In *Proc. TAPSOFT'89, Barcelona, March 1989*, Springer LNCS 352, pp. 225–240

Horn Clause Programs with Polymorphic Types: Semantics and Resolution

Michael Hanus
Fachbereich Informatik, Universität Dortmund
D-4600 Dortmund 50, W. Germany
(uucp: michael@unidoi5)

This paper presents a Horn clause logic where functions and predicates are declared with polymorphic types. Types are parameterized with type variables. This leads to an ML-like polymorphic type system. A type declaration of a function or predicate restricts the possible use of this function or predicate so that only certain terms are allowed to be arguments for this function or predicate. The semantic models for polymorphic Horn clause programs are defined and a resolution method for this kind of logic programs is given. It will be shown that several optimizations in the resolution method are possible for specific kinds of programs. Moreover, it is shown that higher-order programming techniques can be applied in our framework.

1 Introduction

The theoretical foundation of the logic programming language Prolog is Horn clause logic. In this logic the basic objects (terms) are not classified: Each function and predicate may have any term as an argument [Llo87]. This point of view is not justified for the logic programming language Prolog: Several predefined predicates have restrictions on their arguments (e.g., *is* or *name*). Additionally, programs are frequently constructed from data types. In application programs only certain terms are allowed to be arguments for a function or predicate. It is impossible to express these restrictions in a natural way in Prolog. Types for logic programming can help to close the gap between theory and programming practice. Moreover, programming errors in Prolog are frequently type errors; in many typed languages such programming errors can be found at compile time.

In addition, programs of typed logic programming languages may be more efficient than programs of an untyped language. For instance, we want to define the predicate `append` that is satisfied iff the three arguments are lists and the third list is the concatenation of the first and the second. The following classical solution is wrong from a typing point of view:

```
append([],L,L) ←
append([E|R],L,[E|RL]) ← append(R,L,RL)
```

By this definition, the goal `append([],3,3)` is provable in contrast to our intuition. A correct definition is:

```
append([],[],[]) ←
append([],[E|R],[E|R]) ← append([],R,R)
append([E|R],L,[E|RL]) ← append(R,L,RL)
```

If the first and second argument of an `append`-literal are non-empty lists, a proof with the second definition needs more steps than a proof with the first one. In a typed logic language the first definition could be already correct.

Many authors have investigated types in logic programming languages. There are two principal starting points in research:

The *declarative approach*: The programmer has to declare all types he wants to use and the types of all functions and predicates in the program. These proposals have a formal semantics of the notion of type, e.g., types represent subsets of carrier sets of interpretations. Goguen, Meseguer [GM86] and Smolka [Smo86] have proposed ordered sorted type systems for Horn clause logic (with equality). Each type represents a subset of the carrier set in the interpretation, and the order of types implies a subset relation on the corresponding sets. Aït-Kaci and Nasr [AN86] have proposed a logic language with subtypes and inheritence based on a similar semantics. From an operational point of view, these approaches require a unification procedure that takes account of types, i.e., types are present at run-time.

The *operational approach*: The aim of these type systems is to ensure that predicates are only called with appropriate arguments at run time. This should be achieved by a static analysis of the program. A lot of these approaches do not require any type declarations but the types will be inferred by a type checker. These approaches have only a syntactic notion of type. Mishra [Mis84] and Zobel [Zob87] have presented type inference systems for detecting programming errors in a given Prolog program. Kanamori, Horiuchi [KH85] and Kluźniak [Klu87] have developed algorithms for inferring types of variables in a Prolog program. Yardeni and Shapiro [YS87] have presented a type-checking algorithm where types are regular sets of ground atoms.

We are interested in a *polymorphic type system* where type declarations may contain type variables that are universally quantified over all types [DM82]. Mycroft and O'Keefe [MO84] have investigated such a type system for Prolog. In their proposal, the programmer has to declare the types of functions and predicates, but it is not a declarative approach because they have no semantic notion of a type. They have put restrictions on the use of polymorphic types in function declarations and clauses. Their programs can be executed without dynamic type checking. Dietrich and Hagl [DH88] have extended this type system to subtypes on the basis of mode declarations for the predicates. They have also only a syntactic notion of a type. TEL [Smo88] is a logic language with functions and a polymorphic type system with subtypes. Since subtypes are included, there are several restrictions on the use of polymorphic types which prevents in particular the application of higher-order programming techniques.

This paper presents a declarative approach to a generalized polymorphic type system for Horn clause logic. The topics of this paper are:

- We present a rather general polymorphic type system: We do not restrict the use of types. In contrast to [MO84], any polymorphic type expression may be argument or result type of a function or predicate. No difference will be made in the typing of the head and the body of a clause.
- Our approach is declarative: The semantics of types is defined in a model-theoretic way in contrast to other type systems for Prolog where types are viewed as sets of ground terms.
- We present sound and complete deduction and resolution methods for our logic programs.
- Several optimizations of the resolution procedure are presented for specific subclasses of programs. We show that it is possible to translate polymorphic logic programs in our sense into untyped Horn clause programs. The type system and results of [MO84] will be a special case of our type system.
- Higher-order programming techniques can be applied in our framework. We present an interesting class of logic programs that are ill-typed in the sense of other polymorphic type systems for logic programming but are well-typed in our framework.

Let us start by looking at an example of a polymorphically typed Horn clause program in our sense. First the programmer has to specify the types that he wants to use in the clauses. There are basic types like *int* or *bool*, and type constructors that create new types from given types. E.g., the type constructor *list* with arity 1 creates from the type *int* the type of integer lists $list(int)$. Type expressions may contain type variables which are universally quantified over all types. In the following we use α, β for type variables. The type expression $list(\alpha)$ represents the types

$$list(int) \qquad list(bool) \qquad list(list(int)) \qquad \ldots$$

or, in general, a list of any type. Two functions are defined on any list: The constant function `[]` that represents the empty list, and the function $\bullet$ that concatenates an element with a list of the same type (throughout this paper we use the Prolog notation for lists [CM87]). The type declarations for these two functions are:

func `[]`: $\rightarrow\ list(\alpha)$
func $\bullet$: $\alpha,\ list(\alpha)\ \rightarrow\ list(\alpha)$

The predicate `append` has three arguments and is defined on lists of the same type. Therefore `append` has the following type declaration:

pred `append`: $list(\alpha),\ list(\alpha),\ list(\alpha)$

The following clauses define the semantics of `append` and are well-typed in our sense, if the variables `L`, `R` and `RL` are of type $list(\alpha)$ and the variable `E` is of type α:

```
append([],L,L) ←
append([E|R],L,[E|RL]) ← append(R,L,RL)
```

In our type system it is also possible to add the specialized clause

```
append([1,2],[3,4],[1,2,3,4]) ←
```

to the program. Note that the arguments of the head of this clause have types int and $list(int)$. Hence it is not a well-typed clause in the sense of [MO84] since the head of the clause has not the most general type. The application of this feature in order to use higher-order programming techniques and more examples are given in the rest of this paper. Detailed definitions and proofs of results can be found in [Han88b] and the author's dissertation.

2 Polymorphic logic programs

We use notions from algebraic specifications [GTW78] for the specification of types. A **signature** Σ is a pair (S, O), where S is a set of **sorts** and O is a family of **operator** sets of the form $O = (O_{w,s} | w \in S^*, s \in S)$. We write $o: s_1, \ldots, s_n \rightarrow s \in O$ instead of $o \in O_{(s_1,\ldots,s_n),s}$. An operator of the form $o: \rightarrow s$ is also called a **constant** of sort s. A signature $\Sigma = (S, O)$ is interpreted by a **Σ-algebra** $A = (S_A, O_A)$ which consists of an S-sorted domain $S_A = (S_{A,s} | s \in S)$ and an operation $o_A: S_{A,s_1}, \ldots, S_{A,s_n} \rightarrow S_{A,s} \in O_A$ for any $o: s_1, \ldots, s_n \rightarrow s \in O$. A set of **$\Sigma$-variables** is an S-sorted set $V = (V_s | s \in S)$. The set of **Σ-terms** of sort s with variables from V, denoted $T_{\Sigma,s}(V)$, is inductively defined by $x \in T_{\Sigma,s}(V)$ for all $x \in V_s$, $c \in T_{\Sigma,s}(V)$ for all $c: \rightarrow s \in O$, and $o(t_1, \ldots, t_n) \in T_{\Sigma,s}(V)$ for all $o: s_1, \ldots, s_n \rightarrow s \in O$ $(n > 0)$ and all $t_i \in T_{\Sigma,s_i}(V)$. We write $T_\Sigma(V)$ for all Σ-terms with variables from V and T_Σ for the set of **ground terms** $T_\Sigma(\emptyset)$. By $T_\Sigma(V)$ we also denote the term algebra.

A **variable assignment** is a mapping $a: V \rightarrow S_A$ with $a(x) \in S_{A,s}$ for all variables $x \in V_s$ (more precisely, it is a family of mappings $(a_s: V_s \rightarrow S_{A,s} | s \in S)$). A **$\Sigma$-homomorphism** from a Σ-algebra $A = (S_A, O_A)$ into a Σ-algebra $B = (S_B, O_B)$ is a mapping (family of mappings) $h: S_A \rightarrow S_B$ with the properties $h_s(c_A) = c_B$ for all $c: \rightarrow s \in O$ and $h_s(o_A(a_1, \ldots, a_n)) = o_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ for all $o: s_1, \ldots, s_n \rightarrow s \in O$ $(n > 0)$ and all $a_i \in S_{A,s_i}$.

Polymorphic types are represented by single-sorted signatures: $H = (Ty, Ht)$ is a **signature of types** if H is a signature with one sort $Ty = \{type\}$. Operators of the form $h: \rightarrow type$ are called **basic types** (with arity 0), whereas operators of the form $h: type^n \rightarrow type$ are called **type constructors** with arity $n > 0$. By X we denote a set of **type variables**. A **type expression** or (polymorphic) **type** is a term from $T_H(X)$, a **monomorphic type** is a term from T_H. Since we have only one sort in the signature of types, we will also use H to denote the set of type constructors Ht.

A **type substitution** σ is an H-homomorphism $\sigma{:}T_H(X) \to T_H(X)$. $\boldsymbol{TS(H,X)}$ denotes the class of all type substitutions. Two types $\tau, \tau' \in T_H(X)$ are called **equivalent** if there exists a bijective type substitution σ with $\sigma(\tau) = \tau'$.

A **polymorphic signature** Σ for logic programs is a triple $(\boldsymbol{H}, \boldsymbol{Func}, \boldsymbol{Pred})$ with:

- H is a signature of types with $T_H \neq \emptyset$
- $Func$ is a set of **function declarations** of the form $f{:}\tau_1, \ldots, \tau_n \to \tau$ with $\tau_i, \tau \in T_H(X)$, $n \geq 0$, where, in addition, $\tau_f = \tau'_f$ whenever $f{:}\tau_f, f{:}\tau'_f \in Func$.
- $Pred$ is a set of **predicate declarations** of the form $p{:}\tau_1, \ldots, \tau_n$ with $\tau_i \in T_H(X)$ $(n \geq 0)$, where, in addition, $\tau_p = \tau'_p$ whenever $p{:}\tau_p, p{:}\tau'_p \in Pred$.

The additional restrictions exclude overloading. With these restrictions it is possible to compute the most general type of a term. Therefore the user need not annotate terms in a clause with type expressions. Note that there are no restrictions on the use of type variables in function declarations in contrast to other polymorphic type systems for logic programming, e.g., [MO84], [Smo88].

The following specification of a polymorphic signature will be used in later examples. Declarations of basic types and type constructors, functions, and predicates are preceded by the keywords "type", "func" and "pred", respectively.

type nat/0, $list$/1, $pred2$/2
func z: $\to nat$
func s: $nat \to nat$
func []: $\to list(\alpha)$
func $\bullet$: $\alpha,\ list(\alpha) \to list(\alpha)$
func pred_inc: $\to pred2(nat, nat)$
pred inc : $nat,\ nat$
pred map : $pred2(\alpha, \beta),\ list(\alpha),\ list(\beta)$
pred apply2: $pred2(\alpha, \beta),\ \alpha,\ \beta$

The predicate `apply2` will be interpreted like `call` in Prolog: If the first argument has type $pred2(\alpha, \beta)$ and the next arguments have types α and β, then it is equivalent to the application of the first argument to the other two arguments. `pred_inc` is a constant of type $pred2(nat, nat)$. The equivalence of `apply2(pred_inc,...)` and `inc(...)` will be stated in a specific clause (see below).

In the rest of this paper we will assume that $\Sigma = (H, Func, Pred)$ is a polymorphic signature. The variables in a polymorphic logic program are not quantified over all objects, but vary only over objects of a particular type. Thus each variable is annotated with a type expression: If $\boldsymbol{Var}$ is an infinite set of variable names that are distinguishable from symbols in polymorphic signatures and type variables, the **set of typed variables** $Var_{\Sigma,X}$ is defined as $Var_{\Sigma,X} := \{x{:}\tau \mid x \in Var,\ \tau \in T_H(X)\}$. V is a **set of typed variables with unique types**, written $V \subseteq_U Var_{\Sigma,X}$, if $V \subseteq Var_{\Sigma,X}$ and $\tau = \tau'$ whenever $x{:}\tau, x{:}\tau' \in V$.

The notion of "typed variables with unique types" is not necessary for the definition of the semantics and the resolution procedure, but it is useful for optimization and detection of type errors at compile time. Hence we define the semantics for arbitrary sets of typed variables, whereas in polymorphic logic programs the clauses must have variables with unique types so that optimizations and type-checking are possible.

According to [Chu40], we embed types in terms, i.e., each symbol in a term is annotated with a type expression: Let $V \subseteq Var_{\Sigma,X}$. A **$(\Sigma, X, V)$-term of type** $\tau \in T_H(X)$ is either a **variable** $x{:}\tau \in V$, a **constant** $c{:}\tau$ with $c{:} \to \tau_c \in Func$ so that there exists a $\sigma \in TS(H,X)$ with $\sigma(\tau_c) = \tau$, or a **composite term** of the form $f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau$ $(n > 0)$ with $f{:}\tau_f \in Func$ so that there exists a type substitution $\sigma \in$

$TS(H,X)$ with $\sigma(\tau_f) = \tau_1, \ldots, \tau_n \to \tau$ and $t_i{:}\tau_i$ is a (Σ, X, V)-term of type τ_i $(i = 1, \ldots, n)$. $\boldsymbol{Term_\Sigma(X, V)}$ denotes the $T_H(X)$-sorted set of all (Σ, X, V)-terms. A **ground term** is a term from the set $Term_\Sigma(X, \emptyset)$.

Each occurrence of a variable in a term has the same type, whereas different occurrences of a function may have different types (polymorphism). We call terms from $Term_\Sigma(X, V)$ **well-typed terms**, whereas terms that have the same structure as well-typed terms but violate the type conditions are called **ill-typed terms**.

Examples: If we have the declarations

func `f`: $int, bool \to bool$
var `x`:α

then the term `f(x:`α`,x:`α`)`:$bool$ is ill-typed. If we have the additional declaration

func `id`: $\alpha \to \alpha$

then the term `f(id(2:`int`):`int`, id(true:`$bool$`):`$bool$`)`:$bool \in Term_{\Sigma,bool}(\emptyset, \emptyset)$ is a well-typed ground term.

The definition of the other syntactic constructs of polymorphic logic programs is straightforward: A (Σ, X, V)-**atom** has the form $p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)$, where $p{:}\tau_p \in Pred$ and there exists a type substitution $\sigma \in TS(H, X)$ with $\sigma(\tau_p) = \tau_1, \ldots, \tau_n$ and $t_i{:}\tau_i \in Term_\Sigma(X, V)$ $(i = 1, \ldots, n)$. A (Σ, X, V)-**goal** is a finite set of (Σ, X, V)-atoms. A (Σ, X, V)-**clause** is a pair (P, G), where P is a (Σ, X, V)-atom and G is a (Σ, X, V)-goal. If $G = \{A_1, \ldots, A_n\}$, we also write

$P \leftarrow A_1, \ldots, A_n$.

P is called **head** and G **body** of the clause. Note that again there are no restrictions on the use of types in clauses. A **Σ-term** (atom, goal, clause) is a (Σ, X, V)-term (atom, goal, clause) for some $V \subseteq Var_{\Sigma,X}$. In the following, if s is a syntactic construction (type, term, atom, ...), $tvar(s)$ and $var(s)$ will denote the set of type variables and typed variables that occur in s, respectively. Furthermore, we define $uvar(s) := \{x \mid \exists \tau \in T_H(X){:}\ x{:}\tau \in var(s)\}$ as the set of variable names that occur in s.

A **polymorphic logic program** or **polymorphic Horn clause program** $P = (\Sigma, C)$ consists of a polymorphic signature Σ and a set C of Σ-clauses, where $var(c) \subseteq_U Var_{\Sigma,X}$ for all $c \in C$. We require $var(c) \subseteq_U Var_{\Sigma,X}$ rather than $var(c) \subseteq Var_{\Sigma,X}$ because the user may omit the type annotations in the clauses of a polymophic logic program and the most general type of a term can be automatically computed under this assumption. Therefore we will omit the type annotations in the clauses of subsequent examples. We assume that the above polymorphic signature with predicate `map` is given. Then the following clauses define the semantics of the predicate `map`:

```
map(P,[],[]) ←
map(P,[E1|L1],[E2|L2]) ← apply2(P,E1,E2), map(P,L1,L2)
inc(N,s(N)) ←
apply2(pred_inc,N1,N2) ← inc(N1,N2)
```

Note that the last clause is not well-typed in the sense of [MO84] since `apply2` has the declared type "$pred2(\alpha,\beta), \alpha, \beta$" but is used in the clause head with the specialized type "$pred2(nat, nat), nat, nat$". This example illustrates the possibility of higher-order programming in our framework. That will be further investigated in section 8.

The next example is a program for the evaluation of Boolean terms. A Boolean term contains the constants `true` or `false`, the Boolean functions `and` and `or`, and the function `equal` to compare arbitrary terms of the same type. The evaluator is a predicate `isTrue` which is satisfied if such a term can be simplified to `true` by the common interpretation:

type $bool$/0
func `true` : $\to$ $bool$
func `false`: $\to$ $bool$
func `and`: $bool$, $bool$ $\to$ $bool$

```
func or:  bool, bool → bool
func equal:  α, α → bool
pred isTrue:  bool
clauses:
isTrue(true) ←
isTrue(and(B1,B2)) ← isTrue(B1), isTrue(B2)
isTrue(or(B1,B2)) ← isTrue(B1)
isTrue(or(B1,B2)) ← isTrue(B2)
isTrue(equal(T,T)) ←
```

Note that this program is well-typed in our sense but not a well-typed program in the sense of [MO84] because of the type of the function `equal`.

3 Semantics of polymorphic logic programs

We use algebraic structures for the interpretation of polymorphic logic programs [Poi86]. Variables in untyped logic vary over the carrier set of the interpretation. Consequently, type variables in polymorphic specifications vary over all types of the interpretation and typed variables vary over appropriate carrier sets. Hence an interpretation of a polymorphic logic program consists of an algebra for the signature of types and a structure for the derived polymorphic signature. A structure is an interpretation of types (elements of sort *type*) as sets, function symbols as operations on these sets and predicate symbols as predicates on these sets. We give an outline of the necessary notions.

If $H = (Ty, Ht)$ is a signature of types, an H-algebra $A = (Ty_A, Ht_A)$ is also called **H-type algebra**. The **polymorphic signature** $\Sigma(A) = (Ty_A, Func_A, Pred_A)$ **derived from Σ and A** is defined by

$$\begin{array}{rcl} Func_A & := & \{f{:}\sigma(\tau_f) \mid f{:}\tau_f \in Func,\ \sigma{:}\, X \to Ty_A \text{ is a type variable assignment}\} \\ Pred_A & := & \{p{:}\sigma(\tau_p) \mid p{:}\tau_p \in Pred,\ \sigma{:}\, X \to Ty_A \text{ is a type variable assignment}\} \end{array}$$

An **interpretation** of a polymorphic signature Σ is an H-type algebra $A = (Ty_A, Ht_A)$ together with a $\Sigma(A)$-structure(S, δ), which consists of a Ty_A-sorted set S (the **carrier** of the interpretation) and a denotation δ with:

1. If $f{:}\tau_1, \ldots, \tau_n \to \tau \in Func_A$, then $\delta_{f:\tau_1,\ldots,\tau_n\to\tau}{:}\ S_{\tau_1} \times \cdots \times S_{\tau_n} \ \to\ S_\tau$ is a function.

2. If $p{:}\tau_1, \ldots, \tau_n \in Pred_A$, then $\delta_{p:\tau_1,\ldots,\tau_n} \subseteq S_{\tau_1} \times \cdots \times S_{\tau_n}$ is a relation.

If A and A' are H-type algebras, then every H-homomorphism $\sigma{:}\, A \to A'$ induces a **signature morphism** $\sigma{:}\, \Sigma(A) \to \Sigma(A')$ and a **forgetful functor** $U_\sigma{:}\, Cat_{\Sigma(A')} \to Cat_{\Sigma(A)}$ from the category of $\Sigma(A')$-structures into the category of $\Sigma(A)$-structures (for details, see [EM85]). Therefore we can define a **Σ-homomorphism** from a Σ-interpretation (A, S, δ) into another Σ-interpretation (A', S', δ') as a pair (σ, h), where $\sigma{:}\, A \to A'$ is an H-homomorphism and $h{:}\, (S, \delta) \to U_\sigma((S', \delta'))$ is a $\Sigma(A)$-homomorphism. The class of all Σ-interpretations with the composition $(\sigma', h') \circ (\sigma, h) := (\sigma' \circ \sigma, U_\sigma(h') \circ h)$ of two Σ-homomorphisms is a category. Thus we call a Σ-interpretation (A, S, δ) **initial** iff for all Σ-interpretations (A', S', δ') there exists a unique Σ-homomorphism from (A, S, δ) into (A', S', δ').

The notion of "term interpretation" can be defined as usual (in the following, we assume that $V \subseteq Var_{\Sigma,X}$ is a set of typed variables). By $T_\Sigma(X, V)$ we denote the free term interpretation over X and V where the carrier is the $T_H(X)$-sorted set $Term_\Sigma(X, V)$. A homomorphism in the polymorphic framework consists of a mapping between type algebras and a mapping between appropriate structures. Consequently, a variable assignment in the polymorphic framework maps type variables into types and typed variables into objects of appropriate types: If $I = ((Ty_A, Ht_A), S, \delta)$ is a Σ-interpretation, then a **variable assignment** for (X, V) in I is a pair of mappings (μ, val) with $\mu{:}\, X \to Ty_A$ and $val{:}\, V \to S'$, where $(S', \delta') := U_\mu((S, \delta))$ and

$val(x{:}\tau) \in S'_\tau$ $(= S_{\mu(\tau)})$ for all $x{:}\tau \in V$. It can be shown that any variable assignment can be uniquely extended to a Σ-homomorphism. In the following we denote this Σ-homomorphism again by (μ, val).

We are not interested in all interpretations of a polymorphic signature but only in those interpretations that satisfies the clauses of a given polymorphic logic program. In order to formalize that we define the validity of atoms, goals and clauses relative to a given Σ-interpretation $I = (A, S, \delta)$:

- Let $v = (\mu, val)$ be an assignment for (X, V) in I.

  $\boldsymbol{I, v \models L}$ if $L = p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)$ is a (Σ, X, V)-atom with $(val_{\tau_1}(t_1{:}\tau_1), \ldots, val_{\tau_n}(t_n{:}\tau_n)) \in \delta'_{p:\tau_1,\ldots,\tau_n}$ where $U_\mu((S, \delta)) = (S', \delta')$

  $\boldsymbol{I, v \models G}$ if G is a (Σ, X, V)-goal with $I, v \models L$ for all $L \in G$

  $\boldsymbol{I, v \models L \leftarrow G}$ if $L \leftarrow G$ is a (Σ, X, V)-clause where $I, v \models G$ implies $I, v \models L$

- $\boldsymbol{I, V \models L}$ if L is a (Σ, X, V)-atom with $I, v \models L$ for all variable assignments v for (X, V) in I

  $\boldsymbol{I, V \models G}$ if G is a (Σ, X, V)-goal with $I, v \models G$ for all variable assignments v for (X, V) in I

  $\boldsymbol{I, V \models L \leftarrow G}$ if $L \leftarrow G$ is a (Σ, X, V)-clause with $I, v \models L \leftarrow G$ for all variable assignments v for (X, V) in I

We say "L is **valid in** I" if I is a Σ-interpretation with $I, var(L) \models L$ (analogously for goals and clauses). A Σ-interpretation I is called **model** for a polymorphic logic program (Σ, C) if $I, var(L \leftarrow G) \models L \leftarrow G$ for all clauses $L \leftarrow G \in C$. A (Σ, X, V)-goal G is called **valid in** (Σ, C) relative to V if $I, V \models G$ for every model I of (Σ, C). We shall write: $\boldsymbol{(\Sigma, C, V) \models G}$.

This notion of validity is the extension of validity in untyped Horn clause logic to the polymorphic case: In untyped Horn clause logic an atom, goal or clause is said to be true iff it is true for all variable assignments. In the polymorphic case an atom, goal or clause is said to be true iff it is true for all assignments of type variables and typed variables. The reason for the definition of validity relative to a set of variables is that carrier sets in our interpretations may be empty in contrast to untyped Horn logic. This is also the case in many-sorted logic [GM84]. Validity relative to variables is different from validity in the sense of untyped logic. The following example shows such a difference.

Example: Let $T_H = \{void, zero\}$, $Func = \{0{:} \rightarrow zero\}$, $Pred = \{\mathtt{p}{:}void, \mathtt{q}{:}zero\}$ and $\mathtt{x} \in Var$. If C consists of the clauses

$$\begin{array}{lll} \mathtt{p(x:}void\mathtt{)} & \leftarrow & \\ \mathtt{q(0:}zero\mathtt{)} & \leftarrow & \mathtt{p(x:}void\mathtt{)} \end{array}$$

then $M := ((\{void, zero\}, Ht), S, \delta)$ with $Ht_{void} = void$, $Ht_{zero} = zero$, $S_{void} = \emptyset$, $S_{zero} = \{0\}$, $\delta_{0:\rightarrow zero} = 0$ and $\delta_p = \delta_q = \emptyset$ is a model for (Σ, C). It can be shown that

$$(\Sigma, C, \{\mathtt{x:}void\}) \models \mathtt{q(0:}zero\mathtt{)}$$

Hence $\mathtt{q(0:}zero\mathtt{)}$ is valid in M relative to $\{\mathtt{x}{:}void\}$, but $\mathtt{q(0:}zero\mathtt{)}$ is not valid in M.

Validity in our sense is equivalent to validity in the sense of untyped logic if the types of the variables denotes non-empty sets in all interpretations. But a requirement for non-empty carrier sets is not reasonable. For a more detailed discussion of this subject compare [GM84].

"Typed substitutions" are a combination of type substitutions and substitutions on well-typed terms: If $V, V' \subseteq Var_{\Sigma,X}$ be sets of typed variables, then a **typed substitution** σ is a Σ-homomorphism $\sigma = (\sigma_X, \sigma_V)$ from $T_\Sigma(X, V)$ into $T_\Sigma(X, V')$. Since σ_X and σ_V are only applied to type expressions and typed terms, respectively, we omit the indices X and V and write σ for both σ_X and σ_V. We extend typed substitutions on Σ-atoms by: $\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$. $\boldsymbol{Sub(\Sigma, X, V, V')}$ denotes the class of all typed substitution from $T_\Sigma(X, V)$ into $T_\Sigma(X, V')$. A term $t' \in Term_\Sigma(X, V')$ is called an **instance** of a term

$t \in Term_{\Sigma}(X, V)$ if a typed substitution $\sigma \in Sub(\Sigma, X, V, V')$ exists with $t' = \sigma(t)$. The definition of instances can be extended on atoms, goals and clauses. We omit the simple definitions here. The next lemma shows the relationship between the validity of a clause and the validity of all its instances:

Lemma 1 *Let $I = (A, S, \delta)$ be a Σ-interpretation and $L \leftarrow G$ be a (Σ, X, V)-clause. Then:*

$$I, V \models L \leftarrow G \qquad \iff \qquad I, V' \models \sigma(L) \leftarrow \sigma(G) \text{ for all } \sigma \in Sub(\Sigma, X, V, V')$$

A **Herbrand model** for a polymorphic logic program (Σ, C) is a model where the carrier sets are ground terms with monomorphic types. Similarly to the untyped case it can be shown that the intersection of all Herbrand models is an initial model.

4 Deduction

This section presents an inference system for proving validity in polymorphic logic programs. In contrast to the untyped Horn clause calculus it is necessary to collect all variables used in a derivation of the inference system since validity depends on the types of variables. Let C be a set of Σ-clauses. The **polymorphic Horn clause calculus** contains the following inference rules:

1. **Axioms:** If $V \subseteq Var_{\Sigma,X}$ is a set of typed variables and $L \leftarrow G \in C$ is a (Σ, X, V)-clause, then $(\Sigma, C, V) \vdash L \leftarrow G$.

2. **Substitution rule:** If $(\Sigma, C, V) \vdash L \leftarrow G$ and $\sigma \in Sub(\Sigma, X, V, V')$, then $(\Sigma, C, V') \vdash \sigma(L) \leftarrow \sigma(G)$.

3. **Cut rule:** If $(\Sigma, C, V) \vdash L \leftarrow G \cup \{L'\}$ and $(\Sigma, C, V) \vdash L' \leftarrow G'$, then $(\Sigma, C, V) \vdash L \leftarrow G \cup G'$.

If the example program in section 3 on the previous page is given, then the following sequence is a deduction for $(\Sigma, C, \{\mathtt{x}{:}void\}) \vdash \mathtt{q(0}{:}zero\mathtt{)} \leftarrow$:

$$\begin{array}{llll}
(\Sigma, C, \{\mathtt{x}{:}void\}) \vdash & \mathtt{p(x}{:}void\mathtt{)} & \leftarrow & \\
(\Sigma, C, \{\mathtt{x}{:}void\}) \vdash & \mathtt{q(0}{:}zero\mathtt{)} & \leftarrow & \mathtt{p(x}{:}void\mathtt{)} \\
(\Sigma, C, \{\mathtt{x}{:}void\}) \vdash & \mathtt{q(0}{:}zero\mathtt{)} & \leftarrow &
\end{array}$$

This example shows the need for the explicit mentioning of the variables in the deduction since $(\Sigma, C, \emptyset) \models \mathtt{q(0}{:}zero\mathtt{)}$ is not true.

The following theorem states soundness and completeness of the polymorphic Horn clause calculus:

Theorem 2 *Let C be a set of Σ-clauses, $V \subseteq Var_{\Sigma,X}$ and L be a (Σ, X, V)-atom. Then:*

$$(\Sigma, C, V) \vdash L \leftarrow \qquad \iff \qquad (\Sigma, C, V) \models L$$

5 Unification

We are interested in a systematic method for proving validity of goals. The Horn clause calculus is one possibility, but in general it is far from being efficient. In untyped Horn clause logic the resolution principle [Rob65] with SLD-refutation [AvE82] is the basic proof method. The basic operation in a resolution step is the computation of a most general unifier of two terms. We need a similar operation for the resolution method in the polymorphic case. This section defines the unification in the polymorphic case and presents an algorithm for computing the most general unifier that is based on the method in [Lau86].
Example: The polymorphic signature contains the declarations $\mathtt{p}{:}\alpha \in Pred$, $\mathtt{q}{:}int \in Pred$ and $\mathtt{r}{:}\alpha \in Pred$ (α is a type variable). $\mathtt{X,Y,Z} \in Var$ are variable names and assume the following two clauses to be given:

$$
\begin{array}{lcl}
\texttt{p(X:}\mathit{int}\texttt{)} & \leftarrow & \texttt{q(X:}\mathit{int}\texttt{)} \\
\texttt{p(Y:}\alpha\texttt{)} & \leftarrow & \texttt{r(Y:}\alpha\texttt{)}
\end{array}
$$

The first clause is not allowed for proving the goal `p(Z:`*bool*`)`. We can use the second clause and have to prove in the next step the goal `r(Z:`*bool*`)`.
For proving the goal `p(Z:`*int*`)` the first clause can be used. In this case we are left with the goal `q(Z:`*int*`)` for the next resolution step.

As we see, unification of two atoms has to consider the types of the terms. Untyped unification cannot be applied in our case.

In section 3 *typed substitutions* were defined. The composition of two typed substitutions is again a typed substitution. Therefore we define the usual relations on typed substitutions:

- Let $V_1, V_2 \subseteq Var_{\Sigma,X}$ and $\sigma \in Sub(\Sigma, X, V, V_1)$ and $\sigma' \in Sub(\Sigma, X, V, V_2)$ be typed substitutions. σ is **more general** than σ', denoted $\sigma \leq \sigma'$, iff there exists $\phi \in Sub(\Sigma, X, V_1, V_2)$ with $\phi \circ \sigma = \sigma'$.

- Let t and t' be (Σ, X, V)-terms. t and t' are **unifiable** if there exists a typed substitution $\sigma \in Sub(\Sigma, X, V, V')$ with $\sigma(t) = \sigma(t')$ for a set $V' \subseteq Var_{\Sigma,X}$. In this case σ is called a **unifier** for t and t'. σ is a **most general unifier** (**mgu**) for t and t' if $\sigma \leq \sigma'$ for all unifiers σ' for t and t'.

The well-known algorithms for the unification of two terms in a term algebra (without equality) can be applied for the unification in the polymorphic case if we use a particular term algebra: The **untyped signature corresponding to** Σ, denoted $\boldsymbol{\Sigma}^u = (Term, Op)$, is defined as follows:

- $Term = \{term\}$

- $h{:}\underbrace{term, \ldots, term}_{n} \to term \in Op$ for all $h \in H$ with arity n $(n \geq 0)$

- $f{:}\underbrace{term, \ldots, term}_{n} \to term \in Op$ for all $f{:}\tau_1, \ldots, \tau_n \to \tau \in Func$ $(n \geq 0)$

- ':': $term, term \to term \in Op$

The signature Σ^u has only one sort *term*. If $V \subseteq Var$ is a set of variable names and X is a set of type variables, we interpret V and X also as variables of sort *term* and denote by $T_{\Sigma^u}(X \cup V)$ the algebra of Σ^u-terms with variables from $X \cup V$.

$T_{\Sigma^u}(X \cup V)$ is a single-sorted free term algebra over $X \cup V$, where the operation symbols are type constructors from H, function symbols from $Func$ and the symbol ':' with arity 2. It is $Term_\Sigma(X, V') \subseteq T_{\Sigma^u}(X \cup V)$ if $V = uvar(V')$, i.e., we can treat typed terms as terms over the signature Σ^u. For instance, the typed term `[]:`$list(\alpha)$ is also a term over Σ^u (actually, `':'([],`$list(\alpha)$`)` is a term over Σ^u, but we use the infix notation for the operator ':'). The converse is not true, because `equal(1:`*int*`,true:`*bool*`):`*bool* is a Σ^u-term, but not a Σ-term if `equal:`$\alpha, \alpha \to bool \in Func$.

The notions of "substitution" and "unifier" for the algebra $T_{\Sigma^u}(X \cup V)$ are defined as usual (e.g., [Llo87]) and we omit the details here. [Rob65] has found an algorithm for computing a most general unifier in a single-sorted free term algebra. For instance, a most general unifier in $T_{\Sigma^u}(X \cup \{\texttt{v}\})$ for the Σ-terms `[]:`$list(\alpha)$ and `v:`$list(int)$ is $\sigma(\alpha) = int$, $\sigma(\texttt{v}) =$ `[]`. It is an interesting fact that $\sigma' \in Sub(\Sigma, X, \{\texttt{v}{:}list(int)\}, \emptyset)$ with $\sigma'(\alpha) = int$ and $\sigma'(\texttt{v}{:}list(int)) =$ `[]:`$list(int)$ is a most general unifier for `[]:`$list(\alpha)$ and `v:`$list(int)$ in $Term_\Sigma(X, \{\texttt{v}{:}list(int)\})$. Generally, we can compute a most general unifier from a most general unifier in $T_{\Sigma^u}(X \cup V)$. The following theorem shows that the polymorphic unification problem can be reduced to the unification problem in $T_{\Sigma^u}(X \cup V)$.

Theorem 3 (Unification) *Let* $V \subseteq_U Var_{\Sigma,X}$ *and* $V_0 := uvar(V)$.

Two (Σ, X, V)*-terms are unifiable iff they are unifiable in* $T_{\Sigma^u}(X \cup V_0)$. *A most general unifier can be computed from a most general unifier in* $T_{\Sigma^u}(X \cup V_0)$.

Proof: If σ is a most general unifier in $T_{\Sigma^u}(X \cup V_0)$, then we define a typed substitution $\sigma' \in Sub(\Sigma, X, V, V')$ by $\sigma'(\alpha) = \alpha$ for all $\alpha \in X$ and $\sigma'(x{:}\tau) = \sigma(x){:}\sigma(\tau)$ for all $x{:}\tau \in V$. It can be proved by induction on the computation steps of the mgu-algorithm in [Rob65] that $\sigma(x){:}\sigma(\tau) \in Term_\Sigma(X, V)$. ■

The unification problem in the polymorphic case is solved by this theorem. There exist more efficient unification algorithms [MM82] [BC83] [PW78] that can also be used instead of the algorithm from [Rob65].

6 Resolution

The SLD-resolution in untyped Horn logic (see [Llo87]) can be used for polymorphic Horn clause programs if we replace the untyped unification by the polymorphic unification with typed substitutions as defined in the last section. "$(\Sigma, C, V) \models_{\mathrm{R}} \sigma\ G$" denotes a successful resolution ((Σ, C, V)-**refutation**) of the start goal G with the typed substitution σ as the computed answer, where (Σ, C) is the polymorphic logic program and V is the set of all typed variables used in the derivation. The soundness of resolution can be shown by simulating a resolution sequence by a derivation in the polymorphic Horn clause calculus:

Theorem 4 (Soundness of resolution) *Let (Σ, C) be a polymorphic logic program, $V \subseteq_U Var_{\Sigma,X}$ and G be a (Σ, X, V)-goal. If there exists a successful resolution $(\Sigma, C, V) \models_{\mathrm{R}} \sigma\ G$ with computed answer $\sigma \in Sub(\Sigma, X, V, V')$, then $(\Sigma, C, V') \models \sigma(G)$.*

Conversely, the completeness of resolution for polymorphic Horn clause logic can be shown by simulating each deduction in the polymorphic Horn clause calculus by resolution.

Theorem 5 (Completeness of resolution) *Let (Σ, C) be a polymorphic logic program, $V \subseteq_U Var_{\Sigma,X}$ be finite and G be a (Σ, X, V)-goal. If $\sigma \in Sub(\Sigma, X, V, V')$ is a typed substitution with $(\Sigma, C, V') \models \sigma(G)$, then there exist a set $V_0 \subseteq_U Var_{\Sigma,X}$ and a typed substitution $\sigma_0 \in Sub(\Sigma, X, V_0, V_1)$ with $(\Sigma, C, V_0) \models_{\mathrm{R}} \sigma_0\ G$ and there is a typed substitution $\phi \in Sub(\Sigma, X, V_1, V')$ with $\phi(\sigma_0(G)) = \sigma(G)$.*

The last two theorems are the justification for implementing the (Σ, C, V)-resolution as a proof method for polymorphic logic programs. For a complete resolution method, all possible derivations must be computed in parallel. If we use a backtracking method like Prolog, the resolution method becomes incomplete because of infinite derivations. If we accept this drawback, we can implement the resolution like Prolog with the difference that the unification includes the unification of type expressions.

7 Optimization

In the last two sections we have seen that the unification process in a resolution step has to unify the type expressions in every subterm. Thus the resolution is in any case more complex than the resolution in the untyped case. Mycroft and O'Keefe [MO84] have defined a specific class of polymorphic logic programs for which type checking is unnecessary at run-time. Therefore it is possible to disregard the type annotations in subterms at run-time if the polymorphic logic program has specific restrictions.

A first optimization for the resolution of polymorphic logic programs can be applied to a large class of functions: We call a function symbol f **type preserving** if $f{:}\tau_1, \ldots, \tau_n \to \tau \in Func$ and $tvar(\tau_i) \subseteq tvar(\tau)$ for $i = 1, \ldots, n$. In the declaration of a type preserving function all type variables occurring in the argument types also occur in the result type. For instance,

func `[]`: $\to\ list(\alpha)$
func $\bullet$: $\alpha,\ list(\alpha)\ \to\ list(\alpha)$

are type preserving functions, whereas

func `equal`: $\alpha,\ \alpha\ \to\ bool$

is not a type preserving function. We shall see that in the case of type preserving functions the type annotations in the arguments are unnecessary. If $t \in Term_\Sigma(X,V)$, we denote by $\Phi(t)$ the term obtained from t by deleting the type annotations in the arguments of type preserving functions. For instance, $\Phi(\bullet(1{:}int, []{:}list(int)){:}list(int)) = \bullet(1,[]){:}list(int)$ and $\Phi(\mathtt{equal}(1{:}int, 2{:}int){:}bool) = \mathtt{equal}(1{:}int, 2{:}int){:}bool$. Formally, Φ can be defined as a mapping $\Phi{:} T_{\Sigma^u}(X \cup V_0) \to T_{\Sigma^u}(X \cup V_0)$.

The mapping Φ is injective on $Term_\Sigma(X,V)$, i.e., for each $t' \in \Phi(Term_\Sigma(X,V))$ there exists a unique $t \in Term_\Sigma(X,V)$ with $\Phi(t) = t'$. Therefore it is sufficient to compute a unifier for $\Phi(t_0)$ and $\Phi(t_1)$ in $T_{\Sigma^u}(X \cup V_0)$ instead of computing a unifier for t_0 and t_1:

Theorem 6 (Optimized unification for type preserving functions)
Let $V \subseteq_U Var_{\Sigma,X}$, $V_0 := uvar(V)$ and $t_0, t_1 \in Term_\Sigma(X,V)$.
t_0 and t_1 are unifiable iff $\Phi(t_0)$ and $\Phi(t_1)$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$. A most general unifier for t_0 and t_1 can be computed from a most general unifier in $T_{\Sigma^u}(X \cup V_0)$.

The optimized unification can be extended on atoms if we interpret each predicate $p{:}\tau_1,\ldots,\tau_n \in Pred$ as a function symbol with declaration $p{:}\tau_1,\ldots,\tau_n \to bool$ and delete the result type *bool* in the unification. Therefore the optimized unification can be integrated in the resolution method defined in section 6. The theorem shows that type annotations are unnecessary for the unification of atoms if the signature is monomorphic, i.e., if all function and predicate declarations do not contain any type variables.

There is another possibility for optimization if a predicate is defined with *most general types*. For instance, if there is a declaration $g{:}\alpha,\beta \to bool$, then $g(X{:}\alpha, Y{:}\beta){:}bool$ is a term with most general type, but neither $g(X{:}\alpha, I{:}int){:}bool$ nor $g(X{:}\alpha, Z{:}\alpha){:}bool$ is a term with most general type. We omit the precise definitions here but call a predicate **type-generally defined** if in each clause for the predicate the head has a most general type and the predicates in the body are also type-generally defined. In a resolution of a type-generally defined predicate only other type-generally defined predicates occur. It can be shown that the unification of an atom with most general type and another atom with arbitrary types does not depend on the types (for details, see [Han88b]). Thus we obtain the following theorem:

Theorem 7 (Optimized unification for type-generally defined predicates) *Let (Σ, C) be a polymorphic logic program and the predicate p be type-generally defined in (Σ, C). Then type annotations are unnecessary during the resolution of a Σ-atom $p(t_1,\ldots,t_n)$.*

We may use the following algorithm to decide the property "most general type". The 'function' *skolemize* replaces all type variables in a type expression by 'new' type constants. With the use of *skolemize* equivalence of type expressions can be decided by unification of type expressions. In the algorithm, each type substitution σ is extended to a typed substitution by $\sigma(x{:}\tau) := x{:}\sigma(\tau)$. The algorithm must be called by $type\_general(t{:}\tau, \tau)$.

Algorithm *type_general*
Input: Term t, type ρ
Output: A type substition, if t is a term with most general type, and $fail$, otherwise.

1. $\rho' := skolemize(\rho)$
2. *If* $t = x{:}\tau \in Var_{\Sigma,X}$ *then* stop with $mgu(\tau, \rho')$
3. *If* $t = c{:}\tau$ with $c{:} \to \tau_c \in Func$ *then* stop with $mgu(\tau, \rho')$
4. *If* $t = f(t_1{:}\tau_1,\ldots,t_n{:}\tau_n){:}\tau$ *and* $f{:}\phi_1,\ldots,\phi_n \to \phi \in Func$ *and* $\sigma = mgu(\phi,\rho') \neq fail$ *then:*
   $\phi'_1,\ldots,\phi'_n \to \phi' := skolemize(\sigma(\phi_1,\ldots,\phi_n \to \phi))$
   If $mgu(\phi',\tau) = \sigma_0 \neq fail$ *and*
   $type\_general(\sigma_0(t_1{:}\tau_1), \phi'_1) = \sigma_1 \neq fail$ *and*

$\ldots$

$type\_general(\sigma_{n-1}(\ldots(\sigma_0(t_n{:}\tau_n))\ldots), \phi'_n) = \sigma_n \neq fail$

then stop with $\sigma_n \circ \cdots \circ \sigma_1 \circ \sigma_0$

else stop with $fail$

5. stop with $fail$

The next proposition shows that the polymorphic logic programs in the paper of [MO84] can be executed without dynamic type checking since their result holds only if each function is type preserving [Myc87].

Proposition 8 (Mycroft/O'Keefe-polymorphism) *Let* (Σ, C) *be a polymorphic logic program and* $V \subseteq_U Var_{\Sigma,X}$, *where* Σ *contains only type preserving functions.*

If $L = p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)$ *is a* (Σ, X, V)*-atom with* $p{:}\tau_p \in Pred$ *and* τ_p *and* $\tau_1, ..., \tau_n$ *are equivalent, then* L *is an atom with most general type.*

By this proposition, all predicates in a polymorphic logic program with the restrictions of [MO84] are type-generally defined, i.e., type annotations are unnecessary during the resolution of a Σ-goal by theorem 7. Therefore the type system of Mycroft/O'Keefe is a special case of our work because:

1. Every well-typed logic program in the sense of Mycroft/O'Keefe is a polymorphic logic program in our sense.

2. If we use the optimization techniques developed in this section, polymorphic logic programs in the sense of Mycroft/O'Keefe can be executed with the same efficiency as untyped Prolog programs.

On the other hand, our work is a proper extension of Mycroft/O'Keefe's type system because we have no restrictions on the use of polymorphic predicates in the heads of clauses, and we have no restrictions on the use of type variables in function types (compare examples in section 2). For instance, the predicate `isTrue` in the evaluator of Boolean terms is type-generally defined and therefore resolution can be done with the same efficiency as in an untyped program, but it is not a well-typed program in the sense of [MO84].

Mycroft and O'Keefe have proposed to extend polymorphic Horn clause programs by a family of predefined `apply` predicates to permit higher-order programming. But this extension is only necessary because of the restrictions in their type system. In our framework it is possible to simulate higher-order programming techniques without any conceptual extensions. This will be shown in the next section.

8 Higher-order programming

Many logic programming languages permit higher-order programming techniques, i.e., it is possible to treat predicates as first-class objects. For example, in Prolog the predicate `call` interprets the input term as a predicate call. Mycroft and O'Keefe [MO84] argue that for most practical purposes it is sufficient to have a predicate `apply` that takes something like a predicate name and a list of argument terms as input and that is satisfied if the corresponding predicate applied to the argument terms is provable. Hence they introduce a family of predefined predicates `apply` (one predicate for each arity) and a lambda notation for terms of predicate type, but they give only an informal definition of the meaning of `apply`.

Generally, a semantically clean amalgamation of higher-order predicates with logic programming techniques like unification is not trivial because the unification of higher-order terms is undecidable in general [Gol81]. Miller and Nadathur [MN86] have defined an extension of first-order Horn clause logic to include predicate and function variables based on the typed lambda calculus. For the operational semantics it is necessary to unify typed lambda expressions, which yields in a complex and semi-decidable unification [Hue75]. Hence they have a system with a clearly defined underlying logic, their proof procedure is sound and complete for goals without type variables, but the proof procedure is costly because of the unification of typed lambda expressions. Warren [War82] argues that no extension to Prolog or to the underlying first-order logic is necessary because the usual higher-order programming techniques can be simulated in first-order logic. Since he is concerned with Prolog and its untyped logic, he does not have a clear distinction between first-order and higher-order objects.

We suggest a 'middle road' approach to higher-order programming: To have an efficient operational semantics, we keep first-order logic as our theoretical framework. But we want to deal with higher-order objects in the sense of computing and distinguish between higher-order and first-order objects. Since we have an unrestricted mechanism of polymorphic types, we may integrate these higher-order programming techniques without any extensions to our concept of polymorphic logic programs (in contrast to [MO84]). This is demonstrated by the example of the `map` predicate in section 2. The predicate `map` takes a predicate of arity 2 and two lists as arguments and applies the argument predicate to corresponding elements of the lists. In order to specify the type of `map` it is necessary to introduce a type constructor $pred2$ of arity 2 that denotes the types of predicate expressions with two arguments. Hence the type of `map` is

pred `map:` $pred2(\alpha, \beta),\ \ list(\alpha),\ \ list(\beta)$

For each binary predicate p of type τ_1, τ_2 we introduce a corresponding constant $pred\_p$ of type $pred2(\tau_1, \tau_2)$. The relation between each predicate p and the constant $pred\_p$ is defined by clauses for the predicate `apply2`. Hence we get the example program of section 2. If we prove the goal

```
map(pred_inc,[z,s(s(z))],L)
```

by resolution, we get the answer substitution

```
L = [s(z),s(s(s(z)))]
```

(we omit the type annotations). The polymorphic logic program does not ensure that the constant `pred_inc` is interpreted as a relation in every model since we require only first-order structures as interpretations for polymorphic logic programs. But the clause for `apply2` with `pred_inc` as first argument ensures that in any model the constant `pred_inc` and the predicate `inc` are related together.

The `map` example has shown the possibility to deal with higher-order objects in our framework. It is also possible to permit lambda expressions, which can be translated into new identifiers and `apply` clauses for these identifiers (see [War82] for more discussion). If the underlying system implements indexing on the first arguments of predicates (as done in most compilers for Prolog, cf. [War83] and [Han88a]), then there is no essential loss of efficiency in our translation scheme for higher-order objects in comparison to a specific implementation of higher-order objects [War82].

The compilation of higher-order functions into first-order logic was also proposed by Bosco and Giovannetti [BG86], but they perform type-checking only for the source program and not for the target program. Clearly, the target program is not well-typed in the sense of [MO84] because of the clauses for the `apply` predicate (see above). Since we have translated higher-order objects into polymorphic logic programs, the use of higher-order objects is type secure in our framework. We have similar typing rules as in functional languages [DM82], and therefore functions and predicates have always appropriate arguments at run-time.

9 Implementation

The SLD-resolution in untyped Horn logic can be applied to polymorphic Horn clause programs if we use polymorphic unification to compute the most general unifier in a resolution step. Polymorphic unification can be reduced to untyped unification if we treat type expressions as terms and annotate each subterm with the corresponding type by the functor ':'. Hence we have implemented the resolution of polymorphic logic programs as a precompiler to a Prolog system: It takes a polymorphic logic program as input and produces a Prolog program as output. The clauses of the input program need not be annotated with types, because the precompiler computes the most general type of each clause by the type inference algorithm of [DM82].

Furthermore, the precompiler omits type annotations in the output program whenever it is possible by the techniques of section 7. For example, the precompiler translates the polymorphic logic program

type $list/1$, $pred2/2$
func `[]`: $\rightarrow$ $list(\alpha)$
func $\bullet$: α, $list(\alpha)$ $\rightarrow$ $list(\alpha)$
pred `append`: $list(\alpha)$, $list(\alpha)$, $list(\alpha)$

clauses:
`append([1,2], [3,4], [1,2,3,4])` $\leftarrow$
`append([], L, L)` $\leftarrow$
`append([E|R], L, [E|RL])` $\leftarrow$ `append(R, L, RL)`

(the type *int* of integer numbers is predefined) into the Prolog program

```
append(':'([1,2],list(int)), ':'([3,4],list(int)), ':'([1,2,3,4],list(int))).
append(':'([],list(A)), ':'(L,list(A)), ':'(L,list(A))).
append(':'([E|R],list(A)), ':'(L,list(A)), ':'([E|RL],list(A))) :-
   append(':'(R,list(A)), ':'(L,list(A)), ':'(RL,list(A)))
```

The program for the evaluation of Boolean terms (section 2) would be translated into a Prolog program where all type annotations are omitted. If there are type-generally defined predicates as well as other predicates in a polymorphic logic program, then type annotations must be deleted in argument terms before calling a type-generally defined predicate. After the predicate call type annotations must be added to the argument terms. Hence it may be more efficient not to omit type annotations in type-generally defined predicates in the presence of other predicates.

10 Conclusions

We have presented a polymorphic type system for Horn clause programs. Since we have a semantic notion of a type, this can help to close the gap between programming practice with Prolog and the underlying theory. The typing rules are quite simple: Each variable has a fixed type and each type instantiation of a polymorphic function or predicate can be used inside a clause if the result types of the argument terms are equal to the argument types. The semantics of polymorphic types is defined as a universal quantification over all possible types. We have shown that this semantics leads to similar results as in the untyped case: The Horn clause calculus can be extended to polymorphic logic programs, and the well-known resolution method for untyped Horn logic can also be used in the polymorphic case if the unification considers the types of terms. Hence our polymorphic logic programs are also related to “constraint logic programming” [JL87], where the consideration of types corresponds to constraints. We have also shown that the unification can disregard types if declarations and clauses have a particular form. In this case the proof method has the same efficiency as in the untyped case and we have shown that our type system is a proper extension of the type system in [MO84]. On the other hand, type information is useful to reduce the search space in the resolution process [SS85] [HV87]. Thus there are examples where the unification with types leads to a more efficient resolution than in the untyped case (see [Han88b]). In our type system it is allowed to have clauses where the left-hand side is not of the most general type. We have shown that this feature permits the use of higher-order programming techniques without breaking our type system.

Further work remains to be done. If the resolution process uses the standard Prolog left-to-right strategy, then further optimizations could be done to reduce the cases where type information is required for correct unification. If the modes of predicates are known, then there are further possibilities to omit type annotations [DH88]. The extension of our polymorphic type system to subtyping and inheritance would be useful. For practical applications the type system has to be extended to the meta-logical facilities of Prolog.

Acknowledgements

The author is grateful to Harald Ganzinger for his comments on a previous version of this paper.

References

[AN86] H. Aït-Kaci and R. Nasr. LOGIN: A Logic Programming Language with Built-In Inheritance. *Journal of Logic Programming (3)*, pp. 185–215, 1986.

[AvE82] K.R. Apt and M.H. van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, Vol. 29, No. 3, pp. 841–862, 1982.

[BC83] M. Bidoit and J. Corbin. A Rehabilitation of Robinson's Unification Algorithm. In *Proc. IFIP '83*, pp. 909–914. North-Holland, 1983.

[BG86] W. Bosco and E. Giovannetti. IDEAL: An Ideal Deductive Applicative Language. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 89–94, Salt Lake City, 1986.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, Vol. 5, pp. 56–68, 1940.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer, third rev. and ext. edition, 1987.

[DH88] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proc. ESOP 88, Nancy*, pp. 79–93. Springer LNCS 300, 1988.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th POPL*, pp. 207–212, 1982.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[GM84] J.A. Goguen and J. Meseguer. Completeness of Many-Sorted Equational Logic. Report No. CSLI-84-15, Stanford University, 1984.

[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*, pp. 295–363. Prentice Hall, 1986.

[Gol81] W. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science 13*, pp. 225–230, 1981.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pp. 80–149. Prentice Hall, Englewood Cliffs NJ, 1978.

[Han88a] M. Hanus. Formal Specification of a Prolog Compiler. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, Orléans, 1988. To appear in Springer LNCS.

[Han88b] M. Hanus. Horn Clause Programs with Polymorphic Types. Technical Report 248, FB Informatik, Univ. Dortmund, 1988.

[Hue75] G.P. Huet. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, Vol. 1, pp. 27–57, 1975.

[HV87] M. Huber and I. Varsek. Extended Prolog with Order-Sorted Resolution. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 34–43, San Francisco, 1987.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, Munich, 1987.

[KH85] T. Kanamori and K. Horiuchi. Type Inference in Prolog and Its Application. In *Proc. 9th IJCAI*, pp. 704–707. W. Kaufmann, 1985.

[Klu87] F. Kluźniak. Type Synthesis for Ground Prolog. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 788–816. MIT Press, 1987.

[Lau86] S. Launay. Complétion de systèmes de réécriture types dont les fonctions sont polymorphes (Thèse de 3ème cycle). Technical Report 86-5, C.N.R.S Université Paris VII, 1986.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[Mis84] P. Mishra. Towards a theory of types in Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 289–298, Atlantic City, 1984.

[MM82] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.

[MN86] D.A. Miller and G. Nadathur. Higher-Order Logic Programming. In *Proc. Third International Conference on Logic Programming (London)*, pp. 448–462. Springer LNCS 225, 1986.

[MO84] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, Vol. 23, pp. 295–307, 1984.

[Myc87] A. Mycroft. Private Communication, 1987.

[Poi86] A. Poigné. On Specifications, Theories, and Models with Higher Types. *Information and Control*, Vol. 68, No. 1-3, 1986.

[PW78] M.S. Paterson and M.N. Wegman. Linear Unification. *Journal of Computer and System Sciences*, Vol. 17, pp. 348–375, 1978.

[Rob65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, Vol. 12, No. 1, pp. 23–41, 1965.

[Smo86] G. Smolka. Order-Sorted Horn Logic: Semantics and Deduction. SEKI Report SR-86-17, FB Informatik, Univ. Kaiserslautern, 1986.

[Smo88] G. Smolka. TEL (Version 0.9) Report and User Manual. SEKI Report SR-87-11, FB Informatik, Univ. Kaiserslautern, 1988.

[SS85] M. Schmidt-Schauss. A Many Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation. In *Proc. 9th IJCAI.* W. Kaufmann, 1985.

[War82] D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.

[YS87] E. Yardeni and E. Shapiro. A Type System for Logic Programs. Technical Report CS87-05, The Weizmann Institute of Science, 1987.

[Zob87] J. Zobel. Derivation of Polymorphic Types for Prolog Programs. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 817–838. MIT Press, 1987.