In *Proc. of the 6th Int. Conf. on Logic Programming (Lisboa)*,
June 1989, pp. 382–397, MIT Press

Polymorphic Higher-Order Programming in Prolog

Michael Hanus

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

Abstract

Pure logic programming lacks some features known from other modern programming languages, e.g., type systems for detection of particular programming errors at compile time and higher-order facilities for treating programs as data objects and writing more compact programs. On the one hand there are several proposals for polymorphic type systems for logic programming, and on the other hand Warren [35] has shown that higher-order programming can be simulated in first-order logic. But the integration of these proposals fails because Warren's first-order programs are ill-typed in the sense of the polymorphic type systems.

This paper presents a polymorphic type system for logic programming which allows the application of higher-order programming techniques. For Prolog-like applications of higher-order programming techniques it is possible to compute optimizations by abstract interpretation so that the polymorphic logic programs have the same operational efficiency as untyped Prolog programs.

1 Introduction

Pure logic programming is based on untyped first-order logic and does not permit detection of many programming errors at compile time. Therefore various attempts have been made to integrate types into logic programming. One research direction follows an *operational approach*: The main goal of the operational approaches is to ensure that predicates are only called with appropriate arguments at run time. This should be attained by a static analysis of the program. These approaches have only a syntactic notion of "type", i.e., types are sets of terms rather than subsets of carrier sets of interpretations (on which the declarative semantics of logic programs is based [22]). Type information is frequently inferred by a type inference algorithm (see, for instance, [25] [20] [38] [21] [15] [5] [37]). Mycroft and O'Keefe [27] have adapted the polymorphic type discipline of modern functional languages [8] to pure Prolog. In their proposal the programmer has to declare the types of functions and predicates and the types of variables in clauses are inferred by a type checker. Since they have put restrictions on the use of polymorphic types in function declarations and clauses, their programs "do not go wrong" in the sense of well-typedness. But these restrictions prevent the application of higher-order programming techniques in the sense of Warren [35] (see below). The type system was extended to subtypes on the basis of mode declarations by Dietrich and Hagl [10], but they have no semantic notion of a type, similarly to Mycroft and O'Keefe's work.

Another research direction, to which this papers belongs, follows a *declarative approach*: The programmer has to declare all types of functions and predicates that he wants to use in the program. These approaches have a formal semantics of a type, i.e., types represent subsets of carrier sets of interpretations. This influences the operational mechanism because correctness can only be ensured by a typed unification procedure. In many-sorted Horn logic [29] typed unification is the same as untyped unification, but in order-sorted logic [11] [33], polymorphically order-sorted logic [32], or in a logic with subtypes and inheritance [3] the unification procedure has to consider the types of terms.

Higher-order objects well-known from functional languages are another useful extension to pure logic programming. Aït-Kaci, Lincoln and Nasr [2] have proposed an untyped language with functions and relations which permits higher-order functions. Although they have presented an operational semantics for this language based on delayed evaluation of expressions, a declarative semantics is not defined. Generally, a semantically clean amalgamation of higher-order objects with logic programming techniques like unification is not trivial since the unification of higher-order terms is undecidable in general [12]. Miller and Nadathur [24] 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 leads to a complex and semi-decidable unification [18]. The latter efficiency argument is our motivation to keep *first-order* Horn clause logic as our formal framework. But there is still another reason: Higher-order unification means solving equations between functions and guessing or computing new functions that satisfy the equations. This is a new feature not available in functional languages and, in our opinion, not necessary for programming. From a practical point of view it is sufficient to apply only user-defined functions to appropriate arguments at run time.

In order to integrate the higher-order facilities of functional languages into logic programming, Warren [35] has shown that first-order logic need not be extended because the usual higher-order programming techniques can be simulated in first-order logic by an axiomatization of an `apply` predicate. Since he is concerned with Prolog and its untyped logic, he does not have a clear distinction between first-order and higher-order objects. A type system may help to obtain this distinction. But the clauses for the `apply` predicate are not well-typed in the sense of the usual polymorphic type systems for logic programming [27] [10] [32] (see below).

Mycroft and O'Keefe [27] have also proposed an `apply` predicate for higher-order programming, but the precise meaning of the predicate is not defined. Prolog has also a predefined predicate `call` for the application of higher-order programming techniques [7], but the meaning of `call` cannot be described in first-order logic because the called predicate name must be instantiated to an atom at run time (a non-declarative restriction).

Hanus [14] has proposed a generalized polymorphic type system for logic programming based on a semantic notion of polymorphic types. Since the type system is rather general, it is necessary to consider the types of terms in the unification procedure. In this paper we show that Warren's first-order specification of the `apply` predicate is well-typed in the sense of [14]. Therefore this type system is a basis for the application of higher-order programming techniques in a typed framework. Moreover, we show that in most applications it is possible to compute optimizations by abstract interpretation so that the polymorphic logic programs can be executed with the same efficiency as untyped Prolog programs, i.e. the typed unification can be replaced by untyped unification without loss of soundness.

The next section gives an outline of the polymorphic type system of [14] and presents an example with higher-order predicates. In a further section we develop an optimization technique based on abstract interpretation to detect the cases where all type information can be omitted at run time.

2 Polymorphically typed logic programs

We are interested in an ML-like polymorphic type system for logic programming, i.e., types may contain type variables that are universally quantified over all types [8]. The programmer has to declare the types of all functions and predicates occurring in the program. The types of variables in clauses can be inferred by an ML-like type checker. Therefore the clauses of a program need not be annotated with type information by the programmer, but the type annotations are computed by the type checker. Moreover, the type checker can detect a lot of programming errors at compile time.

The typing rules are quite simple. First the programmer has to declare the *basic types* like *int* or *bool* and *type constructors* like *list* which he wants to use in the program. Each type constructor has a fixed *arity* (e.g., *list* has arity 1, denoted by $list/1$). We assume a given infinite set of *type variables* and we denote members of this set by α and β. A (*polymorphic*) *type* is a term built from basic types, type constructors and type variables (see [17] for the notion of term). A *monomorphic type* is a type without type variables. For instance, $list(int)$ and $list(\alpha)$ are monomorphic and polymorphic types which denote lists of integers and lists of elements of an arbitrary type, respectively. A type τ_1 is an *instance* of another type τ_2 if τ_1 can be obtained from τ_2 by replacing type variables in τ_2 by other types. Two types τ_1 and τ_2 are *equivalent* if

τ_1 is an instance of τ_2 and τ_2 is an instance of τ_1, i.e., they are equal up to renaming of type variables.

Next the user has to declare the argument and result types of functions and predicates occurring in the program. A function declaration has the form

func `f:` $\tau_1, \ldots, \tau_n \rightarrow \tau$

(where $\tau_1, \ldots, \tau_n, \tau$ are arbitrary types) and means that function `f` takes n arguments of types $\tau_1, \ldots, \tau_n$ and produces a value of type τ. `f` is called *constant* of type τ if $n = 0$. A predicate declaration has the form

pred `p:` $\tau_1, \ldots, \tau_n$

(where $\tau_1, \ldots, \tau_n$ are arbitrary types) and means that predicate `p` has n arguments of types $\tau_1, \ldots, \tau_n$. In order to compute the most general type of a term and to apply some optimization techniques (see below), we forbid overloading: For each function and predicate symbol there is only one type declaration. Note that there are no restrictions on the use of type variables in function declarations in contrast to [27].[1]

The type variables in a declaration are universally quantified over all types, i.e., functions and predicates can be used with an arbitrary substitution of types for type variables in the declaration. Hence we call a function or predicate declaration a *generic instance* of another declaration if it can be obtained from the other declaration by replacing each occurrence of one or more type variables by other types (cf. [8]). For instance, if there is the declaration

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

for the predicate `append`, then

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

is a generic instance of the declaration.

According to [6], types are embedded in terms, i.e., each symbol in a term is annotated with an appropriate type expression. These annotations are useful for the unification of polymorphic terms (see below). The type annotations need not be provided by the user because most general type annotations can be computed. We assume a given infinite set Var of variable names distinguishable from type variables. A *typed variable* has the form $x{:}\tau$ where $x \in Var$ and τ is an arbitrary type. We call V an *allowed set of typed variables* if V contains only typed variables and $x{:}\tau, x{:}\tau' \in V$ implies $\tau = \tau'$. We call $L \leftarrow G$ a *polymorphic program clause* if there is an allowed set of typed variables V and $V \vdash L \leftarrow G$ is derivable by the inference rules in table 1. Note that there are no restrictions on the use of types and type variables in clauses in contrast to [27], [32] and similar type systems.[2] A

[1] In their type system each type variable occurring in the argument type of a function must also occur in the result type [26].

[2] In these type systems the left-hand side of a clause must have a type that is equivalent to the declared type of the predicate.

Variable:	$\dfrac{}{V \vdash x{:}\tau}$	$(x{:}\tau \in V)$
Term:	$\dfrac{V \vdash t_1{:}\tau_1, \ldots, V \vdash t_n{:}\tau_n}{V \vdash f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau}$	$(f{:}\tau_1, \ldots, \tau_n \to \tau$ is a generic inst. of a function declaration, $n \geq 0)$
Literal:	$\dfrac{V \vdash t_1{:}\tau_1, \ldots, V \vdash t_n{:}\tau_n}{V \vdash p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)}$	$(p{:}\tau_1, \ldots, \tau_n$ is a generic instance of a predicate declaration, $n \geq 0)$
Clause:	$\dfrac{V \vdash L_0, \ldots, V \vdash L_n}{V \vdash L_0 \leftarrow L_1, \ldots, L_n}$	(each L_i has the form $p(\cdots)$, $i = 0, \ldots, n)$

Table 1: Typing rules for polymorphic program clauses

polymorphic logic program is a finite set of polymorphic program clauses.

We give an example of a polymorphic logic program that defines the higher-order predicate `map`. The constant `[]` represents the empty list, and the function • concatenates an element with a list of the same type. We write `[E|L]` instead of `•(E,L)` (throughout this paper we use the Prolog notation for lists, cf. [7]). The literal `map(P,L1,L2)` should be satisfied if predicate `P` is satisfied for each pair of corresponding elements from lists `L1` and `L2`. In order to define the type of `map` we introduce a type constructor *pred*2 of arity 2 that denotes the type of binary predicates. 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 λp of type "$pred2(\tau_1, \tau_2)$". The relation between each predicate p and the constant λp is defined by clauses for the predicate `apply2`. Hence we get the following example program for the predicate `map` (we omit type annotations in program clauses because they can be automatically inferred by an ML-like type checker [8]):

type *bool*, *nat*, *list*/1, *pred*2/2
func `true` : $\to bool$
func `false`: $\to bool$
func `0` : $\to nat$
func `s` : $nat \to nat$
func `[]` : $\to list(\alpha)$
func • : $\alpha, list(\alpha) \to list(\alpha)$
func λ`not`: $\to pred2(bool, bool)$
func λ`inc`: $\to pred2(nat, nat)$
pred `not`: $bool,\ bool$
pred `inc`: $nat,\ nat$
pred `map`: $pred2(\alpha, \beta),\ list(\alpha),\ list(\beta)$

```
pred apply2:  pred2(α,β),  α,  β

clauses:
map(P,[],[]) ←
map(P,[E1|L1],[E2|L2]) ← apply2(P,E1,E2), map(P,L1,L2)
not(true,false) ←
not(false,true) ←
inc(N,s(N)) ←
apply2(λnot,B1,B2) ← not(B1,B2)
apply2(λinc,I1,I2) ← inc(I1,I2)
```

We can use this definition of `map` to compute a new list from a given one or to search for predicates which relate two lists (see next section for operational semantics):

```
?- map(λnot,[true,false,false],L)
L = [false,true,true]
?- map(P,[0],[s(0)])
P = λinc
```

Note that the last two clauses for the predicate `apply2` are not well-typed in the sense of [27] and similar type systems since `apply2` has the declared type "$pred2(\alpha,\beta),\alpha,\beta$" but is used in the clause heads with the specialized types "$pred2(bool,bool),bool,bool$" and "$pred2(nat,nat),nat,nat$", respectively. Therefore such first-order axiomatizations of higher-order predicates cannot be well-typed with the usual polymorphic type systems in a reasonable way.

We remark that it is also possible to permit lambda expressions which can be translated into new identifiers and `apply` clauses for these identifiers (see [35] for more discussion). If the lambda expressions have polymorphic arguments that do not occur in the result (e.g., the length function on lists `length:` $list(\alpha) \rightarrow nat$), then typing of such expressions is no problem because type variables in argument types that do not occur in the result type are permitted in our type system in contrast to [27]. If the underlying system implements indexing on the first arguments of predicates (as done in most compilers for Prolog, cf. [36] and [13]), 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 [35].

Since our foundation is first-order logic, the predicate symbol `map` is semantically not interpreted as a higher-order predicate. The constants λ`not` and λ`inc` are also interpreted as values and not as predicates. But the first `apply2` clause ensures that in each model of the above program the constant λ`not` and the predicate `not` are related together. From an operational point of view the behaviour of our `map` program is similar to the behaviour of a corresponding program in a higher-order language. To be more precise we give a short outline of the semantics in the next section.

3 Declarative and operational semantics

First we give a short outline of the declarative semantics. The type variables in a clause vary over all possible types, i.e., a clause containing a type variable α is true for each type which is substituted for α. The carrier of an *interpretation* is a family of sets with one set for each (monomorphic) type. Function symbols are interpreted as functions on appropriate carrier sets, and predicates are interpreted as relations between appropriate carrier sets. This leads to a similar notion of *validity* and equivalent results as in many-sorted Horn clause logic [29]. Especially, an initial model can be constructed as the intersection of all Herbrand models where the carrier sets contain all ground terms with monomorphic types. We call a *goal* (goals have the same form as the right-hand side of a polymorphic program clause) *valid with respect to a program P* if it is valid in each interpretation which satisfies all clauses from P.

In [14] it is shown that resolution is a sound and complete proof procedure for polymorphic logic programs with goals that may contain type variables if the unification is extended to polymorphic terms. The unification of polymorphic terms can be reduced to common first-order unification [30] if the annotated types are treated as first-order terms. Type terms are distinguished from other terms by their position (type terms occur only after a colon ':'). For instance, a unifier of the polymorphic terms `[]`:$list(\alpha)$ and `v`:$list(int)$ is the substitution that replaces α by int and `v` by `[]`. This could also be computed by a first-order unification algorithm if the symbol ':' is treated as a term constructor of arity 2. Therefore it is possible to translate polymorphic logic programs into Prolog programs (more details can be found in [14]).

Since we have an unrestricted type system, the unification procedure has to consider the types of polymorphic terms. Otherwise the resolution is not sound and produces ill-typed goals. For instance, assume that the above `map` program is given. We may ask for a predicate that relates two lists of naturals and apply this predicate to the constant `0`:

```
?- map(P,[N1],[N2]), apply2(P,0,N3)
```

(`P` has type $pred2(nat, nat)$ and `N1`, `N2` and `N3` have type nat). If we omit all types at run time and solve the goal by the computation rule of Prolog, `P`, `N1` and `N2` are bound to λ`not`, `true` and `false`, respectively, and we obtain the ill-typed goal

```
?- apply2(λnot,0,N3)
```

Resolution with typed terms and typed unifiers is correct, since

`apply2(P:`$pred2(nat, nat)$`,N1:`nat`,N2:`nat`)`

and

`apply2(λnot:`$pred2(bool, bool)$`,B1:`$bool$`,B2:`$bool$`)`

are not unifiable. But

$\texttt{apply2(P:}pred2(nat,nat)\texttt{,N1:}nat\texttt{,N2:}nat\texttt{)}$

and

$\texttt{apply2(}\lambda\texttt{inc:}pred2(nat,nat)\texttt{,I1:}nat\texttt{,I2:}nat\texttt{)}$

are unifiable and therefore we obtain the well-typed goal

$\texttt{?- apply2(}\lambda\texttt{inc:}pred2(nat,nat)\texttt{,0:}nat\texttt{,N3:}nat\texttt{)}$

in the resolution process. This example demonstrates that types are necessary for soundness of resolution in general. But there are some cases where types are unnecessary in the resolution process. This should be shown in the next section.

4 Optimization

The unification procedure has to consider the types of polymorphic terms because of our unrestricted type system. Hence the polymorphic unification is more complex and less efficient than the unification in untyped logic languages. But Mycroft and O'Keefe [27] have shown that types can be completely omitted at run time if the program has particular restrictions. In section 2 we have seen that these restrictions prevent the application of higher-order programming techniques. Nevertheless, there are cases where all type information can be omitted at run time in the presence of clauses for an `apply` predicate. This section presents some sufficient criteria to omit type information in the resolution process. It is shown that in Prolog-like applications of higher-order predicates no type information is needed at run time.

4.1 General optimization techniques

First we review two optimizations mentioned in [14]. One optimization can be applied to most functions: A function symbol f is called *type preserving* if all type variables occurring in the argument type also occur in the result type. For instance, all function symbols in the above `map` program are type preserving, whereas the function

func `first:` $pair(\alpha,\beta) \rightarrow \alpha$

is not type preserving. Since all type information of a type preserving function can be computed from the type declaration and the actual instantiation of the result type, the type annotations of arguments can be omitted at unification time. Hence only the direct result types of the arguments of predicates are needed in the above `map` program. For instance, we need instead of the completely typed goal

$$\texttt{map(P:}pred2(\alpha,\beta)\texttt{,[E1:}\alpha\texttt{|L1:}list(\alpha)\texttt{]:}list(\alpha)\texttt{,[E2:}\beta\texttt{|L2:}list(\beta)\texttt{]:}list(\beta)\texttt{)}$$

only the goal

$$\texttt{map(P:}pred2(\alpha,\beta)\texttt{,[E1|L1]:}list(\alpha)\texttt{,[E2|L2]:}list(\beta)\texttt{)}$$

for correctly typed unification.

All types can be omitted in the resolution proof of a predicate if the predicate is type-generally defined: Intuitively, a predicate is called *type-generally defined* if in each clause for the predicate all predicates in the body are also type-generally defined and the clause head has a most general type according to the type declaration of the predicate. Usually, a literal with predicate symbol p has a *most general type* if the types of the arguments are equivalent to the declared type of p. For instance, the clause head

`map(P:`$pred2(\alpha,\beta)$`,[E1|L1]:`$list(\alpha)$`,[E2|L2]:`$list(\beta)$`)`

has a most general type, but not

`apply2(`λ`not:`$pred2(bool,bool)$`,A1:`$bool$`,A2:`$bool$`)`

because the declared type of `apply2` is "$pred2(\alpha,\beta),\alpha,\beta$".

In a well-typed program in the sense of Mycroft and O'Keefe each function must be type preserving and each predicate must be type-generally defined. This is the reason why these programs can be executed without dynamic type checking. As shown above, the first-order specification of an `apply` predicate is not a type-general definition and therefore only the first optimization technique for type preserving functions is applicable to programs with higher-order predicates. In order to omit all types in this case, better optimization techniques are required.
In which cases does the untyped unification lead to ill-typed goals? Two conditions must be satisfied:

1. There is a predicate p and a clause for p where the head of the clause does not have a most general type w.r.t. the type declaration of p (otherwise all predicates are type-generally defined and all types can be omitted, see above).

2. At run time there is a call to this predicate and after the call some variables are bound to terms with specialized types (for instance, if the `map` program of section 2 is given, a call of `apply2(P,A1,A2)` bind variable `P` to λ`not` which has type $pred2(bool,bool)$).

If type annotations should be omitted at run time, it must be ensured that these conditions cannot be satisfied. Therefore we need some knowledge about the run-time behaviour of the program. Since detailed run-time properties of Prolog programs are generally undecidable, we give sufficient criteria for the absence of the second condition.

We call a predicate p *type-specialized* if there is a program clause of the form

$$p(t_1{:}\tau_1,\ldots,t_n{:}\tau_n) \leftarrow L_1,\ldots,L_m$$

where $\tau_1,\ldots,\tau_n$ is not equivalent to the declared type of p. The *critical points of a program* are the calls of type-specialized predicates. We need a sufficient criterion to ensure that

- the unifiability of literals with clause heads does not depend on argument types, and
- no variable arguments in literals are bound to terms with specialized types while unifying the literal with a clause head.

In this case the argument types can be omitted at run time. A sufficient criterion for this property is *groundness of arguments*: If there is an argument of the predicate call which is a ground term and the corresponding argument in the clause head has a most general type, then the unification does not depend on the argument types and no variable argument is instantiated to a term with a specialized type. In the following we give a precise definition of this criterion.

Let p be a type-specialized predicate with declared type $\nu_1, \ldots, \nu_n$, $p(r_1{:}\rho_1, \ldots, r_n{:}\rho_n)$ be a literal that calls p and

$$p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n) \leftarrow L_1, \ldots, L_m$$

be a clause for p. The type annotations $\rho_1, \ldots, \rho_n$ and $\tau_1, \ldots, \tau_n$ are not required for correct unification if the following conditions are satisfied:

1. There is an argument position i ($1 \leq i \leq n$) such that ν_i contains all type variables occurring in the declared type $\nu_1, \ldots, \nu_n$.
2. $r_i{:}\rho_i$ is a ground term.
3. The term $t_i{:}\tau_i$ has a most general type w.r.t. ν_i, i.e., there is no term $t_i'{:}\tau_i'$ which is equal to $t_i{:}\tau_i$ up to type annotations so that τ_i' is an instance of ν_i, τ_i is an instance of τ_i', and τ_i is not equivalent to τ_i'.

By the first condition, the actual instantiation of type $\nu_1, \ldots, \nu_n$ is determined by the type of the i-th argument. This condition could be extended to more than one argument position that contain all type variables and are ground, but this would complicate the definition. The condition with one argument position is sufficient for our purpose of higher-order programming. The third condition ensures that each other well-typing of the i-th argument term of the clause head gives a type which is an instance of τ_i (the notion "*equal up to type annotations*" means that the terms become equal if all type annotations are deleted).

We show that the types of predicate arguments can be omitted if the above conditions are satisfied. If $p(r_1, \ldots, r_n)$ and $p(t_1, \ldots, t_n)$ are not unifiable, then $p(r_1{:}\rho_1, \ldots, r_n{:}\rho_n)$ and $p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)$ are not unifiable, too. Let $p(r_1, \ldots, r_n)$ and $p(t_1, \ldots, t_n)$ be unifiable. Then r_i is also unifiable with t_i. Since $t_i{:}\tau_i$ has a most general type (condition 3), ρ_i is an instance of type τ_i. Thus a most general unifier σ of $r_i{:}\rho_i$ and $t_i{:}\tau_i$ exists and does not instantiate any type variable in ρ_i (w.l.o.g. all type variables in $t_i{:}\tau_i$ are disjoint from type variables in $r_i{:}\rho_i$). $\sigma(\tau_1, \ldots, \tau_n) = \rho_1, \ldots, \rho_n$ because $\sigma(\tau_i) = \sigma(\rho_i) = \rho_i$ and the declared argument type ν_i contains all type variables of the declared type (condition 1). Therefore $p(r_1{:}\rho_1,\ldots,r_n{:}\rho_n)$ and $p(t_1{:}\tau_1,\ldots,t_n{:}\tau_n)$ are unifiable, too, and a most general unifier does not instantiate any variable argument in $p(r_1{:}\rho_1,\ldots,r_n{:}\rho_n)$ to a term with a specialized type. Hence the absence of argument types does not influence unifiability and the types of arguments can be omitted at run time.

Condition 3 is necessary for this optimization. For instance, a predicate `p` is declared by

pred `p`: $list(\alpha)$, α

and the only call of predicate `p` is of the form `p([]`:$list(bool)$`,B`:$bool$`)`. If the only clause for `p` is

`p([]`:$list(nat)$`,0`:nat`)` $\leftarrow$

then the clause is not applicable to literal `p([]`:$list(bool)$`,B`:$bool$`)`. If we omit the argument types, then the clause would be applicable and the Boolean variable `B` would be instantiated to term `0` of type "*nat*". If the only clause for `P` has the form

`p([]`:$list(\alpha)$`,X`:α`)` $\leftarrow$

then the argument types can be omitted because type variable α is instantiated to "*bool*". The argument types can also be omitted if there is a clause of the form

`p([0]`:$list(nat)$`,X`:nat`)` $\leftarrow$

(we omit the type annotation of `0` because function '•' is type preserving) since the term `[0]`:$list(nat)$ has a most general type.

The above three conditions can be checked at compile time if information about the groundness of variables at run time is available. Condition 3 can be checked by the use of a type checker that computes the most general type of a term. Groundness condition 2 can be verified if we know the modes of type-specialized predicates [23] [9] [34]: If the i-th argument of predicate p is in input mode, then $r_i{:}\rho_i$ is always ground at run time. Since we only need groundness information about some arguments and not the modes of all predicates, we prefer to use abstract interpretation techniques [1] to compute groundness information at compile time. Abstract interpretation have been applied to derive groundness information by Jones, Søndergaard [19] and Nilsson [28]. The results of abstract interpretation are sufficient for our purposes.

4.2 Application to logic programs with higher-order predicates

We have implemented the optimization techniques developed above and we have applied it to a number of logic programs with first-order axiomatizations of higher-order predicates. In the following we outline the implementation and the results.

To compute groundness information of variables we have implemented

Nilsson's abstract interpretation technique [28]. The computation of information is based on distinguished *program points*. The point before a literal L in the body of a clause or in a goal is called *calling point of* L and the point after L is called *success point of* L. Abstract interpretation of a logic program is started by supplying a goal pattern that describes possible initial goals and groundness information of these goals. Then the program is translated into a graph structure and groundness information is computed for each program point. For each critical program point (calling point of a type-specialized predicate) the above three criteria are checked. If these criteria are satisfied for each critical program point, then all type annotations of predicate arguments can be omitted.

For instance, assume that the above polymorphic logic program for the predicate `map` is given. The initial goal information is

```
map(P,L1,L2)  with ground P
```

which means that the `map` program is only started with a goal of the form `map(P,L1,L2)` where the first argument is a ground term. Now groundness information is computed for all program points. Since `apply2` is the only type-specialized predicate, the only critical program point is the calling point of literal `apply2(P,E1,E2)` in the second clause for the `map` predicate. The following groundness information is computed for this polymorphic clause (the list of ground variables is shown for each calling and success point of each literal in the clause body):

```
map(P:pred2(α,β),[E1|L1]:list(α),[E2|L2]:list(β)) ←
*** Ground variables:  [P]
      apply2(P:pred2(α,β),E1:α,E2:β),
*** Ground variables:  [P,α,β]
      map(P:pred2(α,β),L1:list(α),L2:list(β))
*** Ground variables:  [P,α,β]
```

(the type annotations of arguments of type preserving functions are omitted). We see that after a successful call of the `apply2`-literal the type variables α and β are also bound to monomorphic types since there are only `apply2`-clauses for monomorphically typed predicates. Next the three conditions are checked for the `apply2`-literal:

1. The declared type of `apply2` is "$pred2(\alpha,\beta),\alpha,\beta$" and therefore the type of the first argument contains all type variables occurring in the declared type.

2. The first argument of the `apply2`-literal is a ground term.

3. The first arguments of the heads of the `apply2`-clauses are λ`not`:$pred2(bool,bool)$ and λ`inc`:$pred2(nat,nat)$, respectively. Since these constants are declared with monomorphic types, these terms have most general types.

Hence all type annotations can be omitted at run time of this program since all functions are type preserving.

The actual implementation automatically checks all conditions. For condition 3 all types in term $t_i{:}\tau_i$ are deleted, the type checker is called for the term without type annotations to obtain a term $t_i'{:}\tau_i'$ with most general type annotations, and it is checked whether τ_i and τ_i' are equivalent. Since the implementation language is Prolog, equivalence of type expressions can be decided by the use of the common meta-logical predicate *numbervars*.

It is easy to see that conditions 1 and 3 are satisfied by all clauses for an `apply` predicate in our implementation scheme for higher-order predicates: If an higher-order predicate which takes an n-ary predicate as argument should be defined, we have to declare a type constructor *predn* with arity n, a corresponding predicate

pred `apply`n: $\quad predn(\alpha_1, \ldots, \alpha_n),\ \alpha_1,\ \ldots,\ \alpha_n$

and for each n-ary predicate

pred `p`: $\quad \tau_1,\ \ldots,\ \tau_n$

a constant

func λ`p`: $\quad \rightarrow predn(\tau_1, \ldots, \tau_n)$

For each of these constants there is a clause

$$\mathtt{apply}n(\lambda\mathtt{p}{:}predn(\tau_1,\ldots,\tau_n),\mathtt{A}_1{:}\tau_1,\ldots,\mathtt{A}_n{:}\tau_n) \leftarrow \mathtt{p}(\mathtt{A}_1{:}\tau_1,\ldots,\mathtt{A}_n{:}\tau_n)$$

The type of the first argument of `apply`n contains all type variables occurring in the declared type of p (condition 1). Since $predn(\tau_1, \ldots, \tau_n)$ is the declared type of constant λp, the term $\lambda\mathtt{p}{:}predn(\tau_1, \ldots, \tau_n)$ has a most general type (condition 3). Condition 2 is dependent on the actual program, but the abstract interpreter gives precise information in most cases.

We have applied these optimization techniques to several polymorphic logic programs with higher-order predicates and obtained the following results:

- If higher-order predicates are used in a Prolog-like way, i.e., the `apply` predicate is only used with a non-variable first argument at run time (this is the only possibility in Prolog, otherwise a run-time error occurs), then this is recognized by the optimizer and no types are needed at run time. This means that in Prolog-like applications of higher-order predicates we have no overhead because of types.

- The cases where types are needed at run time are rare in practical programs. It occurs when somebody wants to *compute predicates* that could be applied to certain terms. This feature is not available in Prolog and it shows that our notion of higher-order programming has a declarative meaning and is type secure in contrast to Prolog. In these cases types are not superfluous but may reduce the search space (this is also an argument to include types in the computation process of order-sorted logic [16]).

5 Conclusions

We have presented a polymorphic type system for logic programming that allows the application of higher-order programming techniques. In most cases, polymorphic logic programs with higher-order predicates can be executed with the same efficiency as untyped logic programs if an optimization technique based on the computation of groundness information by abstract interpretation is used. The polymorphic logic language has a well-defined semantics based on first-order logic. Higher-order objects are specified by a name and distinguished clauses that defines the application of these objects to other ones. The necessary definitions can be automatically generated. This technique was proposed by Warren for untyped Horn clause logic.

Logic programming with higher-order functions was also proposed by Smolka [31]. In his language Fresh higher-order functional programming is combined with unification. To avoid the difficulties with higher-order unification, a name is associated with each function and equality between functions is defined as identity of associated names. This is similar to our approach except that Fresh is an untyped language.

The compilation of higher-order functions into first-order logic was also proposed by Bosco and Giovannetti [4], but in their language IDEAL type-checking is only performed for the source program and not for the target program. Clearly, the target program is not well-typed in the sense of [27] because of the clauses for the `apply` predicate.

Since higher-order logic programs can be translated into polymorphic logic programs, the use of higher-order objects is type secure in our framework. The typing rules are similar to functional languages and the type system ensures that a predicate is only called with appropriate arguments at run time. Hence our polymorphic logic language is a sound and clearly defined framework for higher-order programming in comparison with other ad-hoc approaches (`call` predicate in Prolog, `apply` predicate in [27]). Since we have restricted the domain of predicate variables to user-defined predicates, our theoretical foundation is first-order logic and not higher-order logic. The advantage of this restriction is an efficient operational semantics, and the occurrence of type variables in goals raises no problems (in contrast to [24]).

Further work remains to be done. From a practical point of view a type inference algorithm should automatically derive type declarations for predicates from the given program [37]. But the distinction between type errors and well-typings is not trivial because of our general type system.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and

Functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 17–23, San Francisco, 1987.

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

[4] 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.

[5] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inferencing. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 669–683, 1988.

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

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

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[9] S.K. Debray and D.S. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming (5)*, pp. 207–229, 1988.

[10] 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.

[11] 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.

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

[13] M. Hanus. Formal Specification of a Prolog Compiler. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, Orléans, 1988. Springer LNCS 348.

[14] M. Hanus. Horn Clause Programs with Polymorphic Types: Semantics and Resolution. In *Proc. of the TAPSOFT '89*, pp. 225–240. Springer LNCS 352, 1989. Extended version to appear in *Theoretical Computer Science.*

[15] K. Horiuchi and T. Kanamori. Polymorphic Type Inference in Prolog by Abstract Interpretation. In *Logic Programming '87 (Tokyo)*, pp. 195–214. Springer LNCS 315, 1987.

[16] 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.

[17] G. Huet and D.C. Oppen. Equations and Rewrite Rules: A Survey. In R.V. Book, editor, *Formal Language Theory: Perspectives and Open Problems.* Academic Press, 1980.

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

[19] N. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of PROLOG. In S. Abramsky and C. Hankin,

editors, *Abstract Interpretation of Declarative Languages*, pp. 123–142. Ellis Horwood, 1987.

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

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

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

[23] C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming (1)*, pp. 43–66, 1985.

[24] 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.

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

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

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

[28] U. Nilsson. Towards an Abstract Interpretation Scheme for Logic Programs. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, Orléans, 1988. Springer LNCS 348.

[29] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1988.

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

[31] G. Smolka. Fresh: A Higher-Order Language Based on Unification and Multiple Results. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 469–524. Prentice Hall, 1986.

[32] G. Smolka. Logic Programming with Polymorphically Order-Sorted Types. In *Proc. First International Workshop on Algebraic and Logic Programming (Gaussig, G.D.R.)*, pp. 53–70. Akademie-Verlag (Berlin), 1988.

[33] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-Sorted Equational Computation. SEKI Report SR-87-14, FB Informatik, Univ. Kaiserslautern, 1987.

[34] Z. Somogyi. A system of precise modes for logic programs. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 769–787. MIT Press, 1987.

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

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

[37] J. Xu and D.S. Warren. A Type Inference System For Prolog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 604–619, 1988.

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