Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System

Extended Report [1]

T. K. Lakshman

Uday S. Reddy

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois 61801, USA

internet: ⟨lakshman,reddy⟩@cs.uiuc.edu

Abstract

Mycroft and O'Keefe [27] presented a declaration-based type system for Prolog. However, they did not clarify the semantics of the type system, leading to several criticisms being voiced against it. We propose that the language accepted by this type system be viewed as a typed variant of Prolog, called Typed Prolog. We define the formal semantics of Typed Prolog along the lines of many-sorted logic and polymorphic lambda calculus. Typed Prolog also supports a form of type inference called *type reconstruction* which takes a Typed Prolog program with missing type declarations, and reconstructs the most general type declarations satisfying the language definition. This approach contrasts with the inference based type systems which have been widely pursued heretofore.

Keywords: Type Systems, Polymorphism, Models, Fixed Points, Type Inference, Type Reconstruction.

[1] Short version published in the proceedings of ILPS '91 at San Diego, California, USA

1 Introduction

Static type checking in typed programming languages facilitates automatic error detection and debugging, code optimization and program analysis. However, logic programming languages such as Prolog do not possess a well defined notion of types. This is evident from the fact that an n-ary function symbol can be uniformly applied to *any* n-tuple of terms. Consequently, researchers in logic programming have focused on incorporating type disciplines into the inherently untyped logic programming languages so as to exploit the benefits of types.

Most of the research in the area of type systems for logic programming has followed a *descriptive* approach where the untyped semantics of the language is first defined independent of types. Types are then used to describe semantic properties of programs. This approach assumes the existence of a Universe of all objects. Types are viewed as subsets of the Universe and are often described by regular tree sets [11, 19, 25, 28, 36, 38] and sometimes by other approaches [16, 35]. One of the advantages of the descriptive approach is that there need not be any type declarations, and indeed, types can be *inferred* from the program. However, the inferred types may not exactly specify the programmer's intent. This approach has been followed in [25, 26, 28, 38]

An alternate *prescriptive* approach does not presuppose a Universe of all objects. Different domains are associated with distinct types. Type declarations form an integral part of programs and are used to determine the semantics of programs. An advantage of this approach is that type declarations can clearly specify the programmer's intent. This approach has been described in [10, 13, 22, 23, 29, 30].

Mycroft and O'Keefe [27] define a type system for a large subset of Prolog. Type declarations for functions and predicates are specified and used to augment the program clauses. However, Mycroft and O'Keefe do not define a typed semantics for their language. Instead, they assume a standard untyped semantics for Prolog and show that the type checker guarantees a certain well-behavedness property for the *operational* behavior of programs. This property, called the "semantic soundness" is offered as the only justification for the formal definition of the type system. This has caused a number of objections to be voiced against the type system. For instance, the type information has been categorized as being "extra-logical" [35]. The role played by the type system for an implementation different from SLD-resolution is also not clear [18].

We believe that the Mycroft-O'Keefe type system should be viewed *prescriptively*, i.e., the type system should enhance the language to a typed language and determine a typed semantics. The "semantic soundness" property of [27] is thus viewed as a *type consistency* property of the execution mechanism.

The notion of well-typed logic programs is defined independently of a specific execution mechanism. This view was originally expressed in [29] and was independently formulated by [10]. The proposal of [13] is also similar.

In this paper, we follow the *prescriptive* approach in defining Typed Prolog, a typed logic programming language that is a rational reconstruction of the ideas in [27]. In the next section, a type system that characterizes *well-formed* Typed Prolog programs is specified. The type systems of Lambda Prolog [23] and Godel [3] are also similar to this type system. This is followed by the specification of a typed model-theoretic semantics for Typed Prolog, based on many-sorted logic [6, 22] and polymorphic lambda calculus [7, 31]. Next, a procedural semantics for Typed Prolog is defined in terms of the least fixed point of an immediate consequence operator T_P over typed Herbrand interpretations. The equivalence of the two semantics is also shown.

We describe a type reconstruction algorithm similar to the one used in ML [24]. Even though type declarations form an integral part of a Typed Prolog program, it is possible to omit the declarations and have them automatically reconstructed by an algorithm. Thus, type inference in the prescriptively typed framework takes the form of *type reconstruction*.

This algorithm reconstructs most general types for "non-recursive" predicates given the types for functions. However, for recursive predicates, the types reconstructed by the algorithm are not guaranteed to be most general. In fact, it has recently been shown [17] that in general, no type reconstruction algorithm can derive most general types for "recursive" predicates. Consequently, the algorithm assumes, as in ML, that a recursive predicate is not used polymorphically within its definition. A Prolog implementation of this algorithm is also described.

Typed Prolog programs can be executed using the conventional untyped SLD resolution mechanism This because the conventional SLD resolution is type consistent for Typed Prolog programs. The evaluation of a well formed goal would never lead to ill formed subgoals. This implies that no runtime type checking is required.

We show that the type system accomodates other non-logical features such as cut, assert and retract, found in Prolog.

2 Typed Prolog

In this section, we define a Prolog-like (prescriptively) typed language following the work of Mycroft and O'Keefe [27]. The syntax of Typed Prolog enriches the Prolog syntax with type declarations. Type rules for Typed Prolog expressions are then defined.

2.1 Syntax

The expressions of Typed Prolog are constructed using three alphabets of symbols:

1. A ranked set T of *type constructors*. With every type constructor $\sigma \in T$ is associated a *rank* (or arity) denoting the number of type parameters it accepts. In particular, there is a type constructor *Unit* of rank 0, a type constructor *int* of rank 0 and a type constructor *list* of rank 1. We use the notation $type^{rank}$ (e.g., $list^1$) to denote type constructors, and omit the rank superscript when it is clear from the context.

2. A signed set F of *function symbols*. With every $f \in F$ is associated a *type signature* of the form $\tau_1 \times \cdots \times \tau_k \rightarrow \tau'$ (for $k \geq 0$) indicating the types of its arguments and result. Further, all the type variables appearing in $\tau_1, \ldots, \tau_k$ must also appear in τ'. This condition on function symbols was left tacit in [27]. Function symbols that satisfy this condition are called "type preserving" in [10].

   The type terms τ are described below. If $k = 0$, the domain type is formally taken to be *Unit*. However, for convenience, we write "$Unit \rightarrow \tau$" as simply τ.

3. A signed set P of *predicate symbols*. With every $p \in P$ is associated a type signature of the form $Pred(\tau_1 \times \cdots \times \tau_k)$ (for $k \geq 0$) indicating the types of its arguments.

In addition, there is a countably infinite set V of variable symbols X, and a countably infinite set Λ of type variable symbols α.

The expressions in the language belong to the following syntactic categories:

$$\begin{aligned} \tau &\in \mathit{Type} \\ t &\in \mathit{Term} \\ A &\in \mathit{Atom} \\ \phi &\in \mathit{Formula} \\ C &\in \mathit{Clause} \\ P &\in \mathit{Program} \end{aligned}$$

The unchecked "pre-expressions" belonging to these categories are given by the abstract syntax:

$$\begin{aligned} \tau &::= \alpha \mid \sigma(\tau_1, \ldots, \tau_k) \\ t &::= X \mid f(t_1, \ldots, t_k) \\ A &::= p(t_1, \ldots, t_k) \\ \phi &::= \epsilon \mid A \mid \phi_1, \phi_2 \mid t_1 = t_2 \\ C &::= [\forall\, X_1 : \tau_1, \ldots, X_n : \tau_n]\,(A \leftarrow \phi) \\ P &::= C_1 \ldots C_l \end{aligned}$$

Here, ϵ denotes the "empty" formula. "ϕ_1, ϕ_2" denotes the conjunction of ϕ_1 and ϕ_2. Clauses must include type declarations for all the variables used in them. This is in keeping with the principle of typed languages that all nonlogical symbols should be introduced with type declarations.

By "pre-expressions" we mean that not all expressions conforming to the above syntax are well-formed. To be well-formed, an expression must also satisfy the type rules given below. The context-free syntax given above is merely a convenient abstraction which, formally speaking, can be ignored. The type rules define a context-sensitive syntax for the language that should be taken to be the "official" syntax.

2.2 Type Rules

To express the type rules, we use the following forms of assertions:

t : τ	(t is a well-formed term of type τ)
A Atom	(A is a well-formed Atom)
ϕ Formula	(ϕ is a well-formed Formula)
C Clause	(C is a well-formed Clause)
P Program	(P is a well-formed Program)

In addition, we use *type contexts* Γ which are finite sets of unique type assertions for variables, each of the form $X : \tau$. A type context may also be viewed as a partial mapping $V \longrightarrow Type$

Let θ be a substitution from type variables to type terms. Then $\theta(\tau)$ denotes the type term τ with the substitution θ applied to it.

The type rules are:

$$\Gamma \vdash \mathrm{X} : \tau \qquad \text{if } (X : \tau) \in \Gamma$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \quad \ldots \quad \Gamma \vdash t_k : \theta(\tau_k)}{\Gamma \vdash f(t_1, \ldots, t_k) : \theta(\tau')} \qquad \text{if } f : \tau_1 \times \cdots \times \tau_k \rightarrow \tau'$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \quad \ldots \quad \Gamma \vdash t_k : \theta(\tau_k)}{\Gamma \vdash p(t_1, \ldots, t_k) \text{ Atom}} \qquad \text{if } p : Pred(\tau_1 \times \cdots \times \tau_k)$$

$$\Gamma \vdash \epsilon \text{ Formula}$$

$$\frac{\Gamma \vdash A \text{ Atom}}{\Gamma \vdash A \text{ Formula}}$$

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (t_1 = t_2) \text{ Formula}}$$

$$\frac{\Gamma \vdash \phi_1 \text{ Formula} \quad \Gamma \vdash \phi_2 \text{ Formula}}{\Gamma \vdash (\phi_1, \phi_2) \text{ Formula}}$$

$$\frac{\Gamma \vdash t_1 : \theta(\pi_1) \quad \ldots \quad \Gamma \vdash t_k : \theta(\pi_k) \quad \Gamma \vdash \phi \text{ Formula}}{\vdash [\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\,(p(t_1, \ldots, t_k) \leftarrow \phi) \text{ Clause}}$$

if $\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$, $p : Pred(\pi_1 \times \cdots \times \pi_k)$, and θ is a renaming substitution

$$\frac{\vdash C_1 \text{ Clause} \quad \ldots \quad \vdash C_l \text{ Clause}}{\vdash (C_1 \ldots C_l) \text{ Program}}$$

An expression is well-formed iff a corresponding judgement can be derived using the above inference rules. The well-formedness of terms and formulas is defined relative to a type context Γ, whereas clauses and programs are well-formed in the empty context. Since each expression form has a unique inference rule, the above rules may also be viewed as an inductive definition of the set of well-formed expressions.

Note that the use of the substitution θ in the function and predicate application rules signifies that, if a functor or predicate has some type, then it also has every instance of that type. This is how *polymorphism* is obtained.

The type rules can easily be transformed into a Prolog program whose termination follows by struc-

tural induction on the expressions. Thus, checking the well-formedness of a Typed Prolog program is decidable.

To illustrate the type rules, let us look at a few example programs in this language.

Example 1

Given the alphabets

$$T = \{man^0, woman^0\},\ \ F = \{john : man,\ mary : woman\},$$
$$P = \{married : Pred(man \times woman)\}$$

The following is a well-formed program:

$$married(john, mary) \leftarrow$$

It is not possible to define a predicate *spouse* with the interpretation:

$$spouse(X, Y) \leftarrow married(X, Y).$$
$$spouse(X, Y) \leftarrow married(Y, X).$$

because in that case, *spouse* must have both the types $Pred(man \times woman)$ and $Pred(woman \times man)$. The only way to include such a predicate would be to coalesce the types *man* and *woman* into a single type, say, *person*. Then, the following signatures can be assigned:

$$T = \{person^0\},\ \ F = \{john : person,\ mary : person\},$$
$$P = \{married : Pred(person \times person),\ spouse : Pred(person \times person)\}$$

Example 2

To see the use of polymorphism, consider the alphabets:

$$T = \{list^1, \ldots\},\ \ F = \{nil : list(\alpha),\ \text{“.”} : \alpha \times list(\alpha) \rightarrow list(\alpha)\},$$
$$P = \{append : Pred(list(\alpha) \times list(\alpha) \times list(\alpha))\}$$

Then the following is a well-formed program:

$$[\forall Y : list(\alpha)] \quad append(nil, Y, Y) \leftarrow$$
$$[\forall A : \alpha, X : list(\alpha), Y : list(\alpha), Z : list(\alpha)]$$
$$append(A.X, Y, A.Z) \leftarrow append(X, Y, Z).$$

The predicate *append* is polymorphic in that it can be used to append lists of any type. The use of type variables such as α in the type signature of the predicate demonstrates such polymorphism.

2.3 Definitional Genericity

The restrictions in the type rule for clauses must be carefully noted. These restrictions state that the type of a *defining occurrence* of a predicate (an occurrence to the left of "$\leftarrow$") must be equivalent upto renaming to the assigned type signature of the predicate. We call this condition the principle of *definitional genericity*. This condition is imposed on all clauses. For example, it would be incorrect to restate the clauses of *append* with α replaced by a specific type such as *int*. The type of *append* in its defining occurrences would then be more specific than its assigned type signature. Such specificity is prohibited by the definitional genericity constraint imposed on the clauses. Note that in contrast, the types of *uses* of a predicate (occurrences to the right of "$\leftarrow$") can be more specific than the assigned type signature.

The type rule for clauses used in [13] is similar to the above type rule. In [10] a similar condition called "type-general definition" is used to characterize the predicates whose clauses have the above property. However an alternate type rule for clauses is used in [10]:

$$\frac{\Gamma \vdash p(t_1 \ldots t_k)\ Atom \quad \Gamma \vdash \phi \text{ Formula}}{\vdash [\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\,(p(t_1, \ldots, t_k) \leftarrow \phi) \text{ Clause}} \quad \text{if } \Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\},$$

The above rule allows the defining occurrences of a predicate to have a more specific type than the assigned type signature.

The definitional genericity principle is consistent with the view of Prolog programs as "definitions" of predicates rather than as axioms. A formal expression of this view is Clark's iff-completion semantics [4]. If programs are viewed as definitions, then the clauses for a predicate like append above should define the polymorphic predicate append, rather than some particular monomorphic instance of it. If, on the other hand, programs are viewed as axioms, then it would seem reasonable to add a fact such as $append([1,2,3],[4],[1,2,3,4])$ which would not satisfy the definitional genericity principle.

3 Semantics of Typed Prolog

In this section, we define a typed model-theoretic semantics for Typed Prolog. The semantics is based on that of many-sorted logic [6, 22] and polymorphic lambda calculus [7, 31]. We show that the classical results for untyped logic programs [20], are also valid for Typed Prolog programs.

The semantics takes types into account in a fundamental way. Firstly, we require that every ground type term τ denote a distinct domain of values. Secondly, since the language consists of only well-formed

terms and formulas, only such terms and formulas are assigned meaning. Further, the interpretation of a term t should denote a value in the domain denoted by "its type". Since terms have types only with respect to particular type contexts, the type context as well as the type of a term play a role in its interpretation. Moreover, the definition of an interpretation needs information about a particular type derivation used to derive the type of the term. Thus, the semantics of Typed Prolog is considerably more complex than that of its untyped variant.

3.1 Interpretations

An *Interpretation* I is a tuple $\langle \mathcal{D}, \mathcal{T}, \mathcal{F}, \mathcal{P} \rangle$ where $\mathcal{D}$ is a set of non-empty sets called "domains", and $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{P}$, are the interpretation functions for type constructors, function symbols and predicate symbols (respectively). Specifically:

- $\mathcal{T}(\sigma) \ : \ \mathcal{D}^n \rightarrow \mathcal{D}$ assigns to every type constructor σ of arity n, a function in $\mathcal{D}^n \rightarrow \mathcal{D}$. Thus, every *ground* type term τ can be interpreted as a domain $\mathcal{D}_\tau$. However, non-ground type terms like $list(\alpha)$ cannot directly be interpreted as domains. Consequently, a *type valuation* $v \ : \ \Lambda \rightarrow \mathcal{D}$ is defined to map type variables to domains. A non-ground type term τ can now be interpreted as a domain $\mathcal{D}_{\tau,v}$ via the use of v to interpret the type variables in τ.

$$\mathcal{D}_{\alpha,v} = v(\alpha).$$
$$\mathcal{D}_{\sigma(\tau_1,\ldots,\tau_k),v} = \mathcal{T}(\sigma)(\mathcal{D}_{\tau_1,v}, \ldots, \mathcal{D}_{\tau_k,v}).$$

- $\mathcal{F}(f)$ assigns to every function symbol f of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$, a family of functions f_v indexed by type valuations v. For every type valuation v for the type variables in the type of f, there is a function f_v in the corresponding function space $\mathcal{D}_{\tau_1,v} \times \ldots \times \ \mathcal{D}_{\tau_n,v} \rightarrow \mathcal{D}_{\tau,v}$ that interprets f.

- $\mathcal{P}(p)$ assigns to every predicate symbol p of type $Pred(\tau_1 \times \ldots \times \tau_n)$, a family of relations p_v indexed by type valuations v. For every type valuation v for the type variables in the type of p, there is a relation p_v over $\mathcal{D}_{\tau_1,v} \times \ldots \times \mathcal{D}_{\tau_n,v}$ that interprets p.

This semantics of polymorphism via parameterization is motivated by polymorphic lambda calculus [7, 31]. Hanus [10] and Yardeni et. al. [37] also use a similar interpretation.

A *valuation* ζ is a mapping $\zeta : V \rightarrow \bigcup_\tau \mathcal{D}_\tau$. A valuation ζ is said to *respect* a type context Γ under a type valuation v if, for all $X \in V$, $\zeta(X) \in \mathcal{D}_{\Gamma(X),v}$. If θ is a type substitution mapping type variables

Λ to type terms over a set of type variables Λ', and v is a type valuation for Λ', then we denote by $v \circ \theta$ the type valuation $v \circ \theta(\alpha) = \mathcal{D}_{\theta(\alpha),v}$.

The *extension of the interpretation* $\mathcal{F}$ to terms, denoted by $\bar{\mathcal{F}}_{v,\zeta}$, is defined with respect to a type valuation v and a valuation ζ. It applies to well-formed terms $\Gamma \vdash t : \tau$ provided ζ respects Γ under v. Since we are only interested in well-formed terms, we refer to sequents of the form $\Gamma \vdash t : \tau$ as "terms".

$$\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash X : \tau] = \zeta(X) \text{ if } X : \tau \in \Gamma.$$
$$\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash f(t_1, \ldots, t_k) \ : \ \theta(\tau')] =$$
$$\mathcal{F}(f)_{v \circ \theta}(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\tau_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\tau_k)])$$
$$\text{if } f : \tau_1 \times \ldots \times \tau_k \rightarrow \tau'.$$

For example, given $nil : list(\alpha)$,

$$\bar{\mathcal{F}}_{v,\zeta}[\Gamma \ \vdash \ nil : \ list(\alpha)] = \mathcal{F}(nil)_{[\alpha \rightarrow v(\alpha)]} \text{ and}$$
$$\bar{\mathcal{F}}_{v,\zeta}[\Gamma \ \vdash \ nil : \ list(int)] = \mathcal{F}(nil)_{[\alpha \rightarrow int]}.$$

In other words, $\Gamma \vdash nil \ : \ list(\alpha)$ denotes a polymorphic value, whereas $\Gamma \vdash nil \ : \ list(int)$ denotes a specific value in $\mathcal{D}_{list(int)}$.

An interpretation I is extended to well-formed formulas with respect to a type valuation v and a valuation ζ. It applies to formulas "$\Gamma \vdash \phi$ Formula", provided ζ respects Γ under v, and yields a truth value.

$\bar{I}_{v,\zeta}[\Gamma \vdash \epsilon \text{ Formula}]$ is true.

$\bar{I}_{v,\zeta}[\Gamma \vdash p(t_1, \ldots t_k) \text{ Formula}]$ iff
$(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\tau_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\tau_k)]) \in \mathcal{P}(p)_{v \circ \theta}$
where $p : Pred(\tau_1 \times \ldots \times \tau_k)$.

$\bar{I}_{v,\zeta}[\Gamma \vdash (t_1 = t_2) \text{ Formula}]$ iff $\bar{\mathcal{F}}_{v,\zeta}\,[\Gamma \vdash t_1 : \tau] = \bar{\mathcal{F}}_{v,\zeta}\,[\Gamma \vdash t_2 : \tau]$.

$\bar{I}_{v,\zeta}[\Gamma \vdash \phi_1, \phi_2 \text{ Formula}]$ iff $\bar{I}_{v,\zeta}[\Gamma \vdash \phi_1 \text{ Formula}]$ and $\bar{I}_{v,\zeta}[\Gamma \vdash \phi_2 \text{ Formula}]$.

Suppose a clause is derived via a type derivation with the following last step

$$\frac{\Gamma \vdash t_1 : \theta(\pi_1) \quad \ldots \quad \Gamma \vdash t_k : \theta(\pi_k) \quad \Gamma \vdash \phi \text{ Formula}}{\vdash [\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\,(p(t_1, \ldots, t_k) \leftarrow \phi) \text{ Clause}}$$

if $\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$, $p : Pred(\pi_1 \times \cdots \times \pi_k)$, and θ is a renaming substitution.

An interpretation I satisfies the clause iff, for all type valuations v and all valuations ζ that respect Γ under v, one of the following holds:

- $\bar{I}_{v,\zeta}[\Gamma \vdash \phi$ Formula] is false.
- $(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\pi_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\pi_k)]) \in \mathcal{P}(p)_{v \circ \theta}$.

We denote this by $I \models [\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\ (p(t_1, \ldots, t_k) \leftarrow \phi)$. Finally, I satisfies a program $C_1, \ldots, C_l$ iff $I \models C_i$ for each $i = 1, \ldots, l$. We also say that I is a *model* of the program.

3.2 Model Intersection Property

Let $\mathcal{M}$ be a (non empty) set of models of a Typed Prolog program which share the same $\mathcal{D}$, $\mathcal{T}$ and $\mathcal{F}$ components but differ in the predicate interpretations $\mathcal{P}$. For $M_1,\ M_2\ \in \mathcal{M}$, we say that $M_1 \subseteq M_2$ if the $\mathcal{P}$ component of M_1 is included in the $\mathcal{P}$ component of M_2. Let $M_o = \bigcap \mathcal{M}$ be the model with the same $\mathcal{D}$, $\mathcal{T}$ and $\mathcal{F}$ components but with the predicate interpretation obtained by intersecting the $\mathcal{P}$ components of the models in $\mathcal{M}$.

Lemma 1 If $\Gamma \vdash \phi$ Formula and for some $M \in \mathcal{M}\ \ \bar{M}_{v,\zeta}[\Gamma \vdash \phi$ Formula] is false then $(\bar{M}_o)_{v,\zeta}[\Gamma \vdash \phi$ Formula] is false.

`Proof:` By induction on the type derivation of "$\Gamma \vdash \phi$ Formula". We show the case $\Gamma \vdash p(t_1, \ldots, t_k)$ Formula where $p : Pred(\tau_1 \times \ldots \times \tau_k)$ and the last step of the derivation uses a type substitution θ. The induction hypothesis amounts to $(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\tau_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\tau_k)]) \notin \bar{M}(p)_{v \circ \theta}$. Therefore, the tuple does not belong to the intersection $(\bar{M}_o)(p)_{v \circ \theta}$.

□

Theorem 2 If $\mathcal{M}$ is a set of models of a program P, with the same $\mathcal{D}$, $\mathcal{T}$ and $\mathcal{F}$ components, then $M_o = \bigcap \mathcal{M}$ is a model of P.

`Proof:` Let $[\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\ (p(t_1, \ldots, t_k) \leftarrow \phi)$ be a clause in the program where $p : Pred(\pi_1 \times \ldots \times \pi_k)$ and the last step in the type derivation uses a renaming substitution θ. Let $\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$ be the induced type context. By the hypothesis, every $M \in \mathcal{M}$ satisfies the clause. That is, for every v and ζ that respects Γ under v, either $\bar{M}_{v,\zeta}[\Gamma \vdash \phi$ Formula] is false or $(\bar{F}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\pi_1)], \ldots, \bar{F}_{v,\zeta}[\Gamma \vdash t_k : \theta(\pi_k)]) \in \bar{M}(p)_{v \circ \theta}$.

Let v_o and ζ_o be a type valuation and a valuation respectively. If, for some $M \in \mathcal{M}$, $\bar{M}_{v_o,\zeta_o}[\Gamma \vdash \phi$ Formula] is false, then $(\bar{M}_o)_{v_o,\zeta_o}[\Gamma \vdash \phi$ Formula] is false. If, on the other hand, for every model

$M \in \mathcal{M}$, $\bar{M}_{v_o,\zeta_o}[\Gamma \vdash \phi \text{ Formula}]$ is true, then $(\bar{F}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\pi_1)], \ldots, \bar{F}_{v,\zeta}[\Gamma \vdash t_k : \theta(\pi_k)]) \in \bar{M}(p)_{v_o \circ \theta}$. So, the tuple belongs to $(M_o)(p)_{v_o \circ \theta}$.

By quantifying over v_o and ζ_o, we conclude that M_o is a model of the clause.

□

Corollary 3 For a given $\mathcal{D}$, $\mathcal{T}$ and $\mathcal{F}$, every Typed Prolog program P has a least model.

`Proof:` Let $\{M_i\}$ be the set of all models of P. Note that $\{M_i\}$ is non-empty, since the interpretation that maps every predicate $p : Pred(\tau_1 \times \ldots \times \tau_k)$ to true everywhere in its v-indexed domain $D_{\tau_1,v} \times \ldots \times D_{\tau_k,v}$, is trivially a model. Thus, by the previous theorem , $M_o = \bigcap \mathcal{M}$ is the least model for P (since it is contained in every other model M).

□

Next, we define typed Herbrand interpretations and typed Herbrand models. We then show an equivalence between the declarative notions of least typed Herbrand model and the procedural notions of least fixpoint.

3.3 Typed Herbrand Interpretations

The domains of typed Herbrand interpretations are sets of well-formed ground terms of types τ. Further, function symbols are interpreted as free functions. Only well-formed ground terms appear in the domains. Hence, the union of the typed Herbrand domains is a proper subset of the untyped Herbrand universe. Ill-formed terms are not included in the domains and are not interpreted at all.

For each ground type term τ, define H_τ to be the set of well-formed ground terms of type τ, i.e., terms t such that $\vdash t : \tau$. A *Typed Herbrand Interpretation* I has the following $\mathcal{D}, \mathcal{T}$ and $\mathcal{F}$ components:

- $\mathcal{D}$ is the set of H_τ's for all type terms τ.
- $\mathcal{T}(\sigma^n)$ maps $H_{\tau_1}, \ldots, H_{\tau_n}$ to $H_{\sigma(\tau_1,\ldots,\tau_n)}$.
- $\mathcal{F}(f)$ is the family of functions $f_v : \mathcal{D}_{\tau_1,v} \times \ldots \times \mathcal{D}_{\tau_n,v} \rightarrow \mathcal{D}_{\tau,v}$ which maps tuples of terms $(t_1, \ldots, t_n)$ to the term $f(t_1, \ldots, t_n)$.

A *Typed Herbrand Model* of a Typed Prolog program P is is a typed Herbrand interpretation which is a model of P.

The *Typed Herbrand Base* B_P for a program P is the typed Herbrand interpretation $\langle \mathcal{D}, \mathcal{T}, \mathcal{F}, \mathcal{P} \rangle$ such that for all predicates $p : Pred(\tau_1 \times \ldots \times \tau_n)$, $\mathcal{P}(p)_v$ is $\mathcal{D}_{\tau_1,v} \times \ldots \times \mathcal{D}_{\tau_n,v}$. Thus, $p(t_1, \ldots, t_n)$ is true for all tuples of well-formed ground terms of the appropriate types. Note that the typed Herbrand base is always a model of a well-formed Typed Prolog program.

A typed Herbrand interpretation is identified with a subset of the $\mathcal{P}$ component of the typed Herbrand base that consists of all well-formed ground atoms which are true with respect to the interpretation. 2^{B_P} is the set of all typed Herbrand interpretations of P. For $I_1,\ I_2\ \in 2^{B_P}$, we say that $I_1 \subseteq I_2$ if the $\mathcal{P}$ component of I_1 is included in the $\mathcal{P}$ component of I_2. Note that $\langle 2^{B_P}, \subseteq \rangle$ is a complete lattice.

Theorem 4 A well-formed Typed Prolog program has a model iff it has a Typed Herbrand model.

`Proof:`

if Trivial, since a Typed Herbrand Model *is* also a model.

only if Consider an interpretation I for a Typed Prolog program. Define the $\mathcal{D}, \mathcal{T}$ and $\mathcal{F}$ components of a typed Herbrand interpretation I' as before. I' is identified with a subset of the typed Herbrand Base B_P as follows:

$I' = \{p(t_1, \ldots, t_n) \mid$ withrespecttoatypevaluation v avaluation ζ, andatypecontext Γ,
suchthatζ respects Γ under$v$
$\bar{I}_{v,\zeta}[\Gamma \vdash p(t_1, \ldots, t_n)$ Formula] is true.$\}$

Since "$\Gamma \vdash p(t_1, \ldots, t_n)$ Formula" is a well-formed formula, the typed Herbrand interpretation I' is identified with well-formed formulas only.

If I is a model then so is I'.

□

As a notational convenience, we denote by $I_{\mathcal{P}}$, a typed Herbrand interpretation I whose predicate component is $\mathcal{P}$.

Let M be a (non empty) set of typed Herbrand models of a Typed Prolog program P. The *Least Typed Herbrand Model* of P is defined as $M_P = \bigcap M$.

3.4 Fixpoint Characterizations

For a Typed Prolog program, let $I_{\mathcal{P}}$ be a typed Herbrand interpretation and let $\bar{I}_{\mathcal{P}}$ be the extension of $I_{\mathcal{P}}$ (as defined earlier) to well-formed formulas. An *immediate consequence operator* $T_P : 2^{\mathcal{P}_P} \longrightarrow 2^{\mathcal{P}_P}$

that operates on the $\mathcal{P}$ components of typed Herbrand interpretations is now defined as follows:

$T_P(I_{\mathcal{P}}) = I_{\mathcal{P}'}$ where $I_{\mathcal{P}'}$ is such that for every clause $[\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\,(p(t_1, \ldots, t_k) \leftarrow \phi)$ in the program with $p : \ Pred(\pi_1 \times \ldots \times \pi_k)$ and $\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$, and for all type valuations v, and valuations ζ that respect Γ under v, $(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\pi_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\pi_k)]) \in \mathcal{P}'(p)_{v \circ \theta}$ if $(\bar{I}_{\mathcal{P}})_{v,\zeta}[\Gamma \vdash \phi : \text{ Formula}]$ is true.

A set S $\subseteq 2^{\mathcal{B}_P}$ is *directed* if every finite subset of S has an upper bound in S.

Theorem 5 T_P is continuous, i.e., for each *directed* subset S of 2^{B_P}, $T_P(lub(S)) = lub(T_P(S))$.

`Proof:` Consider a clause $[\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\,(p(t_1, \ldots, t_k) \leftarrow \phi)$ where $p : \ Pred(\tau_1 \times \ldots \times \tau_k)$. Let $\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$. Then, for all type valuations v and valuations ζ that respect Γ under v, $(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\tau_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\tau_k)]) \in T_P(lub(S))(p)_{v \circ \theta}$.
iff $(\bar{I}_{lub(S)})_{v,\zeta}[\Gamma \vdash \ \phi$ Formula] is true (by definition of T_P).
iff $(\bar{I}_{\mathcal{P}})_{v,\zeta}[\Gamma \vdash \ \phi$ Formula] is true for some $\mathcal{P} \subseteq S$ (since S is directed).
iff $(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\tau_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\tau_k)]) \in T_P(\mathcal{P})(p)_{v \circ \theta}$ (by the definition of T_P).
iff $(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\tau_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\tau_k)]) \in lub(T_P(S))(p)_{v \circ \theta}$ (since $T_P(\mathcal{P}) \subseteq lub(T_P(S))$).

□

Corollary 6 (Tarski) The least fixed point of T_P exists [33].

□

Theorem 7 Let I be a typed Herbrand interpretation of a Typed Prolog program P. Then, I is a typed Herbrand model of P iff $T_P(I) \subseteq I$.

`Proof:` $I_{\mathcal{P}}$ is a model for P iff $I_{\mathcal{P}}$ satisfies each clause in P.
iff for every clause C_l of the form $[\forall X_1 : \tau_1, \ldots, X_n : \tau_n]\,(p(t_1, \ldots, t_k) \leftarrow \phi)$ $I_{\mathcal{P}} \models C_l$.
iff for all type valuations v and all valuations ζ that respect the induced type context
$\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$ under v, either $(\bar{I}_{\mathcal{P}})_{v,\zeta}[\Gamma \vdash \phi$ Formula] is false or
$(\bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_1 : \theta(\pi_1)], \ldots, \bar{\mathcal{F}}_{v,\zeta}[\Gamma \vdash t_k : \theta(\pi_k)]) \in \mathcal{P}(p)_{v \circ \theta}$.
iff $T_P(\mathcal{P}) \subseteq \mathcal{P}$ in either case (by definition of $T_P(\mathcal{P})$).

□

Theorem 8 The least typed Herbrand model M_P of a Typed Prolog program P is the least fixed point of T_P.

`Proof:` $M_P = \bigcap \{ I_{\mathcal{P}} : I_{\mathcal{P}}$ is a typed Herbrand model for P$\}$.

$= \bigcap \{ I_{\mathcal{P}} : T_P(\mathcal{P}) \subseteq \mathcal{P} \}$.

$= \bigcap \{ I_{\mathcal{P}} : T_P(\mathcal{P}) = \mathcal{P} \}$ (by Tarski's Theorem).

$= lfp(T_P)$.

□

Thus, SLD-execution of Typed Prolog programs corresponds to the typed model-theoretic semantics. Further, the type consistency results to be stated in section 5 imply that the standard untyped SLD-resolution can be used to execute Typed Prolog programs without requiring any runtime type checking.

4 Type Reconstruction

In the prescriptively typed framework, a "completely-typed" program in Typed Prolog must contain type declarations for function symbols in F and predicate symbols in P. In addition, each clause must contain type declarations for all the variables used in it. Since requiring type declarations for each of the above alphabets is too cumbersome, a "partially typed" program dispenses with as much of type declarations as possible. Instead, the "omitted" type declarations are *inferred* via a "type reconstruction" algorithm.

A *completely typed* program P^c in Typed Prolog is a tuple $\langle T, F, P, Prog \rangle$ where T is a set of type constructors, F is a set of type declarations for function symbols, P is a set of type declarations for predicate symbols, and $Prog$ is a set of program clauses. With every function (predicate) symbol appearing in the program is associated a unique type declaration in F (P). Likewise, every variable appearing in a clause, has a type declaration for it in the clause.

The *type erasure* of a completely typed program P^c in Typed Prolog is a *partially typed program* P^p which is a tuple $\langle T, F, P^e, Prog^e \rangle$ defined as follows:

- P^e is obtained from P by erasing type declarations for predicate symbols,
- $Prog^e$ is obtained from $Prog$ by erasing from each clause, type declarations for the variables used in it.

We denote this by $erase(P^c) = P^p$.

A *type reconstruction algorithm* takes as input a partially typed program P^p and produces a completely typed program P^c that satisfies the well-formedness rules such that $erase(P^c) = P^p$.

4.1 Type Reconstruction for variables

Type reconstruction for variables is the process of computing the type context. Since each expression has a *unique* type rule, given the type declarations for functors and predicate symbols, we can uniquely compute the most general type context.

Thus, the type reconstruction problem for variables in a Typed Prolog program is well defined. There exists a unique (upto renaming) most general type of a variable. The implementation [21] does reconstruct most general types for variables. Type reconstruction for variables has also been implemented for the Mycroft-O'Keefe type system [27].

4.2 Type Reconstruction for predicates

A predicate symbol p is said to *derive* a predicate symbol q in a program P ($p \rightsquigarrow q$) if there is a clause in P in which p occurs in the head and q occurs in the body. We denote by $\rightsquigarrow^+$ the transitive closure of the derives relation $\rightsquigarrow$ in P. A predicate symbol p is said to be *self-recursive* if $p \rightsquigarrow p$. Predicate symbols p and q are *mutually recursive* if $p \rightsquigarrow^+ q$ and $q \rightsquigarrow^+ p$. p is said to be *recursive* if $p \rightsquigarrow^+ p$ in P. Note that this definition includes self-recursion as well as mutual-recursion. A predicate symbol p is said to be *recursively defined* in a program P if $p \rightsquigarrow^+ q$ and q is "recursive" in P. Otherwise, p is said to be *non-recursively* defined in P.

The problem of reconstructing most general type signatures for recursive polymorphic predicates is closely related to the problem of semi-unification [12]. It has been shown in [12] that semi-unification characterizes (upto polynomial time equivalence) the problem of inferring the most general types for recursive polymorphic predicates in a calculus of type rules that can be used to model Typed Prolog. However, it has recently been shown [17] that semi-unification is in general undecidable. Hence, type reconstruction for recursive polymorphic predicates in Typed Prolog is also undecidable.

In view of the undecidability of the type reconstruction problem for predicates, the algorithm assumes that the type associated with a body occurrence of a recursive polymorphic predicate is *identical* to (rather than an instance of) its type signature. This assumption results in a loss of generality in the reconstructed type (see example 4). However, we believe that this loss of generality is not serious in

practice. This is evidenced by the large number of functional programming languages which use this type reconstruction algorithm [2, 14, 34].

4.3 Type Reconstruction Algorithm

The type reconstruction algorithm is similar to the one used in ML [24] and has been implemented in Prolog [21].

Input A partially typed program in Typed Prolog with type declarations Td, Fd and Pd for type constructors, function symbols, and predicate symbols respectively. Type declarations are mandatory for all function symbols but may be omitted for predicate symbols. Type declarations for variables in clauses are omitted.

Algorithm The algorithm reconstructs missing predicate types and types for variables as follows:

1. In order to reconstruct the type of a predicate symbol p, the algorithm needs to consider all clauses that define p. Further, for each of these clauses, the types of all the body atoms must be known. Finally, for efficiency reasons, while checking for well-formedness, the algorithm should be able to type-check the program, one clause at a time without having to repeat the type-checking of any clause. The algorithm satisfies these conditions by reordering the clauses in P so that

   - Definitions for a predicate symbol p appear before definitions for predicate symbols r that derive p.
   - For a recursively defined predicate symbol p, the definitions of p and the definitions of all predicate symbols q that are mutually recursive to p must appear together and before the definitions of any predicate symbol r which derives p and is not mutually recursive to p.

   The reordering is a topological sort on the predicate definitions, where mutually recursive predicate definitions appear together. Reordering ensures that when reconstructing the type of a non-recursively defined predicate from a clause, the types for the body atoms are known. Furthermore, for recursive predicates, the type of a body occurrence is treated as being identical to the reconstructed type. Hence, even in this case, the types for the body atoms are known.

2. For each clause in the reordered program that defines a predicate symbol p, the algorithm checks for well-formedness and reconstructs types (if required) for p in the head $p(t_1, \ldots, t_n)$. This is

done by computing the type τ_i of each argument t_i from the declared types of function symbols. Thus, the type of p is $p : Pred(\tau_1 \times \ldots \times \tau_n)$, where τ_i (possibly) contains type variables that specify the types of the variables occurring in t_i. At this point, the types for these variables are also reconstructed using the algorithm described earlier. Note that variables are local to a clause and a given variable may have distinct types in different clauses. The algorithm also reconstructs types for any predicate symbol q in the body that is (mutually) recursive to p, by assuming that the type of the body occurrence of q is *identical* to the reconstructed type signature for q. It is due to this assumption that the reconstructed type for such a predicate symbol is not guaranteed to be most general(see example 4).

3. The reconstructed type of the predicate symbol p is the most general unifier of the types $p : Pred(\tau_1 \times \ldots \times \tau_n)$ computed from the different clauses defining p.

Output A completely typed program P^c such that by *erasing* the reconstructed predicate types and the reconstructed types for variables in clauses in P^c, we obtain the partially typed input program P^p.
Implementation The Prolog procedure that implements the type-checking / type-reconstruction algorithm is included in Appendix A. The complete implementation is written in Prolog and has been tested on C Prolog, SB Prolog, SICStus Prolog and Quintus Prolog.

The above type reconstruction algorithm reconstructs *most general* type signatures for non-recursively defined predicates.

In order to obtain a characterization of the reconstructed types for recursively defined predicates, we restrict our attention to the type reconstruction problem for a single recursive predicate p, wherein, given the types of all q (distinct from p), such that $p \rightsquigarrow q$, it is required to reconstruct the type for p. This characterization can be extended to type reconstruction for multiple (possibly mutually recursive) predicates which has been implemented [21].

Let TS be a new type system whose type rules are identical to the type rules for Typed Prolog with the exception that in the type rule for clause, the type variables appearing in the types for variables are replaced by "new" type constructors. TS permits only *monomorphic recursion* i.e., for a predicate p such that $p \rightsquigarrow p$, the type of body occurrences of p in a clause defining p must be identical to (and not a polymorphic instances of) the type of the defining occurrence of p.

Theorem 9 For the type reconstruction problem for a single predicate, the output (P^c) of the type reconstruction algorithm is a well-formed program under TS.

To illustrate the type reconstruction algorithm let us consider a few example programs in Typed Prolog.

Example 3

Consider as input, the program from example 2 where the type declaration for *append* is omitted.

> From the first clause the type $married : Pred(man \times woman)$ is computed (since $john : man,\ mary : woman$ are in F). From the second clause the type $married : Pred(man \times woman)$ is computed (since $bob : man,\ mary : woman$ are in F). The reconstructed type of *married* is $Pred(man \times woman)$ as computed by unifying the reconstructed types of the above definitions of *married*.

□

The following example illustrates the problem with reconstructing types for recursively defined polymorphic predicates.

Example 4

Given the alphabets

$$\mathrm{T} = \{\mathrm{int}^0\},\ \ \mathrm{F} = \{0 : \mathrm{int}\},\ \ \mathrm{P} = \{\}$$

Consider the trivial predicate p defined by

$$\begin{aligned} \mathrm{p(X}, 0) \quad &\leftarrow \\ p(X, Y) \quad &\leftarrow \quad p(0, 0). \end{aligned}$$

From the first clause, assuming that the type of the variable X is α, the reconstructed type is $p : Pred(\alpha \times int)$.

The second clause has a recursive call to p. If we assume that the types of occurrence of p in the head and the body are polymorphic instances of the type of p, then from the second clause, the reconstructed type would be $p :\ Pred(\gamma \times \beta)$. By unifying the types reconstructed from the two clauses of p, we would get the most general type signature $p : Pred(\alpha \times int)$.

However, this means that p would be used polymorphically within its own definition. As mentioned before, the type reconstruction problem becomes undecidable [17]. Consequently, the reconstruction algorithm assumes that the type of the recursive occurrence of p is *identical* to its type signature. Thus,

from the second clause, the algorithm reconstructs the type $p : Pred(int \times int)$. By unifying the types reconstructed from the two clauses, the reconstructed type signature of p is $p : Pred(int \times int)$ which is not the most general type signature.

□

Example 5

Given the alphabets

$$\mathrm{T} = \{\mathrm{list}^1, \mathrm{int}^0, \ldots\}$$
$$F = \{nil : list(\alpha),\ 0 : int, 1 : int, \text{“.”} : \alpha \times list(\alpha) \rightarrow list(\alpha)\}$$
$$P = \{sum : Pred(int \times int \times int)\}$$

Consider the following program:

$$length(nil, 0).$$

$$length(X.L, N) \leftarrow length(L, N1), sum(N1, 1, N).$$

The type reconstruction algorithm reconstructs the type of *length* as follows:

From the first clause, the type $length : Pred(list(\alpha) \times int)$ is computed. (since $nil : list(\alpha)$ and $0 : int$ are in F). From the second clause, both the defining and the using occurrence of *length* are assumed to have the same type. Thus, for both occurrences, the type $length : Pred(list(\alpha) \times int)$ is computed. This follows from *sum* having a declared type $Pred(int \times int \times int)$ and $X.L$ having the type $list(\alpha)$ (since “.” : $\alpha \times list(\alpha) \rightarrow list(\alpha)$). Finally, the type of *length* is reconstructed to be $Pred(list(\alpha) \times int)$ by unifying the types of the above definitions of *length*.

□

Example 6

Given the alphabets

$$\mathrm{T} = \{\mathrm{char}^0, \mathrm{int}^0\}$$
$$F = \{a : char, 5 : int\}$$
$$P = \{\}$$

Consider the following program:

$$r(X) \leftarrow p(X, Y).$$

$$t(X) \leftarrow p(a, X).$$

$$p(X, Y) \leftarrow q(X), r(X).$$

$$q(X) \leftarrow p(X, 5).$$

In this program, p, q and r are mutually recursively defined. Further, t derives p and is not mutually recursive to p. Thus, the clauses are reordered so that the clauses defining p (and the clauses defining q, r which are mutually recursive to p) appear before the clause defining t. The reordered program is as follows:

$$r(X) \leftarrow p(X, Y).$$

$$p(X, Y) \leftarrow q(X), r(X).$$

$$q(X) \leftarrow p(X, 5).$$

$$t(X) \leftarrow p(a, X).$$

Note that the relative order in which the definitions for p, q and r appear is not relevant. The only restriction is that they must appear together. The type reconstruction algorithm reconstructs the types of p, q, r and t as follows:

> Since, p, q and r are mutually recursively defined, their types are reconstructed simultaneously by processing the first three clauses. From the first clause the type of r must be identical to the type of the first argument of p. From the second clause, the type of q and r must be identical to the type of the first argument of p. From the third clause the type of q must be identical to the type of the first argument of p. Further, the type of the body occurrence of p is $p : pred(\alpha \times int)$ and is assumed to be identical to the reconstructed type of p. Thus, the reconstructed types are as follows: $p : pred(\alpha \times int), q : pred(\alpha), r : pred(\alpha)$.
>
> Finally, from the last clause, the type of p is $p : pred(char \times \beta)$ which is a polymorphic instance of the reconstructed type for p rather than being identical to the reconstructed type of p. This is because t is NOT mutually recursive to p. The type of t is an instance of the type of the second argument of p (which was reconstructed to be int). Thus, the type of t is reconstructed to be $t : pred(int)$.

Note that if the third clause in the original program was replaced by the following clause

$$p(X, Y) \leftarrow q(X), r(X), t(Y).$$

then t would be mutually recursive to p. The reordering as specified above would still be valid. Consequently, from the last clause the body type of p would be $p : pred(char \times \beta)$ which would have to be identical to the reconstructed type of p. Thus, the reconstructed types would be $p : pred(char \times int), q : pred(char), r : pred(char), t : pred(int)$.

Note also that if the third clause in the original program was replaced by the following clause

$$p(X, Y) \leftarrow q(X), r(Y).$$

then the reconstructed type for p would be $p : pred(int \times int)$ since the type of r is identical to the type of the second argument of p (from the first clause) and also identical to the type of the first argument of p (from the second clause). The last clause would result in a type error since the first argument to p is not of type int. Thus, the algorithm would print an error message that means types cannot be reconstructed.

□

5 Type Consistency

Type consistency is the property that evaluation rules transform a well-formed expression into another well-formed expression. This property is also referred to as the "subject reduction" property in [1]. For Typed Prolog, since SLD-resolution is the evaluation rule we show that one step of SLD-resolution transforms a well-formed goal expression into another well-formed goal expression.

More precisely, let Γ be a type environment such that $\Gamma \vdash \phi$ Formula. If ϕ' is obtained by one step of SLD-resolution, then there is some extension Γ' of Γ such that $\Gamma' \vdash \phi'$ Formula. This follows from the fact that the types of predicate occurrences to the left of $\leftarrow$ are equivalent to the assigned type signatures (definitional genericity) while the types of the predicate occurrences in the goal formula are instances of the assigned type signatures.

The type system for Typed Prolog satisfies the definitional genericity condition and function symbols are type preserving. It has been shown in [10] that the above conditions imply that untyped SLD-resolution is type-consistent. Consequently, type checking at run-time is obviated. Type-consistency has been shown for the Mycroft-O'Keefe type system in [10].

6 Extensions

The type system for Typed Prolog permits the incorporation of many logical as well as nonlogical constructs in a straightforward manner. For instance, negation, disjunction, cuts, asserts and retracts can all be incorporated into the type system. The syntax of formulas changes to

$$\phi ::= \epsilon \mid ! \mid A \mid not(A) \mid assert(A) \mid retract(A) \mid \phi_1, \phi_2 \mid \phi_1; \phi_2$$

The type rules for formulas are simple restatements of the modified syntax; e.g., the type rule for $not(A)$ is

$$\frac{\Gamma \vdash A \text{ Atom}}{\Gamma \vdash not(A) \text{ Formula}}$$

It must be noted that certain other constructs such as the predicate $functor(T, F, N)$ cannot be incorporated within the present framework for Typed Prolog. For example the clause $A \leftarrow functor(f(a, X), f, 2)$ requires function types for the term f. The type system needs to be enriched with higher order types in order to facilitate the above extension.

If subtyping is allowed in the type system, then the type consistency property is violated [5]. As an example, consider the following Typed Prolog program:

Example 7

Given the alphabets

$$T = \{man^0, woman^0, person^0\}$$
$$F = \{john : man,\ mary : woman\}$$
$$P = \{married : Pred(man \times woman), spouse : Pred(person \times person)\}$$

If we assume that man and $woman$ are both subtypes of $person$ then the following is a well-typed program:

$$\begin{array}{lll} & married(john, mary) & \\ [\forall X : person, Y : person] & & \\ & spouse(X, Y) & \leftarrow \quad married(X, Y) \\ [\forall X : person, Y : person] & & \\ & spouse(X, Y) & \leftarrow \quad married(Y, X) \end{array}$$

However, with the well-typed goal $\leftarrow spouse(john, john)$, one step of SLD resolution yields an ill-typed subgoal $\leftarrow married(john, john)$. Thus, for the type system with subtyping, untyped SLD evaluation is *not* type consistent. This problem can be solved by requiring unification to take types into account and compute new types for atom unifications [9]. A different approach [13] requires runtime type checking to detect such a type inconsistency. An alternative to this approach, adopted in [5], uses the information about the dataflow within the clauses to obviate runtime type checking. We hope to investigate subtyping issues in the future.

7 Conclusions

This paper presents Typed Prolog, a Prolog-like polymorphic logic programming language. A prescriptive type system for Typed Prolog is presented as a semantic reconstruction of the Mycroft-O'Keefe type system [27]. Type declarations are an integral part of the language and are used to determine the semantics of programs. Typed Prolog follows a conventional notion of types that is commonly found in programming languages and logic. A typed semantics is defined for Typed Prolog.

Type inference is made feasible by a type reconstruction algorithm that reconstructs types of predicates given the types of functions. The type system is quite robust in allowing the addition of a variety of extra-logical features to Typed Prolog. We view this prescriptive method as the most pragmatic approach for incorporating types into logic programs.

There are several directions for further research in this area. We hope to investigate issues like the usefulness of "regular trees" [11, 25, 26, 28, 30] in expressing function types within this prescriptively typed framework. We would also like to consider the notion of types within the constraint logic programming framework [15]. Another area of interest is the utilization of order-sorted types [8, 9, 32] to achieve subtyping.

References

[1] H. P. Barendregt. Functional Programming and Lambda Calculus. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 7, pages 321 – 364, The MIT Press/Elsevier, 1990.

[2] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. HOPE: An experimental applicative language. In *ACM LISP Conf.*, pages 136–143, 1980.

[3] A. D. Burt, P. M. Hill, and J. W. Lloyd. *Preliminary Report on the Logic Programming Language Godel.* Technical Report TR-90-02, University of Bristol, October 1990.

[4] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293 – 322, Plenum Press, New York, 1978.

[5] R. Deitrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proc. 2nd European Symp. on Programming*, pages 79 – 93, LNCS, Springer Lecture Notes in Computer Science, 1988.

[6] H. B. Enderton. *A Mathematical Introduction to Logic.* Associated Press, New York, 1972.

[7] J.-Y. Girard. *Interpretation Fonctionelle et Elimination des Coupures Dans L'arithemetique D'order Superieur.* PhD thesis, University of Paris, 1972.

[8] J. A. Goguen. *Order Sorted Algebra.* Semantics and Theory of Computation Report 14, Computer Science Dept., UCLA, 1978.

[9] J. A. Goguen and J. Meseguer. Models and Equality for Logic Programming. In *TAPSOFT'87*, pages 1 – 22, LNCS 250, Springer Lecture Notes in Computer Science, Pisa, Italy, 1987.

[10] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In J. Diaz and F. Orejas, editors, *TAPSOFT 89*, pages 225 – 240, LNCS 352, Springer Lecture Notes in Computer Science, 1989.

[11] N. Heintze and J. Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In *Proc. Annual ACM Symp. on Principles of Programming Languages*, pages 197 – 209, ACM, Jan 1990.

[12] F. Henglein. *Polymorphic Type Inference and Semi-Unification.* Technical Report 443, New York University, May 1989.

[13] P. M. Hill and R. W. Topor. *A semantics for typed logic programs.* Technical Report TR-90-11, University of Bristol, May 1990.

[14] P. Hudak and P. Wadler. *Report on the Functional Programming Language* Haskell. Technical Report YALEU/DCS/RR-666, Dept. of CS, Yale University, Dec 1988.

[15] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. Annual ACM Symp. on Principles of Programming Languages*, pages 111 – 119, Munich, January 1987.

[16] T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. In *Intl. Joint Conf. on Artificial Intelligence*, pages 704 – 707, 1985.

[17] A. J. Kfoury, J. Tiruyn, and P. Urzyczyn. *The Undecidability of the Semi-Unification Problem.* Technical Report BUCS 89-010, Boston University, Dept. of CS, October 1989.

[18] M. Kifer and J. Wu. A First order theory of Types and Polymorphism in Logic Programming. Sept 1991. To appear in Proc. Symp. Logic in Computer Science.

[19] F. Kluzniak. Type synthesis for ground Prolog. In *Proc. Intl. Conf. on Logic Programming*, pages 788 – 816, Melbourne, 1987.

[20] R. A. Kowalski and M. H. van Emden. The semantics of predicate logic as a programming language. *Journal of ACM*, 23(4):733 – 742, 1976.

[21] T. K. Lakshman and U. S. Reddy. Typed Prolog: A Semantic Reconstruction of the Mycroft-O'Keefe Type System. February 1991. Extended Technical Report available via anonymous ftp from a.cs.uiuc.edu.

[22] J. W. Lloyd. *Foundations of Logic Programming, 2ed.* Springer Verlag, 1987.

[23] D. A. Miller and G. Nadathur. Higher-order logic programming. In *Proc. Intl. Conf. on Logic Programming*, pages 448 – 462, London, 1986.

[24] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17():348 – 375, 1978.

[25] P. Mishra. Towards a theory of types in Prolog. In *IEEE International symposium on logic programming*, pages 289 – 298, 1984.

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

[27] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. In *Artificial Intelligence*, pages 295 – 307, 1984.

[28] C. Pyo and U. S. Reddy. Inference of Polymorphic types for logic programs. In E. L. Lusk and R.A. Overbeek, editor, *Logic Programming: Proceedings of the North American Conf.*, pages 1115 – 1134, MIT Press, 1989.

[29] U. S. Reddy. A perspective on types for logic programs. In *the workshop on Types in Logic Programs at NACLP*, MIT Press, 1989. To appear in *Types in Logic Programs*, Frank Pfenning, (ed.) MIT Press, 1991.

[30] U. S. Reddy. Types for logic programs (abstract). In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference*, pages 836–840, MIT Press, Cambridge, Mass., 1990.

[31] J. C. Reynolds. Towards a theory of type structure. In *Coll. sur la Programmation*, pages 408–425, Springer Verlag, 1974.

[32] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Universität Kaiserslautern, W. Germany, May 1989.

[33] A. Tarski. A lattice theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5():285 – 309, 1955.

[34] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Conf. on Functional Programming languages and Computer Architecture*, pages 1 – 16, Springer Verlag, 1985.

[35] J. Xu and D. S. Warren. A type inference system for Prolog. In *Proc. Intl. Conf. on Logic Programming*, pages 604 – 619, 1988.

[36] E. Yardeni and E. Shapiro. A type system for logic programs. In E. Shapiro, editor, *Concurrent Prolog Vol 2*, pages 211 – 244, MIT Press, 1987.

[37] E. Yardeni, E. Shapiro, and T. Fruehwirth. Logic programs as types for logic programs. 1991. To appear in Proc. Symp. Logic in Computer Science.

[38] J. Zobel. Derivation of polymorphic types for Prolog programs. In *Proc. Intl. Conf. on Logic Programming*, pages 817 – 838, MIT Press, 1987.

Appendix

A The type-checking/ type-reconstruction procedure

This section describes the main procedure. The complete Prolog code can be obtained by an anonymous ftp to a.cs.uiuc.edu and retrieving the Types.tar.Z file from the pub/reddy/typed-prolog directory.

```
% TYPE-CHECK: Completely Typed Program P     --> YES/NO (Is P well-formed?)
% TYPE-RECONSTRUCT: Partially Typed Program  --> Completely Typed
%                                                well-formed Program
% ------------------------------------------------------------------------
% well_formed/6(Term,Type,Context,TypeDecl,FuncDecl,PredDecl).
% checks to see if Term is a well formed TYPED-PROLOG term
% of type Type in a Context with declarations TypeDecl,FuncDecl and PredDecl.
% ------------------------------------------------------------------------
% 1. Variable
well_formed(variable(X),Type,Context,_,_,_) :-
        var_assoc(variable(X),Type,Context).

% 2. Term
well_formed(T,Type,Context,Td,Fd,Pd) :-
        T =.. [F|Args],
        assoc(F,Ftype,Fd),
        rename_var(Ftype,fun(ArgTypes,Type)), % computed types
                % can be different instances
                % of the declared type => POLYMORPHISM
        well_formed_list(Args,ArgTypes,Context,Td,Fd,Pd).

% well_formed/8(Term,Type,Context,TypeDecl,FuncDecl,PredDecl,Call_table,H)
% Call_table is the call graph for the program that is used to check
% whether atom is recursively defined.
```

```
% H is the head of the rule from which the atom was derived..

% 3. Atom (body) RECURSIVE
well_formed(A,atom,Context,Td,Fd,Pd,Call_table,H) :-
        A =.. [P|Args],
        in_table(P,Call_table,L), % P and H are MUTUALLY RECURSIVE
        occurs_in(H,L),                 %
        assoc(P,pred(ArgTypes),Pd),
                % body type == declared type
                % No renaming!
        well_formed_list(Args,ArgTypes,Context,Td,Fd,Pd).

% 3. Atom (body) NON-RECURSIVE
well_formed(A,atom,Context,Td,Fd,Pd,_Call_table,_H) :-
        A =.. [P|Args],
        assoc(P,Ptype,Pd), % Type of P had better have been computed
        rename_var(Ptype,pred(ArgTypes)), % computed body types
                % can be  INSTANCES
                % of the declared type => POLYMORPHISM
        well_formed_list(Args,ArgTypes,Context,Td,Fd,Pd).

% 3'. Atom (head)
well_formed_generic(A,atom,Context,Td,Fd,Pd) :-
        A =.. [P|Args],
        assoc(P,pred(ArgTypes),Pd),
        % don't use rename_var(Ptype,Ptype1) since we want
        % M.G.DECLARED and Computed Types to be same and NOT instance
        well_formed_list(Args, ArgTypes, Context,Td,Fd,Pd).

% 4. Formula
well_formed(true,formula,_Context,_Td,_Fd,_Pd,_Call_table,_H).
```

```
% non-logical features like assert, retract, cut, not.
well_formed(!,formula,Context,Td,Fd,Pd,Call_table,H).

well_formed(not(A),formula,Context,Td,Fd,Pd,Call_table,H) :-
        well_formed(A,atom,Context,Td,Fd,Pd,Call_table,H).

well_formed(assert(A),formula,Context,Td,Fd,Pd,Call_table,H) :-
        well_formed(A,clause,Context,Td,Fd,Pd,Call_table).

well_formed(retract(A),formula,Context,Td,Fd,Pd,Call_table,H) :-
        well_formed(A,clause,Context,Td,Fd,Pd,Call_table).

well_formed((F1,F2),formula,Context,Td,Fd,Pd,Call_table,H) :-
        well_formed(F1,formula,Context,Td,Fd,Pd,Call_table,H),
        well_formed(F2,formula,Context,Td,Fd,Pd,Call_table,H).

well_formed(=(T1,T2),formula,Context,Td,Fd,Pd,_Call_table,_H) :-
        well_formed(T1,Type,Context,Td,Fd,Pd),
        well_formed(T2,Type,Context,Td,Fd,Pd).

well_formed(F,formula,Context,Td,Fd,Pd,Call_table,H) :-
        well_formed(F,atom,Context,Td,Fd,Pd,Call_table,H).


% 5. Clause
well_formed(:-(Head,Body),clause,Context,Td,Fd,Pd,Call_table) :-
        well_formed_generic(Head,atom,Context,Td,Fd,Pd),
        well_formed(Body,formula,Context,Td,Fd,Pd,Call_table,Head).

well_formed(Head,clause,Context,Td,Fd,Pd,Call_table) :-
```

```
        well_formed(:-(Head,true), clause,Context,Td,Fd,Pd,Call_table).


% 6. Program
well_formed_program(Program,program,_,Td,Fd,Pd) :-

% a. Compute Call_table =  call_list for each predicate that is defined.
% used to handle mutually recursive definitions
        mk_call_table(Program, Call_table, Program),

% b. Compute the Adj_list = adjacency list for each predicate that is defined
% Used to topologically sort the program
        mk_adj_list(Program,Adj_list),
% c. re-order the program clauses so that well_formed_recon can reconstruct
% a predicate type before its "use".
        order_program(Adj_list,Program,Ord_Prog),

% d. Re-construct (if reqd) the predicate types,
% check each predicate defn for well-formedness
        well_formed_recon(Ord_Prog,program,_,Td,Fd,Pd, Call_table).


% ------------------------------------------------------------------
% reconstructs type of preds (if reqd)
% check each clause for well-formedness
% well_formed_recon/7(Term,Type,Context,TypeDecl,FuncDecl,PredDecl,Call_table)

well_formed_recon([],program,_,_Td,_Fd,_Pd, _Call_table).

well_formed_recon([C|Cl],program,_,Td,Fd,Pd, Call_table) :-
        well_formed(C,clause,_,Td,Fd,Pd,Call_table),
```

```
        well_formed_recon(Cl,program,_,Td,Fd,Pd,Call_table).

% ------------------------------------------------------------------------
% List of terms
well_formed_list([],_,_,_,_,_).

well_formed_list([H|L],[TH|TL],Context,Td,Fd,Pd) :-
        well_formed(H,TH,Context,Td,Fd,Pd),
        well_formed_list(L,TL,Context,Td,Fd,Pd).
% ------------------------------------------------------------------------
```

Correctness of the procedure

Firstly, it is easy to see that the clauses in the procedure implement the well-formedness rules. Corresponding to each well-formedness rule is a clause whose head corresponds to the consequent of the rule and whose body corresponds to the antecedent of the rule.

Secondly, for a Typed Prolog program, if the type signatures for all function symbols are declared, then for *non-recursively defined* predicates, the type reconstruction procedure reconstructs the most general type signatures. This is because while reconstructing the type of a predicate, the reordering of clauses ensures that the most general types of all (non-recursive) body atoms occurring in the predicate definition have already been reconstructed. These types are used to reconstruct the most general type signature for the predicate symbol in the head.