A Polymorphic Type System with Subtypes for Prolog*

Roland Dietrich, Frank Hagl

GMD Forschungsstelle an der Universität Karlsruhe
(German National Research Center for Computer Science)
Haid-und-Neu-Straße 7, D-7500 Karlsruhe

Abstract. From a software engineering point of view, logic programming lacks many properties allowing secure development of large programs by many programmers. In this paper we try to improve these properties by proposing a polymorphic type system for Prolog. Our type system is able to deal with subtype relations if information about dataflow within clauses is available. This information can be given by a mode system. We give an outline of a type checking algorithm for this type system and discuss several problems which do not arise in type systems without subtypes.

1. Introduction

Despite its good rapid prototyping quality, most logic programming systems lack certain properties which are essential when developing large programs by several programmers. Especially in Prolog, there is no possibility to formally specify what kind of arguments a predicate is intended to have, and the input/output behaviour cannot be specified although most predicates of a program are used with a fixed input/output mode (cf. [Drabent 87]). Such informations are useful for several reasons: They provide a good deal of documentation; in a module system they can be part of the interface description of a module; they make the discovery of many typing errors easier; and a Prolog compiler can use them for optimization purposes.

The notion of types has a long tradition in programming languages. Especially, polymorphic type systems have been broadly studied in the functional programming area (see [Cardelli, Wegener 85] for a survey). Imposing type structures to first order logic resulted in many sorted logic for which appropriate deduction mechanism have been developed, for example [Walther 83, Schmidt-Schauss 86]. This also enabled the use of logic as a typed programming language including subtype structures (order sorted logic). Examples are [Goguen, Meseguer 84] and [Smolka, Nutt 87]. In [Huber, Varsek 87] order sorted resolution is used instead of ordinary resolution as an execution model for logic programming including an adaptation of Warren's abstract Prolog machine [Warren 83] to this model. All these approaches to types in logic programming do not allow parametric polymorphism and the type information directly impacts the deduction mechanism which often results in a considerable restriction of the search space.

In contrast, our approach considers a purely static type discipline which leaves the operational semantics of logic programs completely unchanged. It is based on the type system of [Mycroft, O'Keefe 84] which itself is an adaptation of Milner's type system for ML to Prolog. This type system lacks the possibility to deal with subtype relations, which makes some programs impossible to

* This work has been done within a joint project of GMD and SFB 314 (artificial intelligence) at the University of Karlsruhe.

be statically type checked, even if they behave well at runtime. We extend the Mycroft/O'Keefe system to deal with subtypes, when information about the possible dataflow in a clause is available. This dataflow information can be provided by a mode system [Warren 77, Bruynooghe 82, Smolka 84, Dietrich 88], or by global analysis [Mellish 85].

Type checking can be performed completely at compile time, that is, type informations do not change but rather document the runtime behaviour of programs. Therefore a programmer is not forced to fully declare his program. A type-checker is able to derive missing type information to some extent, and it is also possible to develop programs in a rapid prototyping manner adding type specifications later in order to increase the programming security, documentation quality, and reusability of these programs. A somewhat different view of typing and type checking a Prolog program is taken in [Mishra 84] and [Zobel 87], where automatic type inference is used to detect goals which can never succeed.

The paper is organized as follows: After introducing some basic notions in section 2, we recall the type system of Mycroft/O'Keefe and discuss its advantages and disadvantages in section 3. In section 4 we show how it can be extended to deal with subtypes, and we give an outline of the type checking algorithm in section 5. Some conclusions and open problems are considered in section 6.

2. Formal Preliminaries

We assume familiarity of the reader with the basic notions of first order logic and the foundations of logic programming. In this section, we only recall some often used notions.

Programs. We assume the following sets to be given:

- $\mathbf{V}$, a countable, infinite set of *variables* ($x, y, z, \dots \in \mathbf{V}$).
- $\mathbf{F}$, a countable set of *function symbols* with arity ($f/n \in \mathbf{F}$).
- $\mathbf{P}$, a countable set of *predicate symbols* with arity ($p/m \in \mathbf{P}$).

Based on these sets, *terms*, *atomic formulae (atoms)*, *literals* and (*Horn- clauses*) are defined as usual in first order logic. Horn clauses may be *facts* ($H \leftarrow$), *implications* ($H \leftarrow B_1, \dots, B_m$) and *goals* ($\leftarrow B_1, \dots, B_m$). If S is a term, an atom, a clause or a set of those, $\text{var}(S)$ denotes the set of variables occurring in S . A *program* is a finite set of facts and implications. Because we restrict ourselves to *Horn* logic programming, throughout the paper a *clause* is always meant to be a *Horn clause*.

Substitutions. A *Substitutions* can be denoted by a finite set of variable replacements ($\theta = \{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$). Of a and b are terms or atoms, $\text{mgu}(a, b)$ denotes the set of most general unifiers for a and b , which may be empty or contains one element up to renaming of variables. " $\preceq$ " denotes the *subsumption ordering* on terms: $s \preceq t$ iff a substitution θ exists such that $\theta s = t$. In this case we say that s is *more general* than t or t is an *instance* of s by θ . We say that s is a *renaming* of t ($s \simeq t$) iff $s \preceq t$ and $t \preceq s$. That is, s and t only differ in the names of their variables.

SLD-derivations and answer substitutions. Horn logic programs usually are interpreted using the resolution principle, especially linear resolution with selection function. This yields the notion of *SLD-derivations*, *SLD-refutations* (SLD-derivations of the empty clause) and *answer substitutions*. For a comprehensive presentation of these topics see [Lloyd 1984].

Occurrences. In order to refer precisely to subterms or atoms occurring in clauses, we recall the well-known notion of occurrences of terms and extend it to occurrences of atoms and clauses. Occurrences of terms are usually defined as sequences of natural numbers. Occurrences of atoms can be defined in the same way, because their syntactical structure is the same one. Occurrences of clauses can be obtained by prefixing occurrences of atoms with a number, indicating the position where the atom occurs in the clause¹. That is, if c is a clause then $c/0$ denotes the clause head and c/i denotes the i -th literal of the body for $i \geq 1$. For example, if $c = p(f(X), g(Y, X)) \leftarrow q(X), p(X, Y)$, then $c/0.1 = f(X)$, $c/0.2.1 = Y$ and $c/2.2 = Y$. For any syntactic object s $O(s)$ denotes the *set of occurrences* of s .

3. The Type System of Mycroft/O'Keefe

In this section we give an informal outline of the type system defined in [Mycroft, O'Keefe 84] by means of some examples, and we discuss its advantages and drawbacks. The type system is based on Milner's work on polymorphic types for functional programming languages [Milner 78]. A formal description can easily be extracted from the definition of its extension to deal with subtypes given in section 4.

A programmer can define *typings for function and predicate symbols* occurring in his program via *type declarations* and *predicate declarations*. Types itself are terms over a special alphabet disjoint from the program's alphabet. Types are *polymorphic*, if they contain type variables. Typings of n -ary function symbols are of the form $(\tau_1, \dots, \tau_n \longrightarrow \tau)$, where τ and τ_i are types. τ_i are called *argument types* and τ *result type*. Typings for n -ary predicate symbols are of the form $(\tau_1, \dots, \tau_n)$, where τ_i are types.

The following example for a general list type and natural numbers and the predicates *append/3*, *reverse/3*, *member/2*, and *length/2* shows a possible Prolog syntax for such declarations:²

```
:- type list( $\alpha$ ) => nil, .( $\alpha$ , list( $\alpha$ )).
:- type nat => 0, s(nat).
:- pred append(list( $\alpha$ ), list( $\alpha$ ), list( $\alpha$ )).
:- pred length(list( $\alpha$ ), nat).
```

If we denote typings of (function and predicate) symbols by superscribing the symbols with its type, these declarations induce the following typing

$$\{ \text{nil}^{\text{list}(\alpha)}, \text{.}^{\alpha, \text{list}(\alpha) \rightarrow \text{list}(\alpha)}, 0^{\text{nat}}, \text{s}^{\text{nat} \rightarrow \text{nat}}, \text{append}^{\text{list}(\alpha), \text{list}(\alpha), \text{list}(\alpha)}, \text{length}^{\text{list}(\alpha), \text{nat}} \}$$

A *typing for a clause* associates a type with every occurrence of the clause, which means that the term at this occurrence is of the associated type. *Type checking* a clause consists in *deriving* a typing for a clause, such that every occurrence of the clause is associated with a type which is compatible with the given typing of symbols. For example, from the above typing we can derive the

1 Interpreting a clause as a term with binary function symbols ' $\leftarrow$ ' and ' $,$ ' and using occurrences of terms would be quite unreadable.

2 We use the dot-notation to abbreviate the cons-operator. Type variables are denoted by greek letters. They may be replaced by ordinary Prolog variables in practice.

following typings for *append*-clauses:

```
append(nillist(α), Llist(α), Llist(α)).
append((Xα, L1list(α)), L2list(α), L3list(α)) :- append(L1list(α), L2list(α), L3list(α)).
```

If for every occurrence a compatible type and, especially, for every variable (for which no typings must be defined) a *unique* type can be derived, the clause is said to be *well-typed*. This is formally defined by *well-typing rules* in [Mycroft, O'Keefe 84]. If different types for the same variable are derived, these types must be *unifiable* in order to preserve well-typedness. This can effect instantiation of type variables in other parts of the clause, too. For example type checking the goal clause

```
:- length(nil, X), append((X, nil), L, LL).
```

the variable X is first associated with type nat (by the declaration of $length^{list(α), nat}$) and then, by the declaration of $.^{α, list(α)}$, with type $α$, which yields the (type) substitution $α \leftarrow nat$. This substitution expresses among others that *append* is used with type $(list(nat), list(nat), list(nat))$ in this clause.

If a program is well-typed, it is guaranteed that during runtime no type conflicts can happen. That is, every resolvent produced during a refutation of a goal is also well-typed, and every variable of the goal is instantiated according to its associated type by any answer substitution. This is paraphrased by "well-typed programs can't go wrong" in [Mycroft, O'Keefe 84].

The type system of Mycroft and O'Keefe has several advantages: The typing information describes very closely the intention of the programmer when defining predicates. This increases the documentation facilities of Prolog. The type-checker can detect a lot of typing errors. The polymorphism enables the programmer to use instantiations of previously defined types in predicate declarations. A type-checker can easily be implemented in Prolog (see [Mycroft, O'Keefe 1983] for details) and type checking is not expensive because it works locally on clauses.

However, there is a serious drawback: relations between types cannot be specified. Types can be defined neither as unions nor as subtypes of other types. For example, the following type declarations and the predicate declaration for a predicate *diff* specify the differentiation function for arithmetic expressions¹:

```
:- type var => x, y, z, ...
:- type integer => 0, -1, 1, -2, 2, ...
:- type expr => x, y, z, ... ,
    0, -1, 1, -2, 2, ... ,
    - expr, expr + expr, expr * expr, ln(expr), sin(expr), ...
:- pred diff(expr, var, expr).
```

Note that these type declarations make use of overloading (some functors may belong to more than one type), which can easily be built into Mycroft and O'Keefe's type system. The fact

```
diff(X, X, 1).
```

¹ In practice, of course, either the number of symbols belonging to a type must be finite or the type must be "built-in" as, for example, the type "integer".

cannot be well-typed, because the variable X occurs with type var and $expr$ which are not unifiable. Alternatively, we can declare

```
:- pred diff(expr,expr,expr).
```

which avoids type conflicts but reflects less precisely the intention of the programmer. It would be more appropriate to provide a possibility to declare subtype relations or union types, and to change the well-typing conditions for clauses. For example we can define subtype relations

```
:- subtype var < expr, integer < expr.
```

or, redefine the type $expr$ as a *union* of other types. If the well-typing conditions allow different types of variables to be subtypes of each other, the fact

```
diff( $X^{expr}$ ,  $X^{var}$ ,  $1^{expr}$ )
```

is well-typed. But then the goal

```
:- diff( $x+y^{expr}$ ,  $Y^{var}$ ,  $1^{expr}$ )
```

is statically well-typed and yields a dynamic type conflict when applying the above fact with the substitution $\{Y \leftarrow x+y\}$. The reason for this conflict is the following: The second argument of the goal is used as an output argument, whereas the second argument of the `diff` predicate is intended for input. In other words, the goal causes a dataflow from the first into the second argument of the fact and, in parallel, a transition from a supertype to a subtype ($expr \longrightarrow var$).

These observations suggest that there is a close connection between data flow and type consistency when subtype relations are involved. In the next section, we extend the type system of Mycroft and O'Keefe to deal with subtypes and define a new notion of well-typedness based on dataflow informations.

4. A Polymorphic Type System with Subtypes

4.1 Modes and Dataflow

As stated in the previous section, in order to deal with subtypes information about the dataflow within a clause is needed. We suppose that this information is given by input/output modes of predicates. We distinguish three modes: input (I), output (O), and inout (IO).

Definition 1 (modes).

Let P be a program over $\mathbf{V} \cup \mathbf{F} \cup \mathbf{P}$. A set of modes for P is a set of functions $M = \{m_p \mid p \in \mathbf{P}\}$ such that $m_{p/n} : \{1, \dots, n\} \longrightarrow \{I, O, IO\}$ for every $p/n \in \mathbf{P}$. P and a goal g are said to be *mode consistent with respect to M* iff for any goal $\leftarrow p(t_1, \dots, t_n)$ which is produced during a refutation of g and for all $i \in \{1, \dots, n\}$ the following conditions hold:

- (1) If $m_p(i) = I$, then for every answer substitution θ it is $\theta t_i = t_i$.
- (2) If $m_p(i) = O$, then t_i is a variable.

■

In other words, input arguments of a goal are not further instantiated by execution of the goal and output arguments are free variables at call time (that is, they are not instantiated by answer substitutions for previously called goals). For inout arguments nothing is known about the degree of instantiation. Some mode systems additionally require input arguments to be ground at call time and output arguments to be ground at return time [Warren 77, Bruynooghe 82]. For a comprehensive discussion of different modes and mode systems see [Dietrich 88].

We suppose that modes can be assigned to predicates via *mode declarations* like

```
:- mode append(i,i,o), member(i,i), unify(io,io).
```

These mode declarations imply modes m_{append} , m_{member} , and m_{unify} such that

$$\begin{aligned} m_{append}(1) &= m_{append}(2) = I, m_{append}(3) = O, \\ m_{member}(1) &= m_{member}(2) = I, \text{ and} \\ m_{unify}(1) &= m_{unify}(2) = IO. \end{aligned}$$

Another way to obtain mode information is to perform global analysis like [Mellish 85].

Given a set of modes for a program, a unique mode can be assigned to every *occurrence* of a clause. This especially aims at assigning modes to variable occurrences in clauses.

Definition 2. (*modes of occurrences of clauses*).

Let P be a program and M a set of modes for P . Let $c := A_0 \leftarrow A_1, \dots, A_n \in P$ be a clause and $i.u \in O(c)$. The *mode of c at $i.u$* , $m(c, i.u) \in \{I, O, IO\}$ is defined as follows:

- (1) $m(A_0 \leftarrow A_1, \dots, A_n, i.u) = I$ iff $A_i = p(t_1, \dots, t_k)$ and $[i=0 \Rightarrow m_p(i)=I]$ and $[i>0 \Rightarrow m_p(i)=O]$.
- (1) $m(A_0 \leftarrow A_1, \dots, A_n, i.u) = O$ iff $A_i = p(t_1, \dots, t_k)$ and $[i=0 \Rightarrow m_p(i)=O]$ and $[i>0 \Rightarrow m_p(i)=I]$.
- (3) $m(A_0 \leftarrow A_1, \dots, A_n, i.u) = IO$ iff $A_i = p(t_1, \dots, t_k)$ and $m_p(i)=IO$

■

Note that in the body of a clause the *mode of an occurrence* is converse to the *mode of the predicate* of that occurrence. If the mode of a body occurrence is I, values are passed from the head of an eventually applied clause into this occurrence. That is, the mode of the predicate must be O. If the mode of a body occurrence is O, values are passed from this occurrence into the head of an eventually applied clause, and therefore the mode of the predicate must be I.

Based on the notion of occurrences of clauses, we can now define a relation describing dataflow within a clause.

Definition 3. (*dataflow relation*).

Let P be a program and M a set of modes for P . The *dataflow relation* of P w.r.t. M $df^M(c) \subseteq O(c) \times O(c)$ is defined by

$$(u, v) \in df^M(c) \text{ iff } c/u = c/v \in \mathbf{V} \text{ and } m(c, u) \in \{I, IO\} \text{ and } m(c, v) \in \{O, IO\}.$$

■

4.2 Types and Typings

We extend the alphabet given in section 2 with the following sets:

- **TV**, a countable, infinite set of *type variables* ($\alpha, \beta, \gamma, \dots \in \mathbf{TV}$, $\mathbf{V} \cap \mathbf{TV} = \emptyset$),
- **TC**, a countable set of symbols with arity, the *type constructors* ($\mathbf{F} \cap \mathbf{TC} = \emptyset$).

Types are terms over **TC** and **TV** (*type terms*). We denote the set of all type terms over **TC** and **TV** by $\Sigma(\rho, \sigma, \tau \in \Sigma)$. Ground type terms are called *monotypes*, others *polytypes*. *Extended types* are types or sequences of types, written $(\sigma_1, \dots, \sigma_n)$ or $(\sigma_1, \dots, \sigma_k \rightarrow \tau)$. We denote the set of extended types over Σ by Σ^* . *Type substitutions* are substitutions on type terms in the same sense as substitutions on ordinary terms.

We interpret types as sets of terms (subsets of the Herbrand universe). The *subtype relation* is a binary relation $\subseteq$ on type terms, which is interpreted as the subset relation on sets of terms. That is, $\sigma \subseteq \tau$ iff and only if the set of terms belonging to σ is a subset of the set of terms belonging to τ .

A *typing* of a set of (predicate, function, and variable) symbols associates extended types to every symbol. A typing of a clause, or atom, or term associates an extended type to every symbol occurring in the clause or atom or term.

Definition 4. (Typings)

Let $S \subseteq \mathbf{V} \cup \mathbf{F} \cup \mathbf{P}$. A *typing of S* is a relation $t_S \subseteq S \times \Sigma^*$ such that

- (i) If $s \in \mathbf{V}$ and $(s, \sigma) \in t_S$ then $\sigma \in \Sigma$
- (ii) If $s/k \in \mathbf{F}$ and $(s, \sigma) \in t_S$ then
 - $\sigma = \sigma_1, \dots, \sigma_k \rightarrow \sigma_0$, $\sigma_i \in \Sigma$, $\sigma_0 = c(\alpha_1, \dots, \alpha_n)$, $c \in \mathbf{TC}$, $\alpha_i \in \mathbf{TV}$ and $\text{var}(\sigma_j) \subseteq \text{var}(\sigma_0)$ for $j > 0$,
 - for all $\tau \in \Sigma^*$ with $(s, \tau) \in t_S$ $\tau = \tau_1, \dots, \tau_k \rightarrow \tau_0$, $\tau_i \simeq \sigma_i$ for $i > 0^1$.
- (iii) If $s/n \in \mathbf{P}$ and $(s, \sigma) \in t_S$ then
 - $\sigma = \sigma_1, \dots, \sigma_n$, $\sigma \in \Sigma$,
 - for all $\tau \in \Sigma^*$ with $(s, \tau) \in t_S$ $\tau = \sigma$.

If $(\tau_1, \dots, \tau_n \rightarrow \tau_0, f) \in t_S$, τ_i , $i \geq 1$ are called *argument types* of f and τ_0 is called *result type* of f . ■

As shown in section 3, we denote typings for clauses by superscribing the terms occurring in a clause with the corresponding types.

Given a typing of the function symbols, the types themselves can be interpreted as sets of terms as follows: If $\sigma = c(\alpha_1, \dots, \alpha_n)$ and $(f, (\sigma_1, \dots, \sigma_k \rightarrow \sigma)) \in t_S$ then $f(t_1, \dots, t_k) \in \phi\sigma$ iff $t_i \in \phi\sigma_i$ for any type substitution ϕ and $i = 1, \dots, k$.

4.3 Well-Typed Programs

The notion of well-typings expresses consistency between typed clauses and separately given typings of the symbols occurring in the clause. Well-typings for clauses can be derived by the following type inference rules which are taken from [Mycroft, O'Keefe 84] and extended to allow variables

¹ If we admit $\tau_i \not\simeq \sigma_i$ this corresponds to overloading.

and function symbols to have more than one typing (i.e. typings are not functions, but relations).

Definition 5 (*type inference rules*).

Let $S \subseteq \mathbf{V} \cup \mathbf{F} \cup \mathbf{P}$ and $t_S \subseteq S \times \Sigma^*$ a typing of S .

- (TI1) $t_S \vdash p(t_1^{f_1}, \dots, t_k^{f_k}) \leftarrow A_1, \dots, A_m$ if $p/k \in \mathbf{P}$ and
 $(p, \rho) \in t_S$ and $(\tau_1, \dots, \tau_k) \simeq \rho$ and and $t_S \vdash t_j^{f_j}$ for $j \in \{1, \dots, k\}$ and
 $t_S \vdash A_i^{f_i}$ for $i \in \{0, \dots, m\}$.
- (TI2) $(t_S, M) \vdash p(t_1^{f_1}, \dots, t_k^{f_k})$ if $p/k \in \mathbf{P}$ and
 $(p, \rho) \in t_S$ and $(\tau_1, \dots, \tau_k) \succeq \rho$ and and $t_S \vdash t_j^{f_j}$ for $j \in \{1, \dots, k\}$.
- (TI3) $t_S \vdash f(t_1^{f_1}, \dots, t_n^{f_n})^\sigma$ if $f \in \mathbf{F}$ and
 $(f, \rho) \in t_S$ and $\tau_1, \dots, \tau_n \rightarrow \sigma \succeq \rho$ and $t_S \vdash t_i^{f_i}$ for $i \in \{1, \dots, n\}$.
- (TI4) $t_S \vdash x^\sigma$ if $x \in \mathbf{V}$ and $(x, \sigma) \in t_S$

■

As shown in section 3, the principle "well-typed programs can't go wrong" is not longer true, if variables can have different types. To re-establish this principle we have to introduce a new notion of well-typedness with respect to the dataflow in clauses given by modes.

Definition 6 (*well-typedness modulo a set of modes*).

Let c be a typed clause, $S \subseteq \mathbf{V} \cup \mathbf{F} \cup \mathbf{P}$ containing every symbol occurring in c , M a set of modes for the predicate symbols occurring in c and t_S a typing for S . t_S is called a *well-typing modulo M* for c ($t_S \vdash_M c$) iff $t_S \vdash c$ and the following *variable typing condition* holds:

- (VTC) For all $u, v \in O(c)$ such that $c/u = x_\sigma$ and $c/v = x_\tau$ and $x \in \mathbf{V}$:
 $(u, v) \in df^M(c) \Rightarrow \sigma \subseteq \tau$ and $(v, u) \in df^M(c) \Rightarrow \tau \subseteq \sigma$.

■

The variable typing condition (VTC) guarantees that in well-typed clauses dataflow cannot happen from a supertype occurrence to a subtype occurrence. In the example shown in section 3

```
:- mode diff(i,o,i).
diff(Xexpr, Xvar, Iexpr).
```

this condition obviously is hurt ($((0.1, 0.2) \in df^M(c))$), which results in a possible runtime type conflict.

The proof for our main theorem that well-typed clauses produce well-typed resolvents (modulo a set of modes) works along the lines shown in [Mycroft, O'Keefe 84]. We only have to pay special attention to multiple occurrences of variables with different types (which is always ill-typed in Mycroft/O'Keefe's system).

In the following, let P be a program, M a set of modes for P and t_S a typing for every function and predicate symbol occurring in P . Lemma 7 states that most general unifiers computed when building resolvents from well-typed clauses instantiate variables according to their derived types up to instantiations of type variables.

Lemma 7. Let $c := B_0 \leftarrow B_1, \dots, B_n \in P$, $g := \leftarrow A_1, \dots, A_m$ a goal, and $\theta \in mgu(B_0, A_1)$. If P and g are mode consistent with respect to M , $t_S \vdash_M c$ and $t_S \vdash_M g$ then there is a type substitution ϕ such that

- (a) If $x \in \text{var}(B_0)$ and $t_S \vdash x^\sigma$ then $t_S \vdash \theta x^{\phi\sigma}$,
- (b) If $x \in \text{var}(A_1)$ and $t_S \vdash x^\tau$ then $t_S \vdash \theta x^\tau$.

Proof. We first show how the type substitution ϕ can be determined: Let $B_0 = p(s_1^{\sigma_1}, \dots, s_k^{\sigma_k})$, $A_1 = p(t_1^{\tau_1}, \dots, t_k^{\tau_k})$, and $(p, \rho) \in t_S$. Because c and g are well-typed modulo M we can infer from (TI1) and (TI2) that

- (1) $\rho \preceq (\sigma_1, \dots, \sigma_k)$ and $t_S \vdash s_i^{\sigma_i}$ for $i=1, \dots, k$ and
- (2) $\rho \preceq (\tau_1, \dots, \tau_k)$ and $t_S \vdash t_i^{\tau_i}$ for $i=1, \dots, k$.

Defining $\phi := \text{mgu}((\sigma_1, \dots, \sigma_k), (\tau_1, \dots, \tau_k))$ we have

- (3) $\phi(\sigma_1, \dots, \sigma_k) = (\tau_1, \dots, \tau_k)$.

If there is no variable occurring twice in B_0 or A_1 with different types, (a) and (b) follow directly from (1), (2) and (3). Otherwise (a) and (b) are guaranteed by (VTC). Suppose, for example, that there are $x \in \mathbf{V}$, $iu \in O(B_0)$ and $ju \in O(B_0)$ such that $m_p(i) = I$, $m_p(j) = O$, $B_0 iu = x^\sigma$ and $B_0 ju = x^\tau$. Then $(iu, ju) \in \text{df}^M(c)$ and $\sigma \subseteq \tau$ (VTC). Because $m_p(i) = I$ it is $\theta A_1 i = A_1 i$ and therefore $\theta x = A_1 iu$. Then from (2) and (3) follows that $t_S \vdash \theta B_0 iu^{\phi\sigma}$ and because $\sigma \subseteq \tau$ also $t_S \vdash \theta B_0 ju^{\phi\tau}$. All other cases can be proven by similar considerations. ■

Theorem 8. Let $c \in P$, g a goal such that P and g are mode consistent with respect to M . If $t_S \vdash_M c$ and $t_S \vdash_M g$ then $t_S \vdash_M g'$ for any SLD-resolvent g' of g .

Proof. Let $c = B_0 \leftarrow B_1, \dots, B_n$, $g = \leftarrow A_1, \dots, A_m$ and $\theta \in \text{mgu}(B_0, A_1)$. Then $g' = \leftarrow \theta B_1, \dots, \theta B_n, \theta A_2, \dots, \theta A_m$. We show that

- (*) $t_S \vdash \theta B_i^\phi$ for $i=1, \dots, n$ and
- (**) $t_S \vdash \theta A_i$ for $i=2, \dots, m$

where ϕ is the type substitution determined by Lemma 7 and B_i^ϕ denotes the typed literal obtained from B_i by applying ϕ to all its associated types. Let $B_i = p(s_1^{\sigma_1}, \dots, s_k^{\sigma_k})$ and $(p, \rho) \in t_S$. Because $t_S \vdash_M c$ it is $t_S \vdash B_i$ and (TI2) implies that

- (1) $\rho \preceq (\sigma_1, \dots, \sigma_k) \preceq \phi(\sigma_1, \dots, \sigma_k)$ and
- (2) $t_S \vdash s_j^{\sigma_j}$ for $j=1, \dots, k$

and we can show

- (3) $t_S \vdash \theta s_j^{\phi\sigma_j}$ for $j=1, \dots, k$.

by induction on the structure of s_j :

- If s_j is a constant, then there is $\rho' \in \Sigma$ such that $(s_j, \rho') \in t_S$ and $\rho' \preceq \sigma_j \preceq \phi\sigma_j$ and (TI3) implies (3).
- If s_j is a variable not occurring in B_0 then $\theta s_j = s_j$ and (3) follows from (2). If $s_j \in \text{var}(B_0)$ then (3) is guaranteed by (VTC): If, for example, there is a dataflow from an occurrence s_j' in B_0 to $s_j^{\sigma_j}$ in B_i then $\tau \subseteq \sigma_j$ and $t_S \vdash \theta s_j^{\phi\tau}$ (Lemma 7 (a)) which implies (3).
- If $s_j = f(t_1^{\tau_1}, \dots, t_l^{\tau_l})$ then there is $\rho' \in \Sigma$ such that $(f, \rho') \in t_S$ and $\rho' \preceq (\tau_1, \dots, \tau_l \rightarrow \tau) \preceq \phi(\tau_1, \dots, \tau_l \rightarrow \tau)$ and $t_S \vdash t_r^{\tau_r}$ for $r=1, \dots, l$. By induction hypothesis it follows that $t_S \vdash \theta t_r^{\phi\tau_r}$ and (TI3) implies (3).

Now (1) and (3) imply (*) and (**) can be proven in a similar way from $t_S \vdash A_i$, $i=1, \dots, m$,

and Lemma 7 (b). It remains to show that (VTC) holds for g' . This can be deduced from the fact that (VTC) holds for c and g and that all variables are instantiated according to their types (Lemma 7). (Note that if variables of some θB_i , $i=1, \dots, n$ also occur in one of θA_j , $j=2, \dots, m$ there must have been a dataflow between A_1 and A_j and between B_0 and B_i).

Corollary 9. If P and a goal g are mode consistent with respect to M and $t_S \vdash_M c$ for all $c \in P \cup \{g\}$ then for every variable $x' \in \text{var}(g)$ and every answer substitution θ for g with respect to P $t_S \vdash \theta x'$.

Proof. Induction on the length of an SLD-refutation of g . The base case is given by Lemma 7 and Theorem 8.

5. Type Checking

In practice, a Prolog programmer only defines a typing for the predicate and function symbols (or only some of them) occurring in his program. Then a *type-checker* can automatically derive typings for the variables and undeclared functors of the program such that the given typing is a well-typing for any clause. If this is possible, the program is said to be well-typed. In this section we give an outline of this type checking and type derivation process and discuss some special problems which only occur in the presence of subtypes.

The type-checker processes the literals of a clause "top down" in the following way: From the (given) typing of a predicate symbol or a function symbol, type requirements for the arguments can be inferred by (TI1), (TI2), or (TI3), respectively: If $p(t_1, \dots, t_n)$ is the processed literal and $((\tau_1, \dots, \tau_n), p) \in t_S$ the type of the arguments t_i must be an instance of τ_i (for the head literal, they must be identical up to renaming of variables). This leads to type requirements for the arguments which must be checked to be kept. If a term $t = f(s_1, \dots, s_n)$ has a type requirement τ , there must be types $\tau_1, \dots, \tau_n$ such that $((\tau_1, \dots, \tau_n \rightarrow \tau), f) \in t_S$, and then s_i must be checked against τ_i in the same way, until constants or variables are reached. The type of a constant must be of the required type. If an untyped variable is reached, the type of this variable occurrence becomes the required type.

If there is a (program) variable which has been assigned different types at different occurrences, the variable typing condition (VTC) must be verified. (VTC) can be enforced by instantiating type variables, if they are not part of the types in the clause head.¹ If this is not possible, the program is not well-typed.

The verification of (VTC) may be quite complicated. We demonstrate the steps which must be performed by some examples. Consider the following clause with type and mode declarations:

```
:- pred p( $\alpha$ , list( $\alpha$ )), q(nat, nat), r( $\beta$ , list( $\beta$ )).
:- mode p(i, o), q(i, o), r(i, o).
p(X, Y) :- q(X, Z), r(Z, Y).
```

¹ In Mycroft/O'Keefe's type system this is performed by simply *unifying* the different type assignments.

The type checking process as outlined above leads to the following type assignments for variables (every column contains the assignments derived when processing one literal):

$X: \alpha$	$X: \text{nat}$	$Z: \beta$
$Y: \text{list}(\alpha)$	$Z: \text{nat}$	$Y: \text{list}(\beta)$

To ensure (VTC), these types must meet the following subtype relations:

- (1) $\alpha \subseteq \text{nat}$,
- (2) $\text{list}(\alpha) \subseteq \text{list}(\beta)$,
- (3) $\text{nat} \subseteq \beta$.

The next problem is how to decide for two given types σ and τ , whether $\sigma \subseteq \tau$. Consider the following type declarations:

```

:- type nat => 0,1,2,3,...
:- type int => 0,1,-1,2,-2,...
:- type list( $\gamma$ ) => nil, .( $\gamma$ ,list( $\gamma$ )).
:- type tree( $\gamma$ ) => empty, t(t( $\gamma$ ), $\gamma$ ,t( $\gamma$ )).
:- type tl( $\alpha$ , $\beta$ ) = list( $\alpha$ ) ; tree( $\beta$ ).
```

Obviously, we have that $\text{nat} \subseteq \text{int}$, because every term of type nat (which are only constants in this definition) is also of type int . Although the type $\text{tl}(\alpha, \beta)$ is the union of $\text{list}(\alpha)$ and $\text{tree}(\beta)$, we cannot generally say that $\text{list}(\gamma)$ or $\text{tree}(\gamma)$ are subtypes of $\text{tl}(\alpha, \beta)$. It depends on the (type) values which the type variables α , β and γ can take. But they are subtypes, if γ is a subtype of α or β , respectively. As a special case, $\text{list}(\alpha)$ and $\text{tree}(\beta)$ are subtypes of $\text{tl}(\alpha, \beta)$. Due to these properties of polymorphic types, we define the notion of a *conditional subtype relation*:

$$[\sigma \subseteq \tau \mid \alpha_1 \subseteq \beta_1, \dots, \alpha_n \subseteq \beta_n]$$

where $\{\alpha_1, \dots, \alpha_n\} \subseteq \text{var}(\sigma)$ and $\{\beta_1, \dots, \beta_n\} \subseteq \text{var}(\tau)$. The ' $\mid$ ' in the expression can be interpreted as "if". From the above declarations we can infer the following conditional subtype relations:

```

nat  $\subseteq$  int.
list( $\alpha$ )  $\subseteq$  list( $\beta$ )  $\mid \alpha \subseteq \beta$ .
tree( $\alpha$ )  $\subseteq$  tree( $\beta$ )  $\mid \alpha \subseteq \beta$ .
tl( $\alpha_1, \beta_1$ )  $\subseteq$  tl( $\alpha_2, \beta_2$ )  $\mid \alpha_1 \subseteq \alpha_2, \beta_1 \subseteq \beta_2$ .
list( $\alpha$ )  $\subseteq$  tl( $\beta, \gamma$ )  $\mid \alpha \subseteq \beta$ .
tree( $\alpha$ )  $\subseteq$  tl( $\beta, \gamma$ )  $\mid \alpha \subseteq \gamma$ .
```

The following definition determines how all conditional subtype relations of a typing can be computed.

Definition 10 (*conditional subtype relation*).

Let P be a program and t_S a typing of all symbols occurring in P . For $c/n \in TC$ let $F(c) := \{f/k \in F \mid \exists \tau_1, \dots, \tau_k \in \Sigma, \exists \alpha_1, \dots, \alpha_n \in TV: ((\tau_1, \dots, \tau_k \rightarrow c(\alpha_1, \dots, \alpha_n)), f) \in t_S\}$ the set of all function symbols with result type $c(\alpha_1, \dots, \alpha_n)$. The *set of conditional subtype relations* $CSR(t_S)$ is defined by

$$[c_1(\alpha_1, \dots, \alpha_n) \subseteq c_2(\beta_1, \dots, \beta_m) \mid \alpha'_1 \subseteq \beta'_1, \dots, \alpha'_p \subseteq \beta'_p] \in CSR(t_S) \text{ iff} \\ c_1/n, c_2/m \in TC \text{ and } F(c_1) \subseteq F(c_2) \text{ and}$$

$\{\alpha'_1 \subseteq \beta'_1, \dots, \alpha'_p \subseteq \beta'_p\}$ is the minimal (possibly empty) set such that
 $(\tau_1, \dots, \tau_k \rightarrow c_1(\alpha_1, \dots, \alpha_n), f) \in t_S$ and $(\sigma_1, \dots, \sigma_k \rightarrow c_2(\beta_1, \dots, \beta_m), f) \in t_S$
 and $\forall i \in \{1, \dots, k\}: \phi\tau_i = \sigma_i$ and $\phi = \{\gamma_1 \leftarrow \gamma'_1, \dots, \gamma_q \leftarrow \gamma'_q\}$
 $\Rightarrow \{\gamma_1 \subseteq \gamma'_1, \dots, \gamma_q \subseteq \gamma'_q\} \subseteq \{\alpha'_1 \subseteq \beta'_1, \dots, \alpha'_p \subseteq \beta'_p\}.$

■

Note that in the above definition it is

$$\{\gamma_1, \dots, \gamma_q\} \subseteq \{\alpha'_1, \dots, \alpha'_q\} \subseteq \{\alpha_1, \dots, \alpha_n\} \text{ and } \{\gamma'_1, \dots, \gamma'_q\} \subseteq \{\beta'_1, \dots, \beta'_q\} \subseteq \{\beta_1, \dots, \beta_m\}.$$

The type substitution ϕ is the renaming substitution required for argument types of function symbols with more than one typing (cf. Definition 4 (ii)).

Deciding whether or not $\sigma \subseteq \tau$ for two type terms consists in "applying" the conditional subtype relations and evaluating the conditions. For example we have

$$\text{list}(\text{nat}) \subseteq \text{list}(\text{int}), \text{list}(\text{int}) \not\subseteq \text{list}(\text{nat}).$$

because

$$\text{nat} \subseteq \text{int}.$$

Checking whether

$$\text{tl}(\alpha, \text{nat}) \subseteq \text{tl}(\text{int}, \alpha)$$

leads to subtype relations

$$(4) \text{ nat} \subseteq \alpha, \alpha \subseteq \text{int}$$

which cannot be further evaluated, but can be shown to be satisfiable because of the transitivity of the subtype relation. That is, *nat* and *int* can be seen as some kind of lower and upper bound for the (type) value of α , and we have $\text{tl}(\alpha, \text{int}) \subseteq \text{tl}(\text{nat}, \alpha)$ whenever $\text{nat} \subseteq \alpha \subseteq \text{int}$. Evaluating

$$\text{tl}(\alpha, \text{nat}) \subseteq \text{tl}(\text{nat}, \alpha)$$

even yields that

$$(5) \alpha = \text{nat}$$

which means that *nat* is a lower and upper bound for α .

That is, in general one cannot absolutely determine whether a subtype relation between two polytypes hold. It can only be said that the relation holds if the values of type variables range between some lower and upper bounds. These lower and upper bounds can be computed by means of the conditional subtype relations $CSR(t_S)$, as described by the following algorithm:

compute_buonds($\sigma, \tau, CSR, LB, UB$):

Input: σ, τ : type terms; CSR : a set of conditional subtype relations.

Output: LB, UB : The sets of lower and upper bounds for type variables occurring in σ and τ with respect to CSR . (If $(\alpha, \tau) \in LB$ then τ is a lower bound for α , if $(\alpha, \tau) \in UB$ then τ is an upper bound for α .) If the algorithm stops with failure then $\sigma \not\subseteq \tau$.

$LB := \emptyset; UB := \emptyset; \text{compute_bounds1}(\sigma, \tau);$

if compute_bounds1 has stopped with failure then stop with failure else stop with success.

$\text{compute_bounds1}(\sigma, \tau, CSR):$

Input: σ, τ : type terms; CSR : a set of conditional subtype relations.

Global Variables: LB, UB .

1. If $\sigma \in \mathbf{TC}$ and $\tau \in \mathbf{TC}$
then [if $[\sigma \subseteq \tau] \in CSR$ then stop with success else stop with failure];
2. If $\tau \in \mathbf{TV}$ then $LB := LB \cup \{(\tau, \sigma)\}$;
3. If $\sigma \in \mathbf{TV}$ then $UB := UB \cup \{(\sigma, \tau)\}$;
4. If there are $c_1, c_2 \in \mathbf{TC}$, $\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m \in \mathbf{TV}$ and a type substitution ϕ such that
 $\sigma = \phi c_1(\alpha_1, \dots, \alpha_n)$ and $\tau = \phi c_2(\beta_1, \dots, \beta_m)$ and
 $[c_1(\alpha_1, \dots, \alpha_n) \subseteq c_2(\beta_1, \dots, \beta_m) \mid \alpha'_1 \subseteq \beta'_1, \dots, \alpha'_p \subseteq \beta'_p] \in CSR$
then for $i = 1, \dots, p$ execute $\text{compute_bounds1}(\phi \alpha'_i, \phi \beta'_i, CSR)$;
5. If one of the compute_bounds1 executions in step 4. has stopped with failure then stop with failure else stop with success.

Coming back to our type checking example above which resulted in subtype relations (1) - (3), we can evaluate the relation (2) to

$$(6) \alpha \subseteq \beta$$

and from (1), (3) and (6) we obtain the following lower and upper bounds for α and β ($\alpha.lb, \alpha.ub, \beta.lb$ and $\beta.ub$):

$$\begin{array}{ll} \beta.ub = \{ \beta \} & \beta.lb = \{ \alpha, \text{nat} \} \\ \alpha.ub = \{ \text{nat}, \beta \} & \alpha.lb = \{ \alpha \} \end{array}$$

These upper and lower bounds among others restrict the kind of terms the predicate p can be called with without causing dynamic type conflicts (α must be a subtype of nat !). This information can be useful for the programmer (either he has specified the predicate too generally or there is a programming error).

Summarizing the type checking process, we can determine four main tasks:

- (1) derive type requirements for program variables by means of (T11), ..., (T14),
- (2) from these derive subtype relations which must be kept by means of (VTC),
- (3) determine lower and upper bounds for type variables by means of the conditional subtype relations (algorithm compute_bounds),
- (4) determine whether the upper and lower bounds for every variable are consistent. That is, for any type variable every lower bound must be a subtype of every upper bound.

6. Conclusions

Trying to improve some software engineering qualities of logic programming, we have proposed a system of polymorphic types for Prolog. It extends the type system of [Mycroft, O'Keefe 84] to deal with subtypes. Static type checking is possible, if information about the dataflow within clauses is available. Among others, this information can be provided by a mode system. We have outlined the type checking process and discussed several problems which make type checking more sophisticated when subtype relations are involved. In general, it is not possible to decide whether or not a type is a subtype of another type if the types are polytypes. It depends on the values that type variables can be instantiated with. This led to the notion of conditional subtype relations.

Unification of type terms, which is a central operation in Mycroft/O'Keefe's type checking algorithm, must be replaced by showing a set of subtype relations among type terms to be satisfiable. It is a subject of further research to develop efficient algorithms for this problem and to characterize the typings for which it is decidable. This includes the question whether some of the restrictions imposed on typings (Definition 4) can be dropped without weakening our results.

Type and mode declarations do not have any influence on the execution of programs, therefore they can be seen as optional. A type checker can be part of a Prolog programming environment and implemented by preprocessing. The program itself can be interpreted or compiled without modifications as usual.

A Prolog implementation of a type checker for our type system is described in [Hagl 87].

Acknowledgements. We thank our colleagues from the KAP group (KARlsruhe Prolog project) A. Bockmayr, P. Kursawe and I. Varsek, for many fruitful discussions and their critical comments on earlier versions of this paper, and H. Lock for providing me with useful references about type systems in the functional programming area.

References

- M. Bruynooghe, *Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs*, 1st International Conference on Logic Programming, Marseille, 1982.
- L. Cardelli, P. Wegner, *On understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol. 17, No. 4, December 1985.
- R. Dietrich, *Modes and Types for Prolog*, Arbeitspapiere der GMD Nr. 285, January 1988.
- W. Drabent, *Do Logic Programs Resemble Programs in Conventional Languages?*, 4th IEEE Symposium on Logic Programming, San Francisco, 1987, 389-397.
- J. A. Goguen, J. Meseguer, *Equality, Types, Modules, and Generics for Logic Programming*, Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, 1984, 115-126.
- F. Hagl, *Statische Analyse von Prolog Programmen mit Datenfluß- und Typangaben*, Diplomarbeit, Universität Karlsruhe/GMD Karlsruhe, July 1987
- M. Huber, I. Varsek, *Extended Prolog for Order-Sorted Resolution*, 4th IEEE Symposium on Logic Programming, San Francisco, 1987, 34-45

- J. W. Lloyd, *Foundations of Logic Programming*, Springer, 1984 (2nd Edition 1987).
- C. S. Mellish, *Some Global Optimizations for a Prolog Compiler*, J. Logic Programming 1985, No. 1, 43-66.
- R. Milner, *A Theory of Type Polymorphism in Programming*, J. Computer System Science 17(3), 1978, 348-375.
- P. Mishra, *Towards a Theory of Types in Prolog*, Proc. 1st IEEE International Symposium on Logic Programming, 1984, 289-298.
- A. Mycroft, R. A. O'Keefe, *A Polymorphic Type System for Prolog*, DAI Research Paper 211, Dept. of Artificial Intelligence, Edinburgh University, 1983.
- A. Mycroft, R. A. O'Keefe, *A Polymorphic Type System for Prolog*, Artificial Intelligence 23, 1984, 295-307.
- M. Schmidt-Schauß, *Unification in a Many-Sorted Calculus with Declarations*, Proc. 8th Conference on Automated Deduction, Oxford, UK., July 1986, LNCS 230, 538-552.
- G. Smolka, *Making Control and Data Flow in Logic Programs Explicit*, Proc. 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas, August 1984.
- G. Smolka, W. Nutt, *Order-Sorted Equational Computation*, Proc. CREAS, Austin, Texas, May 1987.
- C. Walther, *A Many-Sorted Calculus Based on Resolution and Paramodulation*, Proc. 8th International Joint Conference on Artificial Intelligence, Karlsruhe, W. Germany, 1983, 882-891.
- D. H. D. Warren, *Applied Logic - It's Use and Implementation as a Programming Tool*, PhD Thesis, University of Edinburgh, 1977. Available as: Technical Note 290, June 1983, SRI International.
- D. H. D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, Artificial Intelligence Center, SRI International, 1983.
- J. Zobel, *Derivation of Polymorphic Types for Prolog Programs*, Proc. 4th International Conference on Logic Programming, Melbourne, Australia, May 1987, 817-838.