Type Declarations as Subtype Constraints in Logic Programming

Dean Jacobs
University of Southern California
Los Angeles, CA 90089-0782

Abstract

This paper presents a type system for logic programs that supports parametric polymorphism and subtypes. This system follows most knowledge representation and object-oriented schemes in that subtyping is *name-based*, i.e., τ_1 is considered to be a subtype of τ_2 iff it is declared as such. We take this as a fundamental principle in the sense that type declarations have the form of subtype constraints. Types are assigned meaning by viewing such constraints as Horn clauses that, together with a few basic axioms, define a subtype predicate. This technique provides a (least) model for types and, at the same time, a sound and complete proof system for deriving subtypes. Using this proof system, we define well-typedness conditions which ensure that a logic program/query respects a set of predicate types. We prove that these conditions are consistent in the sense that every atom of every resolvent produced during the execution of a well-typed program is consistent with its type.

1 Introduction

Type systems for logic programming languages may be generally classified as being either descriptive or prescriptive. In a descriptive system [Mis84, MR85, Red88, Zob87], types are automatically inferred by the compiler. The goal here is to derive safe upper bounds on the success set of predicates for the purposes of program optimization. In a prescriptive system [MO84, YS87, DH88], on the other hand, types are explicitly declared by the programmer. The goal here, as in most conventional type systems, is to restrict the allowable usage of predicates for the purposes of security, documentation, and optimization. In all of the references above, types appear only at compile-time and standard Prolog-like computation mechanisms may be used. In contrast, there are also prescriptive systems where the underlying computation mechanisms are extended to support some form of typed unification [GM86, AKN86, Smo88].

This paper presents a prescriptive type system for logic programs, along the lines of [MO84], that supports parametric polymorphism and subtypes. This system follows most knowledge representation and object-oriented schemes in that subtyping is *name-based*, i.e., τ_1 is considered to be a subtype of τ_2 iff it is declared as such. We take this as a fundamental principle in the sense that type declarations have the form of subtype constraints. As an example, the declarations

```
FUNC 0, succ, pred.
TYPE nat, unnat, int.
  nat >= 0 + succ(nat).
  unnat >= 0 + pred(unnat).
  int >= nat + unnat.
```

introduce a type `int` with subtypes `nat` and `unnat` and elements `0`, `succ(0)`, `pred(0)`, `succ(succ(0))`, etc. Here, the function symbols `0`, `succ`, and `pred` have fixed interpretations as type constructors and the polymorphic type constructor `+` is predefined as follows.

```
TYPE +.
  A+B >= A.
  A+B >= B.
```

As a further example, the declarations

```
FUNC nil, cons.
TYPE elist, nelist, list.
  elist >= nil.
  nelist(A) >= cons(A,list(A)).
  list(A) >= elist + nelist(A).
```

introduce a polymorphic type `list(A)` with subtypes `elist` and `nelist(A)`. Types are assigned meaning by viewing such declarations as Horn clauses that, together with a few basic axioms, define a subtype predicate `>=`. This technique provides a (least) model for

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

types and, at the same time, a sound and complete proof system for deriving subtypes.

In our type system, predicate types are used to restrict the allowable usage of predicates. For example, the declaration

```
PRED app(list(A),list(A),list(A)).
  app(nil,L,L).
  app(cons(X,L),M,cons(X,N)) :- app(L,M,N).
```

restricts the usage of the append predicate app to lists; note that this rules out certain successful queries, such as `:- app(nil,0,0)`. We define well-typedness conditions which ensure that a program respects a set of predicate types; programs that are not well-typed are expected to be rejected by the type checker. We prove that these conditions are consistent in the sense that every atom of every resolvent produced during the execution of a well-typed program is consistent with its type.

Our type system is considerably more expressive than previous proposals of this nature, in particular, it supports the notion of non-uniform polymorphic types. As an example, given types `males` and `females`, the declaration

```
FUNC m, f.
TYPE id.
  id(males) >= m(nat).
  id(females) >= f(nat).
```

introduces a non-uniform polymorphic type `id`. This type models the idea that a variety of different things have id numbers, e.g., we might work with the type `id(vehicles)` of id numbers for `vehicles` in some particular circumstance. Continuing with this example, given the declaration

```
TYPE person.
  person >= male + female.
```

the type `id(person)` contains the elements of `id(males)` and `id(females)`. This paper assigns meaning to all types, however, for simplicity, our well-typedness conditions are defined only for uniform polymorphic types.

This paper is organized as follows. Section 2 introduces types and type declarations. Section 3 develops a deterministic strategy for deriving subtypes given certain syntactic restrictions on type declarations, notably, that they are uniform polymorphic. Section 4 uses this strategy to define a function *match* that forms the basis of our well-typedness conditions. *match* returns a most general typing for the variables of a given term under a given type, if such a typing exists. The correctness of *match* is proven. Section 5 introduces predicate types and informally discusses type checking. Section 6 presents the conditions under which a program/query is considered to be well-typed and proves their consistency.

2 Types and Type Declarations

Let the following disjoint sets of symbols be given.

- V of variables
- F of function symbols with given arity
- T of type constructor symbols with given arity

A term over a set of symbols S is either a variable or a symbol $s/n \in S$ applied to n terms over S. Here, and at several other points in this paper, we abuse the notation slightly by treating 0-ary symbols as if they were arbitrary n-ary symbols. The Herbrand Universe $\mathcal{H}$ consists of the set of all ground terms over F.

Definition 1 (Syntax of Types) *A type $\tau \in Type$ is a term over $F \cup T$.*

Intuitively, a type represents some subset of $\mathcal{H}$ over which computation takes place. Function symbols $f/n \in F$ have a fixed interpretation as type constructors: the type $f(\tau_1, \ldots, \tau_n)$ represents the set of all ground terms $f(t_1, \ldots, t_n)$ where t_i has type τ_i. Type symbols are defined by subtype constraints of the following form.

Definition 2 (Syntax of Subtype Constraints) *A subtype constraint for $c/n \in T$ has the form*

$$c(\tau_1, \ldots, \tau_n) >= \tau$$

where $\tau_1, \ldots, \tau_n$ and τ are types such that

$$\mathrm{var}(\tau) \subseteq \mathrm{var}(c(\tau_1, \ldots, \tau_n))$$

We define the semantics of types under a given set C of subtype constraints in terms of a set H_C of Horn clauses for an infix predicate >=. The set H_C contains each constraint in C as a fact, a substitution axiom

$$s(\alpha_1, \ldots, \alpha_n) >= s(\beta_1, \ldots, \beta_n) :- \alpha_1 >= \beta_1, \ldots, \alpha_n >= \beta_n.$$

for each $s/n \in F \cup T$, including the degenerate case

$$s >= s.$$

where $n = 0$, and the transitivity axiom

```
A >= C :- A >= B, B >= C.
```

We define the notion of subtype in terms of the standard notion of SLD-resolution, e.g., see [Apt88]. Note that this does not imply that we intend to directly execute type declarations in a Prolog-like manner. Rather, it simply gives us a convenient model for types together with a sound and complete proof system for deriving subtypes.

Definition 3 (Subtypes) *Type τ_1 has subtype τ_2 under a set C of subtype constraints, denoted $\tau_1 \succeq_C \tau_2$, iff there exists an SLD-refutation of $H_C \cup \{ :- \tau_1 >= \tau_2 \}$.*

The semantics of types is given by the following function $\mathcal{M}_C : Type \rightarrow \mathcal{P}(\mathcal{H})$.

Definition 4 (Semantics of Types)
$\mathcal{M}_C[\![\tau]\!] = \{t \in \mathcal{H} | \tau \succeq_C t\}$

As an example, given that C consists of the declarations appearing in the introduction, the following SLD-refutation shows that `cons(foo,nil)` $\in \mathcal{M}_C[\![$`list(A)`$]\!]$.

```
:- list(A) >= cons(foo,nil).
:- list(A) >= B1, B1 >= cons(foo,nil).
:- elist + nelist(A) >= cons(foo,nil).
:- elist + nelist(A) >= B3,
                        B3 >= cons(foo,nil).
:- nelist(A) >= cons(foo,nil).
:- nelist(A) >= B3, B3 >= cons(foo,nil).
:- cons(A,list(A)) >= cons(foo,nil).
:- A >= foo, list(A) >= nil.
:- list(foo) >= nil.
:- list(foo) >= B4, B4 >= nil.
:- elist + nelist(foo) >= nil.
:- elist + nelist(foo) >= B5, B5 >= nil.
:- elist >= nil.
:- .
```

$\tau_1 \succeq_C \tau_2$ is an existentially quantified statement about subtypes: by the soundness and completeness of SLD-resolution, $\tau_1 \succeq_C \tau_2$ iff there exists a substitution θ such that $(\tau_1 >= \tau_2)\theta$ is a semantic consequence of H_C. We are also interested in universally quantified statements about subtypes. In particular, we view τ_1 as being a more general type than τ_2 iff there exists a θ such that $\tau_1\theta >= \tau_2$ is a semantic consequence of H_C; recall that free variables in a logical sentence are implicitly universally quantified. For example, `list(A)` is more general than `nelist(int)` but `list(int)` is not more general than `nelist(A)`. Such formulas can be derived as follows. Let $\overline{\tau}$ be τ with each variable replaced by a unique constant not appearing in any type.

Definition 5 (More General Type) *τ_1 is more general than τ_2 iff $\tau_1 \succeq_C \overline{\tau_2}$.*

The correctness of this definition follows from Clark's generalization of the completeness of SLD-resolution [Cla79]. This result states that for every correct answer substitution there is a computed answer substitution that is more general. Thus, there exists a θ such that $\tau_1\theta >= \tau_2$ is a semantic consequence of H_C iff there exists an SLD-refutation of $H_C \cup \{ :- \tau_1 >= \tau_2\}$ in which no variables of τ_2 are instantiated.

3 Deriving Subtypes

In the previous section, we presented a proof system for deriving subtypes. In this section, we develop a deterministic strategy for applying the rules in this system to carry out such derivations. In particular, we show how to select the appropriate clause from H_C to apply at each step in an SLD-refutation of $H_C \cup \{ :- \tau_1 >= \tau_2\}$. To simplify this task, we introduce two syntactic restrictions on type declarations, one that ensures uniform polymorphism and one that ensures that recursive type definitions are "guarded". The results of this section are collected together as an algorithm in the next section, where we define a function *match* that forms the basis of our well-typedness conditions.

Our strategy will be to select clauses from H_C on the basis of the outermost symbol of the supertype. As a first step, we handle the case where this symbol is a function symbol. In the proofs below, we omit discussions of 0-ary symbols where they are simply a degenerate form of the more general n-ary case. In the following, τ_i and σ_i range over types.

Theorem 1 (Refutation Strategy for $f \in F$) *Let C be a set of subtype constraints.*

1. *There is no SLD-refutation of*

$$H_C \cup \{ :- f(\tau_1, \ldots, \tau_n) >= s(\sigma_1, \ldots, \sigma_m)\}$$

where $s \in T \cup F \setminus \{f\}$.

2. *There is an SLD-refutation of*

$$H_C \cup \{ :- f(\tau_1, \ldots, \tau_n) >= f(\sigma_1, \ldots, \sigma_n)\}$$

iff there is one starting with an application of the substitution axiom for f.

Proof:

1. By induction over the length of the derivation. For the base case, since no fact can be immediately applied, there is no refutation of length one. For the inductive step, transitivity is the only axiom that can be applied initially, leading to the resolvent

$$:- f(\tau_1, \ldots, \tau_n) >= \alpha, \alpha >= s(\sigma_1, \ldots, \sigma_m).$$

Without loss of generality, assume the leftmost atom here is selected. Another application of transitivity at this point will not make significant progress. The only choice left is an application of substitution for f, leading to the resolvent

$$:- \tau_1 >= \beta_1, \ldots, \tau_n >= \beta_n, \quad f(\beta_1, \ldots, \beta_n) >= s(\sigma_1, \ldots, \sigma_m).$$

By the inductive hypothesis, there is no SLD-refutation of the last atom in this clause.

2. By induction over the length of the derivation. In the base case, $n = 0$ and the substitution axiom for f is a fact that can be directly applied. For

the inductive step, it suffices to show that for any refutation starting with an application of transitivity there is one starting with an application of substitution for f. Following the reasoning as in 1 above, a refutation starting with transitivity will eventually lead to the resolvent

$$:- \tau_1 >= \beta_1, \ldots, \tau_n >= \beta_n, \quad f(\beta_1, \ldots, \beta_n) >= f(\sigma_1, \ldots, \sigma_n).$$

Assume the rightmost atom here is selected. By the inductive hypothesis, we may assume that substitution for f is applied, leading to the resolvent

$$:- \tau_1 >= \beta_1, \ldots, \tau_n >= \beta_n, \beta_1 >= \sigma_1, \ldots, \beta_n >= \sigma_n.$$

This same resolvent can also be obtained from the original clause by starting with an application of substitution for f, to produce the resolvent

$$:- \tau_1 >= \sigma_1, \ldots, \tau_n >= \sigma_n.$$

followed by an application of transitivity for each individual atom.

□

We now develop a similar strategy for the case where the outermost symbol of the supertype is a type constructor. To accomplish this, we introduce our first restriction on type declarations. This restriction ensures that polymorphism is used in a uniform manner.

Definition 6 (Uniform Polymorphic)
A uniform polymorphic subtype constraint has the form $c(\alpha_1, \ldots, \alpha_n) >= \tau$ *where each* α_i *is a distinct variable. A set* C *of subtype constraints is uniform polymorphic iff each of its members is uniform polymorphic.*

The following definition introduces a notion of the "two-step application" of a uniform polymorphic subtype constraint.

Definition 7 (Two-Step Application)
Two-step application of constraint $c(\alpha_1, \ldots, \alpha_n) >= \tau$ *to resolvent*

$$:- c(\tau_1, \ldots, \tau_n) >= s(\sigma_1, \ldots, \sigma_m).$$

consists of an application of transitivity to produce

$$:- c(\tau_1, \ldots, \tau_n) >= \alpha, \alpha >= s(\sigma_1, \ldots, \sigma_m).$$

followed by an application of the constraint to produce

$$:- \tau\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\} >= s(\sigma_1, \ldots, \sigma_m).$$

Theorem 2 (Refutation Strategy for c ∈ T) *Let* C *be a uniform polymorphic set of subtype constraints.*

1. *There is an SLD-refutation of*

$$H_C \cup \{ :- c(\tau_1, \ldots, \tau_n) >= s(\sigma_1, \ldots, \sigma_m)\}$$

where $s \in F \cup T \setminus \{c\}$ *iff there is one starting with the two-step application of a constraint* $c(\alpha_1, \ldots, \alpha_n) >= \tau \in C$.

2. *There is an SLD-refutation of*

$$H_C \cup \{ :- c(\tau_1, \ldots, \tau_n) >= c(\sigma_1, \ldots, \sigma_n)\}$$

iff there is one starting with either an application of the substitution axiom for c *or the two-step application of a constraint* $c(\alpha_1, \ldots, \alpha_n) >= \tau \in C$.

Proof:

1. By induction over the length of the derivation. In the base case, a constraint $c(\alpha_1, \ldots, \alpha_n) >= \tau \in C$ can be directly applied. Note that this implies that $\tau\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ and $s(\sigma_1, \ldots, \sigma_m)$ can be unified. Two-step application of the constraint will produce

$$:- \tau\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\} >= s(\sigma_1, \ldots, \sigma_m).$$

It can be shown that if t_1 and t_2 are unifiable, then $t_1 \succeq_C t_2$, thus the desired result follows. In the inductive step, the only choice possible initially is an application of transitivity, leading to the resolvent

$$:- c(\tau_1, \ldots, \tau_n) >= \alpha, \alpha >= s(\sigma_1, \ldots, \sigma_m).$$

Assume the leftmost atom here is selected. Another application of transitivity at this point will not make significant progress. Thus, it suffices to show that for any refutation beginning with an application of substitution for c there is one beginning with an application of a constraint for c. Suppose substitution is applied, leading to the resolvent

$$:- \tau_1 >= \beta_1, \ldots, \tau_n >= \beta_n, \quad c(\beta_1, \ldots, \beta_n) >= s(\sigma_1, \ldots, \sigma_m).$$

Assume the rightmost atom here is selected. By the inductive hypothesis, we may assume that the two-step application of a constraint $c(\alpha_1, \ldots, \alpha_n) >= \tau \in C$ occurs, leading to the resolvent

$$:- \tau_1 >= \beta_1, \ldots, \tau_n >= \beta_n, \quad \tau\{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\} >= s(\sigma_1, \ldots, \sigma_m).$$

Assume all but the last atom here are processed, leading to the resolvent

$$:- \tau\{\alpha_1 \mapsto \tau_1', \ldots, \alpha_n \mapsto \tau_n'\} >= s(\sigma_1, \ldots, \sigma_m).$$

where $\tau_i \succeq_C \tau_i'$ for $1 \leq i \leq n$. Returning now to the point of comparison, if the constraint for c is directly applied, the resolvent

$$:- \tau\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\} >= s(\sigma_1, \ldots, \sigma_m).$$

will be produced. It can be shown that if $\tau_i \succeq_C \tau_i'$ for $1 \leq i \leq n$ then $\tau\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\} \succeq_C \tau\{\alpha_1 \mapsto \tau_1', \ldots, \alpha_n \mapsto \tau_n'\}$, thus the desired result follows.

2. The proof is omitted since this case is not needed in this paper.

□

Our second restriction on type declarations ensures the termination of our iterative strategy for deriving subtypes. This restriction requires that recursively defined types "guard" their recursion by an outermost function symbol. As examples, given $\mathtt{f} \in F$ and $\mathtt{c} \in T$, the constraint `c >= f(c).` is acceptable but the constraints `c >= c.` and `c(A) >= c(f(A)).` are not. This restriction applies to mutually recursive types as well, for example, given $\mathtt{b} \in T$,

```
c(A) >= b(f(A)).
b(B) >= c(f(B)).
```

is not acceptable. It also applies to recursion occurring through the use of polymorphism, for example,

```
b(A) >= A.
c >= b(c).
```

is not acceptable. To define this restriction, we introduce the notion of "direct dependence" between type constructors.

Definition 8 (Direct Dependence) *Let C be a uniform polymorphic set of subtype constraints. $c \in T$ directly depends on $d \in T$ iff*

1. *there is a constraint $c(\alpha_1, \ldots, \alpha_n) >= \tau \in C$ and an occurrence of d in τ that is not in an argument to a function symbol, or*
2. *c directly depends on $b \in T$ and b directly depends on d.*

Definition 9 (Guarded) *A uniform polymorphic set C of subtype constraints is guarded iff there is no $c \in T$ which directly depends on itself.*

Theorem 3 (Correctness of Guarding) *Let C be a uniform polymorphic, guarded set of subtype constraints. Given any initial resolvent $:- c(\tau_1, \ldots, \tau_n) >= s(\sigma_1, \ldots, \sigma_m)$. where $c \in T$, every sequence of two-step applications of constraints in C eventually reaches a resolvent $:- \tau' >= s(\sigma_1, \ldots, \sigma_m)$. where the outermost symbol of τ' is not a type constructor.*

Proof: We give a proof sketch. The right-hand side of a constraint in C consists of either 1) a variable, 2) a type with an outermost function symbol, or 3) a type with an outermost type constructor. Two-step application of a constraint in the second form immediately reaches a desired resolvent, thus we need only consider the other two forms. Suppose we start with the resolvent $:- c(\tau_1, \ldots, \tau_n) >= s(\sigma_1, \ldots, \sigma_m)$. and perform a sequence of two-step applications of constraints in the first and third forms. Constraints of the third form will embed the arguments τ_i in terms containing type constructors d such that previous outermost type constructors directly depend on d. Constraints of the first form will strip off these type constructors d. Since no type constructor can directly depend on itself, it can be shown that after at most a finite number of steps, some τ_i will be uncovered. If the outermost symbol of τ_i is a type constructor, then the above reasoning process can be repeated. It can be repeated only a finite number of times since τ_i is strictly smaller than $c(\tau_1, \ldots, \tau_n)$. □

4 Variable Typings

In this section, we define a function *match* that forms the basis of our well-typedness conditions and prove its correctness. *match* returns a most general typing for the variables of a given term under a given type, if such a typing exists. As a first step, we introduce the notion of a typing for the variables in a term.

Definition 10 (Typings) *A typing for term t under type τ is a substitution θ mapping each variable in t to a type such that $\tau \succeq_C \overline{t\theta}$. Typing θ is respectful if $\overline{\tau} \succeq_C \overline{t\theta}$.*

For example, the following substitutions are typings for `X` under `list(A)`: $\{\mathtt{X} \mapsto \mathtt{list(A)}\}$, $\{\mathtt{X} \mapsto \mathtt{nelist(A)}\}$, $\{\mathtt{X} \mapsto \mathtt{list(int)}\}$, and $\{\mathtt{X} \mapsto \mathtt{list(B)}\}$. Of these, only the first and second are respectful. As a further example, every substitution over $\{\mathtt{X}\}$ is a typing for `f(X)` under `A`, but none is respectful. The following definition extends the notion of more general type, as introduced in definition 5, to typings.

Definition 11 (More General Typing) *Typing θ_1 for t is more general than typing θ_2 for t iff for all $x \in \mathrm{var}(t)$, $x\theta_1$ is more general than $x\theta_2$.*

For example, $\{\mathtt{X} \mapsto \mathtt{list(A)}\}$ is a more general typing for `X` than either $\{\mathtt{X} \mapsto \mathtt{nelist(A)}\}$ or $\{\mathtt{X} \mapsto \mathtt{list(int)}\}$.

Intuitively, $match(\tau, t) = \theta$ implies that 1) θ is a respectful typing for t under τ and 2) θ is more general than every other typing for t under τ. For example, $match(\mathtt{list(A)}, \mathtt{X}) = \{\mathtt{X} \mapsto \mathtt{list(A)}\}$. There are cases where no typing of any kind is possible, e.g., $match(\mathtt{int}, \mathtt{cons(X, Y)})$. When *match* recognizes such

cases, it returns the special value *fail*. There are also cases where several typings are possible but none is both respectful and most general. This can occur when a function symbol takes arguments of different types, e.g., $match(\mathtt{f(int)+f(list(A))}, \mathtt{f(X)})$; here both $\{\mathtt{X} \mapsto \mathtt{int}\}$ and $\{\mathtt{X} \mapsto \mathtt{list(A)}\}$ are respectful but neither is most general. Note that $\{\mathtt{X} \mapsto \mathtt{int} + \mathtt{list(A)}\}$ is not a typing here. It can also occur when the first argument to *match* is a variable, e.g., $match(\mathtt{A}, \mathtt{f(X)})$; here $\{\mathtt{X} \mapsto \mathtt{B}\}$ is most general but it is not respectful. In such cases, *match* returns the special value $\bot$. There are also cases where *match* loses track of what is going on and simply returns the value $\bot$. Specifically, *match* may fail to recognize that a respectful, most general typing exists, e.g., as in $match(\mathtt{f(int)} + \mathtt{f(nat)}, \mathtt{f(X)})$ and $match(\mathtt{f(int,nat)}), \mathtt{f(X,X)})$, or that no typing is possible, e.g., as in $match(\mathtt{f(int,list(A))}), \mathtt{f(X,X)})$. In general, the problem here is that some form of name-based type union and intersection are required. It is possible to extend *match* to pick up some of these cases.

Two typings, possibly for different terms, are said to be in agreement iff they produce equivalent types for common variables. Since this is a name-based system, type equivalence is taken to be syntactic equality.

Definition 12 (Agreement of Typings) *Typings θ_1 and θ_2 are in agreement iff for all $x \in \mathrm{dom}(\theta_1) \cap \mathrm{dom}(\theta_2)$, $x\theta_1 = x\theta_2$. A set S of typings is in agreement, denoted agree(S), iff its elements are pairwise in agreement.*

In the following definition, $c(\tau_1, \ldots, \tau_n) \rightarrow_C \sigma$ is taken to mean that $\sigma = \tau\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ for some constraint $c(\alpha_1, \ldots, \alpha_n) >= \tau \in C$.

Definition 13 (*match*) *Let C be a uniform polymorphic, guarded set of subtype constraints. Assume $x \in V$, $f, g \in F$, and $c \in T$.*

$match(\tau, x) = \{x \mapsto \tau\}$

$match(x, f(t_1, \ldots, t_n)) = \bot$

$match(g(\tau_1, \ldots, \tau_n), f(t_1, \ldots, t_m)) =$
  if $g/n \neq f/m$ then *fail*
  elseif $n = 0$ then $\{\}$
  else
    let $S = \{match(\tau_i, t_i) | 1 \leq i \leq n\}$ in
      if *fail* $\in S$ then *fail*
      elseif $\bot \in S$ *or* $\neg agree(S)$ then $\bot$
      else $\bigcup S$

$match(c(\tau_1, \ldots, \tau_n), f(t_1, \ldots, t_m)) =$
  let $S = \{match(\sigma, f(t_1, \ldots, t_m)) |$
    $c(\tau_1, \ldots, \tau_n) \rightarrow_C \sigma\}$ in
  if $S = \{fail\}$ then *fail*
  elseif $S = \{\theta\}$ *or* $S = \{\theta, fail\}$ then θ
  else $\bot$

Theorem 4 (Correctness of *match*) *Let τ be a type and t be a term over F.*

1. *$match(\tau, t) = \theta$ implies θ is a respectful, most general typing for t under τ.*
2. *$match(\tau, t) = fail$ implies there is no typing for t under τ.*

Proof: We prove these two claims by simultaneous induction over the height of the computation tree for $match(\tau, t)$. We first consider the base case for each of these claims.

1. In the first clause for *match*, $\{x \mapsto \tau\}$ is clearly a respectful, most general typing for x under τ. If $n = 0$ in the third clause for *match*, then $\{\}$ is clearly a respectful, most general typing for f under f.
2. Suppose $g/n \neq f/m$ in the third clause for *match*. By definition, θ is a typing for t under τ iff there is an SLD-refutation of $:- \tau >= \overline{t\theta}$. By theorem 1, such a refutation exists iff there is one which starts with an application of the substitution axiom for g. Since this axiom cannot be applied in this case, there is no such refutation and no such typing.

There are two cases to consider for each of the inductive steps. First, suppose τ is $f(\tau_1, \ldots, \tau_n)$ and t is $f(t_1, \ldots, t_n)$ in the third clause for *match*. By theorem 1, θ is a typing for t under τ only if it is a typing for every t_i under τ_i.

1. Suppose the members of S are typings that are in agreement. By the inductive hypothesis, the members of S are respectful, most general typings for the t_i under the τ_i. Therefore, their composition θ is a respectful, most general typing for every t_i under τ_i. Thus, θ is a respectful, most general typing for t under τ.
2. Suppose $fail \in S$, say, because of the i-th subterm. By the inductive hypothesis, there is no typing for t_i under τ_i, thus, there is no typing for t under τ.

Second, suppose τ is $c(\tau_1, \ldots, \tau_n)$ and t is $f(t_1, \ldots, t_m)$ in the fourth clause for *match*. By theorem 2, θ is a typing for t under τ iff it is a typing for t under $\tau'\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ for some $c(\alpha_1, \ldots, \alpha_n) >= \tau' \in C$.

1. Suppose $S = \{\theta\}$ or $S = \{\theta, fail\}$. By the inductive hypothesis, θ is a respectful, most general typing for t under $\tau'\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$ for some $c(\alpha_1, \ldots, \alpha_n) >= \tau' \in C$. Moreover, no other constraints defining c can produce typings for t. Thus, θ is a respectful, most general typing for t under τ.

2. Suppose $S = \{fail\}$. By the inductive hypothesis, none of the constraints defining c can produce typings for t under τ, thus, there is no typing for t under τ.

□

Theorem 5 (Termination of *match*) *$match(\tau, t)$ terminates for all τ and t.*

Proof: $match(\tau, t)$ terminates directly if the size of t is one. All recursive calls of *match* either decrease the size of t or leave it the same. The latter case occurs only in the fourth clause for *match*. By theorem 3, after at most a finite number of iterations here, *match* will be called with a first argument that is either a variable or a type with an outermost function symbol. If it is a variable, then the call terminates directly. If it is a type with an outermost function symbol, then a subsequent recursive call will decrease the size of t. □

We now present several lemmas concerning *match* that will be used to prove the consistency of our well-typedness conditions. The first lemma shows that instantiation of type variables propagates through *match*.

Lemma 1 (On Instantiation) *If $match(\tau, t) = \theta$ then $match(\tau\eta, t) = \theta\eta$ for any substitution η mapping variables of τ to types.*

Proof: The lemma follows from a straight-forward inductive argument over the height of the computation tree for $match(\tau, t)$. □

The second lemma shows that, under certain conditions, unification does not change the typing for variables. Throughout this paper, we assume that most general unifiers are idempotent and relevant [Apt88].

Lemma 2 (On Unification) *Let*

1. $\mathbf{var}(t_1) \cap \mathbf{var}(t_2) = \emptyset$,
2. $mgu(t_1, t_2) = \theta$,
3. $match(\tau, t_1) = \theta_1$, *and*
4. $match(\tau, t_2) = \theta_2$.

Then for all $x \in \mathbf{var}(t_1) \cap \mathbf{dom}(\theta)$, $match(x\theta_1, x\theta)$ is in agreement with θ_2.

Proof: Compare the computation tree T_1 of $match(\tau, t_1)$ with the computation tree T_2 of $match(\tau, t_2)$. Since t_1 and t_2 can be unified, these trees are identical up to their leaves that are variables. Consider any such variable $x \in \mathbf{var}(t_1) \cap \mathbf{dom}(\theta)$. T_1 records a type, say τ', for x while T_2 makes a recursive call $match(\tau', x\theta)$. The typings returned at this point appear directly in θ_1 and θ_2, thus, $x\theta_1 = \tau'$ and $match(\tau', x\theta)$ is in agreement with θ_2. The lemma follows directly. □

A corollary of the above lemma is that $match(\tau, t_1\theta)$, $match(\tau, t_1)$, and $match(\tau, t_2)$ are in agreement.

5 Predicate Types

In this section, we define predicate types and informally discuss type checking. As a first step, we review the syntax of logic programs and queries. Let a set P of predicate symbols with given arity, disjoint from V, F, and T, be given. An atom is a predicate symbol p/n applied to n terms over F. A program clause has the form h :- b. where h is an atom, called the head, and b is a list of atoms, called the body. A logic program consists of a sequence of program clauses. A negative clause, or query, has the form :- b. where b is a list of atoms.

The following definition introduces the notion of a predicate type.

Definition 14 (Predicate Types) *A predicate type for $p \in P$ has the form $p(\tau_1, \ldots, \tau_n)$ where $\tau_1, \ldots, \tau_n$ are types.*

Definition 15 (Type of an Atom) *Let $\mathcal{D}$ be a fixed set of predicate types, one for each $p \in P$. For any atom A, type(A) is the member of $\mathcal{D}$ associated with the predicate symbol of A.*

Predicate types are intended to restrict the allowable usage of predicates. In particular, we define well-typedness conditions which ensure that a program respects a set of predicate types; programs that are not well-typed are expected to be rejected by the type checker. These conditions ensure that every atom of every resolvent produced during execution is consistent with its type.

To ensure this property, we require that every variable in a clause appear in exactly one "type context". As an example of the problems that can arise if this restriction is not observed, consider the following declarations.

```
PRED p(int).
PRED q(list(A)).
```

Under these declarations, the query `:- p(X), q(X).` can lead to ill-typed resolvents such as `:- q(0)`. The problem here is that `X` appears as both an `int` and a `list(A)`. Similarly, the predicates

```
PRED r(list(A)).
  r(X) :- p(X).
```

and

```
PRED s(int,list(A)).
  s(X,X).
```

can lead to ill-typed resolvents because `X` appears as both an `int` and a `list(A)`. Thus, we require that the type checker reject programs where a variable appears in more than one type context in the same clause.

We also place requirements on the circumstances in which type variables can be instantiated. In particular, an invocation of a polymorphic predicate may make commitments regarding its type variables, however, a defining clause for the predicate may not. As an example, consider the predicates

```
PRED p(list(A)).
PRED q(list(int)).
```

The query `:- p(X), q(X)` is acceptable since `X` may be assigned the type `list(int)`. However, the program clause

```
p(cons(nil,nil)).
```

must be rejected because, for example, it would allow the above query to lead to the ill-typed resolvent `:- q(cons(nil,nil))`.

6 Well-Typed Programs

A program/query is well-typed iff each of its clauses is well-typed. Intuitively, a clause is well-typed iff a typing can be found for each of its atoms and these typings are in agreement. In the following definition, we treat predicate symbols as function symbols so *match* can be applied to atoms.

Definition 16 (Well-typed Clauses) *A program clause $A_0 :- A_1, \ldots, A_k.$ is well-typed iff there exist substitutions $\eta_1, \ldots, \eta_k$ such that*

$$match(type(A_0), A_0)$$

and

$$match(type(A_i)\eta_i, A_i) \;\; 1 \leq i \leq k$$

are in agreement. A negative clause $:- A_1, \ldots, A_k.$ is well-typed iff the above conditions excluding A_0 hold.

The substitutions $\eta_1, \ldots, \eta_k$ above allow the body atoms to make commitments regarding type variables.

The following theorem shows the consistency of our well-typedness conditions.

Theorem 6 (Consistency) *Every resolvent of a well-typed negative clause and a well-typed program clause is well-typed.*

Proof: Let

1. $N = :- B_1, \ldots, B_j.$ be a well-typed negative clause, in particular, let $match(type(B_i)\zeta_i, B_i)$ $1 \leq i \leq j$ be in agreement,
2. $C = A_0 :- A_1, \ldots, A_k.$ be a well-typed program clause, in particular, let $match(type(A_0), A_0)$ and $match(type(A_i)\eta_i, A_i)$ $1 \leq i \leq k$ be in agreement, and
3. $N' = (:- A_1, \ldots, A_k, B_2, \ldots, B_j.)\theta$ be any resolvent of N and C where $\theta = mgu(B_1, A_0)$; without loss of generality we assume the leftmost atom is always selected.

We will prove the resolvent N' is well-typed by showing $match(type(A_i)\eta_i\zeta_1, A_i\theta)$ $1 \leq i \leq k$ and $match(type(B_i)\zeta_i, B_i\theta)$ $2 \leq i \leq j$ are in agreement.

By lemma 1, $match(type(A_0)\zeta_1, A_0)$ and $match(type(A_i)\eta_i\zeta_1, A_i)$ $1 \leq i \leq k$ are in agreement. For any $1 \leq i \leq k$, compare the computation tree T_1 of $match(type(A_i)\eta_i\zeta_1, A_i)$ with the computation tree T_2 of $match(type(A_i)\eta_i\zeta_1, A_i\theta)$. These trees are identical up to the leaves of T_1, which consist of variables $x \in \mathbf{var}(C)$. For each variable $x \in \mathbf{var}(C) \setminus \mathbf{dom}(\theta)$, the trees are identical and therefore record the same type for x. Thus, $match(type(A_i)\eta_i\zeta_1, A_i\theta)$ $1 \leq i \leq k$ are in agreement for all variables in $\mathbf{var}(C) \setminus \mathbf{dom}(\theta)$, in particular, they are in agreement with $match(type(A_0)\zeta_1, A_0)$ and $match(type(A_i)\eta_i\zeta_1, A_i)$ $1 \leq i \leq k$. For each variable $x \in \mathbf{var}(C) \cap \mathbf{dom}(\theta)$, on the other hand, T_1 records a type, say τ, for x while T_2 makes a recursive call $match(\tau, x\theta)$ where $\mathbf{var}(x\theta) \subseteq \mathbf{var}(B_1)$. Since τ was also recorded for x by $match(type(A_0)\zeta_1, A_0)$, we can apply lemma 2 and conclude that $match(\tau, x\theta)$ is in agreement with $match(type(B_1)\zeta_1, B_1)$. Thus, $match(type(A_i)\eta_i\zeta_1, A_i\theta)$ $1 \leq i \leq k$ are in agreement for all variables in $\mathbf{var}(B_1)$, in particular, they are in agreement with $match(type(B_1)\zeta_1, B_1)$.

By a similar argument, $match(type(B_i)\zeta_i, B_i\theta)$ $2 \leq i \leq j$ are in agreement with $match(type(B_i)\zeta_i, B_i)$ $1 \leq i \leq j$ for all $x \in \mathbf{var}(N) \setminus \mathbf{dom}(\theta)$ and in agreement with $match(type(A_0)\zeta_1, A_0)$ for all $x \in \mathbf{var}(A_0)$. Thus, $match(type(A_i)\eta_i\zeta_1, A_i\theta)$ $1 \leq i \leq k$ and $match(type(B_i)\zeta_i, B_i\theta)$ $2 \leq i \leq j$ are in agreement and N' is well-typed. □

As a final comment, note that a corollary of the above theorem is that every answer substitution computed by a well-typed program is type consistent.

7 Concluding Remarks

This paper has presented a prescriptive type system for logic programs that supports parametric polymorphism and name-based subtypes. We introduced the notion of predicate types and defined well-typedness conditions which ensure that a program respects a set of predicate types. We are currently implementing a type checker that determines whether a program satisfies these conditions. The only non-effective part of these conditions

is the substitutions $\eta_1, \ldots, \eta_k$, occurring in the definition of well-typed clauses, that allow the body atoms to make commitments regarding type variables. To deal with this problem, our type checker uses a modified version of *match* that returns constraints on variables in its first argument. The constraints generated by the atoms in a clause are collected together and solved.

In our future work, we plan to more fully explore the use of subtypes in our system. Generally speaking, the introduction of subtypes into logic programming is somewhat problematic. As an example, given the declarations

```
PRED p(nat).
PRED q(int).
```

we would like to allow queries such as `:- p(X), q(X).` where information flows from the subtype to the supertype, as in the resolvent `:- q(succ(0))`. However, due to the non-directional nature of logic programming, information may also flow the other way, as in the inconsistent resolvent `:- p(pred(0))`. Note that this problem cannot be solved simply by keeping track of the order in which goals execute. In the above example, p might execute first but leave `X` uninstantiated, in which case q could instantiate `X` to `pred(0)`. Here, a type inconsistent answer substitution would be produced.

One solution to this problem, proposed in [DH88], is to require input/output modes which ensure that information flows in the appropriate direction, e.g.,

```
PRED p(OUT nat).
PRED q(IN int).
```

Another alternative, possible only in a system that supports typed unification [GM86, AKN86, Smo88], is to constrain `X` to be a `nat`, e.g., `:- p(X), X:nat, q(X).`

In the type system of this paper, the only way to formulate the above query is to explicitly define and use a "type conversion" predicate, e.g., `:- p(X), int2nat(X,Y), q(Y).` where

```
PRED int2nat(int,nat).
  int2nat(0,0).
  int2nat(succ(X),succ(X)).
```

This predicate filters out all `int`s that are not `nat`s. We are currently exploring a more general solution to this problem based on this notion of filtering.

References

[AKN86] H. Ait-Kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[Apt88] K. R. Apt. Introduction to logic programming. Technical Report TR-87-35, Department of Computer Sciences, University of Texas, Austin, 1988.

[Cla79] K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Department of Computing, Imperial College, 1979.

[DH88] Roland Dietrich and Frank Hagl. A polymorphic type system with subtypes for prolog. In *Proc 2nd European Symposium on Programming, Springer-Verlag LNCS 300*, 1988.

[GM86] J.A. Goguen and J. Meseguer. Eqlog: equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, Englewood Cliffs, 1986.

[Mis84] P. Mishra. Towards a theory of types in prolog. In *Proc of the International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[MO84] A. Mycroft and R. A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MR85] P. Mishra and U.S. Reddy. Declaration-free type checking. In *Proc Symposium on Principles of Programming Languages*, pages 7–21. ACM, 1985.

[Red88] Uday S. Reddy. Notions of polymorphism for predicate logic programs. In *Proc of the 5th International Symposium on Logic Programming*. IEEE, 1988.

[Smo88] Gert Smolka. Logic programming with polymorphically order-sorted types. In *Proceedings 1st International Workshop on Algebraic and Logic Programming*, 1988. Gaussig, GDR.

[YS87] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. In Ehud Shapiro, editor, *Concurrent Prolog Vol. 2*. MIT Press, 1987.

[Zob87] J. Zobel. Derivation of polymorphic types for prolog programs. In *Proc of the 4th International Symposium on Logic Programming*, pages 817–838. IEEE, 1987.