an polymorphic order-sorted setting. Of the cited work, only [13] and [8] explicitly address this topic. However, there are problems with both approaches. As is already pointed out in [12, 13] the type inferencing algorithm presented there is incomplete. We will show in this paper that the type inferencing algorithm in [8] is also incomplete. Moreover, both problems still remain even if we consider only typed programs that do not exploit the full power of the respective typing

systems. Whereas TEL [12, 13] allows explicit subtype relationships between polymorphic types having different arities (e.g. $list(\alpha) \leq my\_type(\alpha, \beta)$), in [8] subtype relationships between polymorphic type are allowed only between type constructors of the same arity. The type system of PROTOS-L [3, 1] was derived form TEL by disallowing any explicit subtype relationship between polymorphic type constructors. The type inferencing algorithms of both [12, 13] and [8] are also incomplete for the PROTOS-L type system.

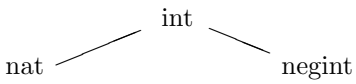
The type inferencing algorithm we present here covers the type system of PROTOS-L and can easily be extended to the type system of [8].

The paper is organized as follows: In the following section, we illustrate the special difficulties arising in type inferencing for polymorphic order-sorted logic programs and show the problems with previous approaches. In Section 3, polymorphic order-sorted types and the notion of well-typed programs are precisely defined as a basis for the type inferencing algorithm $\mathcal{TI}$ presented in Section 4, while Section 5 contains the corresponding correctness and completeness results.

2 Motivation and related work

The combination of order-sortedness and polymorphism causes some special difficulties. Having already derived different types for different occurrences of the same variable within a clause, the type inferencer must then "type-unify" these different types in order to get a consistent typing for the clause. If this involves the instantiation of a type variable there may be different choices from hierarchically ordered types.

Example 1 As an example, consider the type hierarchy



and the usual polymorphic type constructors `list(`α`)` and `pair(`α, β`)`. From a polymorphic predicate declaration `p : list(`α`)` and the subgoals

`... X = 1` $\wedge$ `p([X|L]) ...`

we might derive the simultaneous type requirements

`X:nat` `X:`α `L:list(`α`)`

since `1:nat`, and the type inferencer could (prematurely) instantiate α to `nat`. However, for the extended subgoal sequence

`... X = 1` $\wedge$ `p([X|L])` $\wedge$ `Y = -1` $\wedge$ `p([Y|L]) ...`

this type substitution is not possible. The derived type requirements

`X:nat` `X:`α `L:list(`α`)` `Y:negint` `Y:`α

are not consistent with the type instantiation α `= nat`, whereas α could still be instantiated to `int`. □

Typed logic programming with polymorphically order-sorted types as studied in [13] provides the theoretical foundations for TEL [12]. Given declarations for all predicates and functors, TEL's type inferencing component infers the types for the variables occurring in a clause. However, as already pointed out in [13], it is incomplete in the sense that it may fail on a clause involving polymorphic predicates although the clause could be well-typed, or that it computes a variable typing that is not most general. Whereas [13] gives examples where another ordering of the literals leads to a successful typing, the following example shows that for *no* ordering of the literals TEL's type inferencing algorithm succeeds although there is a well-typing.

Example 2 Consider the types as in Example 1 and a predicate with declaration

`member3:` $\texttt{negint} \times \alpha \times \texttt{list}(\alpha) \times \beta \times \texttt{list}(\beta)$

For the goal

`member3(X,2,[X|L],2,[Y|M])` $\wedge$ `member3(Y,2,[X|L],2,[Y|M])`

as well as for the rearranged goal

`member3(Y,2,[X|L],2,[Y|M])` $\wedge$ `member3(X,2,[X|L],2,[Y|M])`

TEL's type inferencing algorithm fails to generate a variable typing. However, there is a most general variable typing

{`X:negint, Y:negint, L:list(int), M:list(int)`}

under which both goals are well-typed. □

Another approach to type inferencing for polymorphic order-sorted types is presented in [8]. Its central component is the "most general type unifier" (mgtu) algorithm defined in Theorem 1.1.2 of [8], which is, however, incomplete as the following example shows.

Example 3 Consider the type hierarchy as in Example 1 and a predicate declaration `q : pair(`α`,`α`)`. The subgoals

`q(X)` $\wedge$ `X = mk_pair(1,-1)`

imply the simultaneous type requirements

`X:pair(`α, α`)` `X:pair(nat,negint)`

Then applying the mgtu algorithm to the resulting set of types

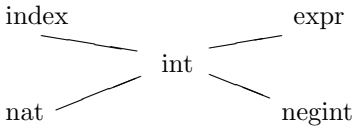
S = {`pair(`α`,`α`), pair(nat,negint)`}

returns a failure although $\{\alpha/\texttt{int}\}$ is a most general type unifier for S. □

Note that the mgtu algorithm of Theorem 1.1.2 is of central importance for the further development in [8]. Thus, the situation might be repaired by replacing the proof of Theorem 1.1.2 in [8] with a correct version of the mgtu algorithm. Unfortunately, however, no such algorithm exists since there are finite sets of unifiable types that do not have a "most general solution" in

the sense of [8]. This is in contrast to Lemma 1.1.10 of [8] which is falsified by the following counter-example.

Example 4 Let the type hierarchy of Example 1 be extended to



For the set of types $S = \{\texttt{pair(}\alpha\texttt{,}\alpha\texttt{), pair(nat,negint)}\}$ generated by the two type requirements of Example 3 there are 3 different type unifiers ($\{\alpha/\texttt{int}\}$, $\{\alpha/\texttt{index}\}$, $\{\alpha/\texttt{expr}\}$), but none of them is a most general type unifier (in the sense of [8]). □

Since all of the examples above do not use use any subtype relationships between polymorphic types, the type inferencing components of the cited approaches are still incomplete w.r.t. type systems that do not support explicit subtype relationships between polymorphic types. In the following, we will present a complete type inferencing algorithm for such polymorphic order-sorted type systems which is thus applicable to PROTOS-L and which can be extended to the type system of [8].

3 Typed logic programs

3.1 Types

A type alphabet T is a finite set of type symbols each of which comes with an arity ≥ 0. Type symbols with arity 0 are called monomorphic, type symbols with arity strictly greater than 0 are called polymorphic. T_{mono} (resp. T_{poly}) is the set of all monomorphic (resp. polymorphic) type symbols. T_{mono} comes with a partial order $\leq$; we will assume that $(T_{mono}, \leq)$ has a greatest lower bound $glb(s_1, s_2)$ for any two elements s_1 and s_2 having a lower bound at all (this ensures unitary unification; we will, however, do not deal with unification in this paper). Thus, since T_{mono} is finite also the least upper bound $lub(s_1, s_2)$ exists provided s_1 and s_2 have an upper bound at all.

V_{type} is a set of type variables. $Type$ is the set of types which are exactly the terms freely generated by the type symbols T over V_{type} in the usual way, i.e.

- every type variable α is a type
- every monomorphic type symbol s (also called type constant) is a type
- if η is a polymorphic type symbol with arity $n \geq 1$ and $\tau_1, \ldots, \tau_n$ are types, then $\eta(\tau_1, \ldots, \tau_n)$ is a type.

The partial order $(T_{mono}, \leq)$ is extended to a partial order $(Type, \leq)$ on the set $Type$ of all types as follows:

- for any type variable α we have $\alpha \leq \alpha$
- for any types $\eta(\tau_1, \ldots, \tau_n)$ and $\eta(\tau'_1, \ldots, \tau'_n)$ with $n \geq 1$ we have

  $$\eta(\tau_1, \ldots, \tau_n) \leq \eta(\tau'_1, \ldots, \tau'_n)$$

  iff $\tau_i \leq \tau'_i$ for any $i \in \{1, \ldots, n\}$

Thus, $\alpha \not\leq \beta$ for any $\alpha \neq \beta$ and $\eta(\ldots) \not\leq \eta'(\ldots)$ for any $\eta \neq \eta'$.

Since types are defined as terms, we can define a type substitution as an ordinary substitution on terms, i.e. as a morphism

$$\vartheta : Type \to Type$$

on the set of all type terms whose restriction to the set of type variables is the identity almost everywhere. We will represent type substitutions by

$$\vartheta = \{\alpha_1 \doteq \tau_1, \ldots, \alpha_n \doteq \tau_n\}$$

where $\{\alpha_1, \ldots, \alpha_m\}$ is the domain of ϑ.

Notation: In the rest of this paper we assume a fixed, but arbitrary type alphabet. We will use the following (possibly subscripted and dashed) symbols for denoting elements taken from the respective sets:

s	monomorphic type symbol
η	polymorphic type symbol
ξ	monomorphic or polymorphic type symbol
τ	type from $Type$
α, β	type variable
ϑ, ρ, σ	type substitution

3.2 Polymorphic order-sorted signature

A polymorphic order-sorted signature (over the set $Type$ of types) is a pair (**Func**, **Pred**) with:

- **Func** is a set of function declarations of the form

  $$f : \tau_1 \times \ldots \times \tau_n \to \tau_0$$

  with $\tau_i \in Type$, $n \geq 0$; τ_o is of the form $\xi(\alpha_1, \ldots, \alpha_m)$ with $m \geq 0$ and pairwise distinct variables α_j, and for any $i \in \{1, \ldots, n\}$ all variables of τ_i are contained in $\{\alpha_1, \ldots, \alpha_m\}$.

- **Pred** is a set of predicate declarations of the form

  $$p : \tau_1 \times \ldots \times \tau_n$$

  with $\tau_i \in Type$, for $i \in \{1, \ldots, n\}$.

Since we do not consider overloading we require that **Func** and **Pred** contain exactly one declaration for every function resp. predicate symbol. Moreover, in order to avoid the "empty set" problem [5] we assume that for any type there is a ground term of that type.

Let **Var** be an infinite set of variables. When considering from the type declaration of function and predicate symbols just their number of arguments, (untyped) first-order terms, literals, and formulas are freely generated by **Func** and **Pred** over **Var** in the usual way.

3.3 Well-typed Logic Programs

A *(type) prefix* P (or a *variable typing*) is a finite set of the form

$$P = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

with pairwise distinct variables x_i. The set $dom(P) = \{x_1, \ldots, x_n\}$ is called the *domain* of P. For a type substitution ϑ we let $\vartheta(P)$ denote the prefix

$$\{x_1 : \vartheta(\tau_1), \ldots, x_n : \vartheta(\tau_n)\}.$$

In the following, we use the notation:

P	type prefix
x	variable from **Var**
t	first order term built in the usual way over **Func** and **Var**

A well-typed term of type τ w.r.t. a type prefix P is defined by:

1. If $x : \tau$ is in P, then x is a well-typed term of type τ w.r.t. P.
2. If $f : \tau_1 \times \ldots \times \tau_n \to \tau_o$ is the declaration for f, σ is a type substitution, t_i is a well-typed term of type $\tau_i' \leq \sigma(\tau_i)$ w.r.t. P for $i \in \{1, \ldots, n\}$, then $f(t_1, \ldots, t_n)$ is a well-typed term of type τ w.r.t. P for $\tau = \sigma(\tau_o)$.

Likewise, we define well-typed clauses w.r.t. a prefix P:

1. If $p : \tau_1 \times \ldots \times \tau_n$ is the declaration for p, σ is a type substitution, t_i is a well-typed term of type $\tau_i' \leq \sigma(\tau_i)$ w.r.t. P for $i \in \{1, \ldots, n\}$, then $p(t_1, \ldots, t_n)$ is a well-typed atomic formula w.r.t. P.
2. If $F, F_1, \ldots, F_n$ are well-typed atomic formulas w.r.t. P then

   $$F \leftarrow F_1 \wedge \ldots \wedge F_n.$$

   is a well-typed clause w.r.t. P.

As usual, a logic program *Prog* is a set of clauses. *Prog* is well-typed if each of its clauses is well-typed, and a clause is well-typed if it is well-typed w.r.t. some prefix. [1]

The problem of type inferencing for logic programs can thus be stated precisely as follows:

[1] Note that this notion of well-typedness does not ensure the head-condition (the head of a clause must belong to a type that is a *variant* - rather than an instance - of the head predicate's declaration). However, only some of the approaches to polymorphic order-sorted logic programming require the head condition (e.g. [10], [13], [8]), but not all of them (e.g. [7]). Once one has established a (most general) well-typing for a clause it is easy to check whether the head condition is satisfied.

- Given a program *Prog* decide whether *Prog* is well-typed, i.e. for any of its clauses *Cl* find a prefix P such that *Cl* is well-typed w.r.t. P.

As argued in Section 2, there are clauses that do not have a most general well-typing in the sense that any other well-typing can be obtained from it by further instantiation. Therefore, the type inferencing algorithm $\mathcal{TI}$ presented in the next section produces a (representation of a) set of well-typings. $\mathcal{TI}$ is correct and complete in the following sense: It identifies exactly all clauses that can not be well-typed, and for any well-typing prefix P for a clause *Cl* it derives a well-typing prefix P' such that P' is more general than P, i.e. there is a type substitution ρ such that $\rho(P') = P$.

4 Type inferencing

Given a clause

$$Cl = L_o \leftarrow L_1 \wedge \ldots \wedge L_n$$

we must find a prefix P such that for any $i = \{0, \ldots, n\}$, L_i is a well-typed atomic formula w.r.t. P. The type inferencing procedure $\mathcal{TI}$ to be presented in this section will use the following notion of type constraint.

Definition 5 (type constraint) A *type constraint* is one of the following:

a subtype constraint: $\tau \leq^{\iota} \tau'$

a monomorphic set constraint: $\alpha :: M$

where $M \subseteq T_{mono}$. A subtype constraint is called *basic* if it is of the form $\alpha \leq^{\iota} s, s \leq^{\iota} \alpha$, or $\alpha \leq^{\iota} \beta$. A type substitution ϑ *satisfies* the type constraint

$\tau \leq^{\iota} \tau'$ if $\vartheta(\tau) \leq \vartheta(\tau')$

$\alpha :: M$ if $\vartheta(\alpha) \in M$ □

With this definition, the type inferencing procedure $\mathcal{TI}$ consists of the following 5 steps:

1. Transform *Cl* into a set R of type constraints.
2. From R infer a set R' of basic type constraints and a corresponding type substitution ϑ.
3. Split R' into polymorphic and monomorphic type constraints, yielding R_{poly} and R_{mono}.
4. Solve R_{poly} by extending ϑ, yielding ϑ'.
5. Solve R_{mono} by
   - 5.1 constraint propagation, yielding R'_{mono},
   - 5.2 choosing solutions for R'_{mono} by extending ϑ', yielding ϑ''.

If these steps are carried out successfully, *Cl* is well-typed w.r.t. the prefix

(1) $$\frac{p(t_1,\ldots,t_n)\ \&\ M,R}{t_1:\tau\ \&\ \ldots\ \&\ t_n:\tau_n\ \&\ M,R}$$ if $p:\tau_1\times\ldots\times\tau_n$ is a variant of the declaration of p

(2) $$\frac{f(t_1,\ldots,t_n):\tau\ \&\ M,R}{t_1:\tau_1\ \&\ \ldots\ \&\ t_n:\tau_n\ \&\ M,\tau_o\leq^{\imath}\tau\ \&\ R}$$ if $f:\tau_1\times\ldots\times\tau_n\rightarrow\tau_o$ is a variant of the declaration of f

(3) $$\frac{x:\tau\ \&\ M,R}{M,\alpha_x\leq^{\imath}\tau\ \&\ R}$$ where α_x is a unique type variable associated to x

Figure 1: Computing type constraints from terms and literals

$$P=\{\,x:\vartheta''(\alpha_x)\mid x\ \textit{is a variable in}\ Cl\,\}$$

where α_x is a unique type variable associated to x.

In the following five subsections, we will give precise definitions of these five steps. Each step will be defined by a terminating set of transformation rules that are applied exhaustively to the respective input configuration.

4.1 Computation of subtype constraints

In order to ease our notation we will use & as an associative, commutative, and idempotent operator for sets. The pair $(M,\emptyset)$ with $M = L_o\ \&\ \ldots\ \&\ L_n$ is our starting configuration to which the transformation rules of Figure 1 are applied.

Rule (1) introduces for every argument t of a literal a *type membership requirement* of the form $t{:}\tau$. The idea is that a prefix P satisfies such a requirement if it renders t to be of a subtype of an instance of τ, i.e. if there is ϑ and τ' with $\tau'\leq^{\imath}\vartheta(\tau)$ such that t is a well-typed term of type τ' w.r.t. P. (c.f. the definition of well-typed atomic formula is Section 3.3).

These type membership requirements are decomposed by rule (2) into type membership requirements for the subterms and a subtype constraint for the type of the term according to the definition of well-typed term. Rule (3) finally transforms the remaining type membership requirements for the variables into corresponding subtype constraints where for every variable x a unique associated type variable α_x is introduced.

It is easy to show that the rules of Figure 1 terminate, thereby eliminating M and yielding the pair

$$\emptyset, R$$

where R is a set of subtype constraints.

4.2 Inferring basic subtype constraints

Input to our second set of transformation rules given in Figure 2 is the pair

(1) $$\frac{\alpha \leq^{\iota} \alpha \ \&\ R, \vartheta}{R, \vartheta}$$

(2) $$\frac{\tau_1 \leq^{\iota} \tau_2 \ \&\ \tau_2 \leq^{\iota} \tau_3 \ \&\ R, \vartheta}{\tau_1 \leq^{\iota} \tau_2 \ \&\ \tau_2 \leq^{\iota} \tau_3 \ \&\ \tau_1 \leq^{\iota} \tau_3 \ \&\ R, \vartheta} \quad \text{if } \tau_1 \leq^{\iota} \tau_3 \notin R$$

(3) $$\frac{\alpha \leq^{\iota} \tau \ \&\ \tau \leq^{\iota} \alpha \ \&\ R, \vartheta}{\rho(R), \rho \circ \vartheta}$$ if α does not occur in τ and where ρ is the type substitution $\{\alpha \doteq \tau\}$

(4) $$\frac{s_1 \leq^{\iota} s_2 \ \&\ R, \vartheta}{R, \vartheta}$$ if $s_1 \leq s_2$

(5) $$\frac{s_1 \leq^{\iota} \alpha \ \&\ s_2 \leq^{\iota} \alpha \ \&\ R, \vartheta}{s \leq^{\iota} \alpha \ \&\ R, \vartheta}$$ if $s = lub(s_1, s_2)$

(6) $$\frac{\alpha \leq^{\iota} s_1 \ \&\ \alpha \leq^{\iota} s_2 \ \&\ R, \vartheta}{\alpha \leq^{\iota} s \ \&\ R, \vartheta}$$ if $s = glb(s_1, s_2)$

(7) $$\frac{\eta(\tau_1, \ldots, \tau_n) \leq^{\iota} \eta(\tau'_1, \ldots, \tau'_n) \ \&\ R, \vartheta}{\tau_1 \leq^{\iota} \tau'_1 \ \&\ \ldots \ \&\ \tau_n \leq^{\iota} \tau'_n \ \&\ R, \vartheta}$$

(8) $$\frac{\eta(\tau_1, \ldots, \tau_n) \leq^{\iota} \alpha \ \&\ R, \vartheta}{\tau_1 \leq^{\iota} \alpha_1 \ \&\ \ldots \ \&\ \tau_n \leq^{\iota} \alpha_n \ \&\ \rho(R), \rho \circ \vartheta}$$ if α does not occur in $\tau_1, \ldots, \tau_n$ and where $\alpha_1, \ldots, \alpha_n$ are new, pairwise distinct variables and ρ is the type substitution $\{\alpha \doteq \eta(\alpha_1, \ldots, \alpha_n)\}$

(9) $$\frac{\alpha \leq^{\iota} \eta(\tau_1, \ldots, \tau_n) \ \&\ R, \vartheta}{\alpha_1 \leq^{\iota} \tau_1 \ \&\ \ldots \ \&\ \alpha_n \leq^{\iota} \tau_n \ \&\ \rho(R), \rho \circ \vartheta}$$ if α does not occur in $\tau_1, \ldots, \tau_n$ and where $\alpha_1, \ldots, \alpha_n$ are new, pairwise distinct variables and ρ is the type substitution $\{\alpha \doteq \eta(\alpha_1, \ldots, \alpha_n)\}$

Figure 2: Transformation and simplification of type requirements

$$R, id$$

with id denoting the identity type substitution.

Rule (1) eliminates trivial constraints. Rule (2) computes the transitive closure of $\leq^{\iota}$. Rule (3) exploits the fact that $\leq^{\iota}$ obviously is antisymmetric w.r.t. to any solution. Rule (4) eliminates subtype constraints between monomorphic types that hold trivially. Rule (5) and (6) replace a common monomorphic upper bounds (resp. lower bound) by the least upper bound (resp. greatest lower bound). Rule (7) decomposes a subtype constraint between complex types. Rules (8) and (9) replace a subtype constraint

between a type variable α and a complex type τ by an instantiation α and a set of simplified subtype constraints involving the arguments of τ.

Let the pair

$$R', \vartheta$$

be the result of applying the rules (1) - (9) exhaustively to (R, id). A *failure condition* in R' is:

$s_1 \leq^\iota s_2$	if $s_1 \not\leq s_2$
$s_1 \leq^\iota \alpha \ \& \ s_2 \leq^\iota \alpha$	if $lub(s_1, s_2)$ does not exist
$\alpha \leq^\iota s_1 \ \& \ \alpha \leq^\iota s_2$	if $glb(s_1, s_2)$ doest not exist
$\eta(\tau_1, \ldots, \tau_n) \leq^\iota \eta'(\tau'_1, \ldots, \tau'_m)$	if $\eta \neq \eta'$
$\eta(\tau_1, \ldots, \tau_n) \leq^\iota \alpha$	if α occurs in $\tau_1, \ldots, \tau_n$
$\alpha \leq^\iota \eta(\tau_1, \ldots, \tau_n)$	if α occurs in $\tau_1, \ldots, \tau_n$

Proposition 6 If R' contains a failure condition, then there is no prefix P such that Cl is well-typed w.r.t. P. If R' does not contain any of the above conditions, then R' contains only basic type constraints and is said to be in *solved form.* □

4.3 Distinguishing polymorphic and monomorphic constraints

From the basic subtype constraints in R' we can now easily determine which type variables must be instantiated to a monomorphic type constant. This is obviously the case for $\alpha \leq^\iota s$ or $s \leq^\iota \alpha$, but also for every α' with a (direct or indirect) subtype constraint w.r.t. α. Thus, we split R' into two dispoint subsets R_{mono} and R_{poly}:

- R_{mono} is the smallest subset of R' with

$\alpha \leq^\iota s \in R_{mono}$	if $\alpha \leq^\iota s \in R'$
$s \leq^\iota \alpha \in R_{mono}$	if $s \leq^\iota \alpha \in R'$
$\alpha \leq^\iota \alpha' \in R_{mono}$	if $\alpha \leq^\iota \alpha' \in R'$ and α or α' occurs in R_{mono}

- $R_{poly} := R' \backslash R_{mono}$

Proposition 7 Any well-typing for Cl must instantiate every variable occurring in R_{mono} to a monomorphic type constant, whereas any variable occurring in R_{poly} may remain a type variable. □

4.4 Solving polymorphic subtype constraints

We can sharpen the previous proposition with regard to the variables in R_{poly}. In fact, in any well-typing that is "most general" (in the sense that as few as possible type variables are instantiated to type terms that are as

(1) $$\frac{\alpha \leq^{\iota} \beta \ \&\ R_{poly}, \vartheta}{\rho(R_{poly}), \rho \circ \vartheta}$$ where ρ is the type substitution $\{\alpha \doteq \beta\}$

(2) $$\frac{\alpha \leq^{\iota} \alpha \ \&\ R_{poly}, \vartheta}{R_{poly}, \vartheta}$$

Figure 3: Solving polymorphic type constraints

"small as possible", see Section 5) every variable α in R_{poly} will indeed remain a variable, only identified with another type variable β if there is a subtype relationship between α and β in R_{poly}. Thus, the subtype requirements in R_{poly} are solved by applying the rules of Figure 3 to the pair

$$(R_{poly}, \vartheta)$$

Since R_{poly} contains only basic type constraints between type variables we can prove that this will always resolve all elements in R_{poly}, leading to a pair $(\emptyset, \vartheta')$.

4.5 Solving monomorphic subtype constraints

We are now left with the pair (R_{mono}, ϑ') where for each variable in R_{mono} a monomorphic type constant from T_{mono} must be found such that the constraints in R_{mono} are satisfied. This process is carried out in two steps. First the monomorphic set constraints in R_{mono} are propagated to all variables as far as possible (4.5.1). Whereas so far all steps have been carried out deterministically, the final step nondeterministically chooses monomorphic type instantiations from the remaining possibilities (4.5.2).

4.5.1 Propagating monomorphic set constraints

For any variable α in R_{mono} we can safely add the initialization constraint $\alpha :: T_{mono}$. Therefore, the rules of Figure 4 are applied to

$$R_{mono} \cup \{\alpha :: T_{mono} \mid \alpha \text{ occurs in } R_{mono}\} \qquad (*)$$

Rules (1) and (2) evaluate a subtype restriction between a type variable and a monomorphic constant. Two monomorphic set constraints for the same type variable can be simplified to a single one by taking the respective intersection (rule (3)). Rule (4) allows to sharpen the monomorphic set constraints of two variables that must be instantiated to subtypes of each other: If $\alpha \leq^{\iota} \alpha'$ then the set M of possible type constants for α can be restricted to those elements for which there is some supertype in the set M' of possible type constants for α'. More formally, the set descriptions used in rule (4) are defined by

$$(1) \qquad \frac{\alpha \leq^{\imath} s \ \&\ R}{\alpha :: \{s' \in T_{mono} \mid s' \leq s\} \ \&\ R}$$

$$(2) \qquad \frac{s \leq^{\imath} \alpha \ \&\ R}{\alpha :: \{s' \in T_{mono} \mid s \leq s'\} \ \&\ R}$$

$$(3) \qquad \frac{\alpha :: M \ \&\ \alpha :: M' \ \&\ R}{\alpha :: (M \cap M') \ \&\ R}$$

$$(4) \qquad \frac{\alpha :: M \ \&\ \alpha' :: M' \ \&\ \alpha \leq^{\imath} \alpha' \ \&\ R}{\alpha :: M_{|\leq M'} \ \&\ \alpha' :: M'_{|\geq M} \ \&\ \alpha \leq^{\imath} \alpha' \ \&\ R} \qquad \text{if } M \neq M_{|\leq M'} \text{ or } M' \neq M'_{|\geq M}$$

Figure 4: Propagating monomorphic set constraints

$$\begin{aligned} M_{|\leq M'} &:= \{s \in M \mid \exists s' \in M'.s \leq s'\} \\ M_{|\geq M'} &:= \{s \in M \mid \exists s' \in M'.s \geq s'\} \end{aligned}$$

Let R'_{mono} be the result of applying these rules exhaustively to (*). A *failure condition* in R'_{mono} is of the form

$$\alpha :: \emptyset$$

since there is no possible instantiation for α.

Proposition 8 If R'_{mono} contains a failure condition, then there is no prefix P such that Cl is well-typed w.r.t. P. □

Therefore, for the last step let us assume that R'_{mono} does not contain a failure condition.

4.5.2 Choosing monomorphic types

We have now obtained the pair (R'_{mono}, ϑ') where R'_{mono} contains a monomorphic set restriction $\alpha :: M$ for every variable α occurring in it, together with subtype constraints of the form $\alpha_1 \leq^{\imath} \alpha_2$. This pair (R'_{mono}, ϑ') represents a set of solutions to our type inferencing problem: Instantiating any such α by an element of its associated set M such that the subtype constraints are satisfied yields a well-typing prefix for Cl. Moreover, this pair is a *minimal* representation of all (most general) well-typings of Cl in the following sense: For any such $\alpha :: M$ we can select *any* $s \in M$ and still yield a well-typing under this selection.

Therefore, whereas all previous steps of $\mathcal{TI}$ were carried out deterministically, the final inferencing step non-deterministically chooses a type variable together with a possible monomorphic type constant by applying the rule of Figure 5 successively to the pair

(1) $\dfrac{\alpha :: \{s_1, \ldots, s_n\} \ \&\ R_{mono}, \vartheta}{R'_{mono}, \rho \circ \vartheta}$

where ρ is the type substitution $\{\alpha \doteq s\}$, s is any of monomorphic types $s_1, \ldots, s_n$, and R'_{mono} results from $\rho(R_{mono})$ by exhaustively applying the rules of Figure 4

Figure 5: Choosing monomorphic type instantiations

R'_{mono}, ϑ'

Exhaustively applying the propagation rules of Figure 4 after each selection ensure that no failure condition will be generated.

Proposition 9 During the propagation of constraints initiated by the selection rule in Figure 5 no failure condition of the form $\alpha :: \emptyset$ will be generated. □

Thus, applying the selection rule terminates with a pair $(\emptyset, \vartheta'')$. ϑ'' is said to be a type substitution *generated by the type inferencing algorithm* $\mathcal{TI}$, and

$$P_{\vartheta''} := \{\, x : \vartheta''(\alpha_x) \mid x \textit{ is a variable in } Cl \,\}$$

is a prefix inferred by $\mathcal{TI}$.

Example 10 For the goal in Example 2, $\mathcal{TI}$ infers the type prefix

{`X:negint, Y:negint, L:list(int), M:list(int)`}

□

Example 11 Consider the situation of Example 4. After execution of step 5.1 in $\mathcal{TI}$ the set R'_{mono} contains

$\alpha ::$ {`int, index, expr`}

and thus each of {`X:int`}, {`X:index`}, and {`X:expr`} is a prefix inferred by $\mathcal{TI}$. □

5 Correctness and completeness of type inferencing

We will now precisely state our correctness and completeness results for the type inferencing procedure $\mathcal{TI}$. Due to lack of space the complete proofs can not be given here, but are given in the full version of this paper [2]. The proofs use the propositions given in the previous section; the termination proof is by induction on the involved terms.

We assume the notation of the previous section; in particular for $Cl, R, R', R_{poly}, R_{mono}, \vartheta, \vartheta'$ and ϑ''.

Theorem 12 (Correctness of successful derivations) *If no failure condition occurs, then Cl is well-typed w.r.t. to the prefix* $P_{\vartheta''}$.

Theorem 13 (Correctness of derivation failures) *If a failure condition occurs, then there is no prefix P such that Cl is well-typed w.r.t. P.*

Theorem 14 (Termination/Completeness of type inferencing) *The type inferencing algorithm always terminates, i.e. each of the steps 1 to 5 terminates if all preceding steps have been carried out successfully.*

The three theorems above imply the correctness and completeness of $\mathcal{TI}$. However, we can even prove a stronger completeness result. The completeness result above says that whenever there is a well-typing prefix P then $\mathcal{TI}$ will derive a type substitution and thus some well-typing prefix P'. The strong completeness of $\mathcal{TI}$ ensures that for any such P $\mathcal{TI}$ derives a well-typing prefix P' such that P' is more general than P.

Theorem 15 (strong completeness) *If there is a prefix P such that Cl is well-typed w.r.t. P then $\mathcal{TI}$ derives a type substitution ϑ_P such that Cl is well-typed w.r.t. P_{ϑ_P} and there is a type substitution ρ such that $\rho(P_{\vartheta_P}) = P$.*

6 Conclusions and further work

In this paper we have studied automatic type inferencing for polymorphic order-sorted logic programs. After pointing out the difficulties with previous approaches we have presented a complete type inferencing algorithm for this problem. Whereas we addressed the correctness and completeness issues, we did not deal with efficiency or complexity matters in this paper. For instance, in an implementation of $\mathcal{TI}$ one would look for an efficient representation of the monomorphic set constraints $\alpha :: M$. An obvious choice for the representation of M would be the representation by its lower and upper bounds. Another aspect not yet studied in this paper that needs further investigation is the relationship of our approach to the work on polymorphic type inference done for functional programming languages.

Acknowledgements

I would like to thank Gregor Meyer for his comments on previous versions of this paper as well as the anonymous referees for their valuable hints.

References

[1] C. Beierle. Concepts, implementation, and applications of a typed logic programming language. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 5, pages 139–167. Elsevier Science B.V./North-Holland, Amsterdam, Holland, 1995.

[2] C. Beierle. Type inferencing for logic programming with polymorphic order-sorted types. Informatik-Bericht, FB Informatik, FernUniversität Hagen, 1995. (to appear).

[3] C. Beierle and G. Meyer. Run-time type computations in the Warren Abstract Machine. *Journal of Logic Programming*, 18(2):123–148, February 1994.

[4] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proceedings of the 2nd European Symposium on Programming*, Lecture Notes in Computer Science, pages 79–93, Berlin, 1988. Springer-Verlag.

[5] J.A. Goguen and J. Meseguer. Remarks on Remarks on Many-Sorted Equational Logic. In *Bulletin of the EATCS* , number 30, 1986.

[6] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In *Proceedings TAPSOFT'89*. Springer-Verlag, 1989.

[7] M. Hanus. Logic programming with type specifications. In F. Pfenning, editor, *Types in Logic Programming*. MIT Press, 1992.

[8] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*. MIT Press, 1992.

[9] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, 1984.

[10] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[11] F. Pfenning. *Types in Logic Programming*. MIT Press, Cambridge, MA, 1992.

[12] G. Smolka. TEL (Version 0.9), Report and User Manual. SEKI-Report SR 87-17, FB Informatik, Universität Kaiserslautern, 1988.

[13] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, FB Informatik, Univ. Kaiserslautern, 1989.

[14] J. Xu and D. S. Warren. A type inference system for Prolog. In K. A. Bowen and R. A. Kowalski, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 604–619, Seattle, Washington, August 1988. MIT PRESS.

[15] J. Zobel. Derivation of polymorphic types for Prolog programs. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 817–838, Melbourne, May 1987. MIT PRESS.