In *Proc. Int. Symposium on Design and Implementation of Symbolic Computation Systems*,
pp. 215-224, Springer LNCS 429, 1990

A Functional and Logic Language with Polymorphic Types
(Extended Abstract)

Michael Hanus
Fachbereich Informatik, Universität Dortmund
D-4600 Dortmund 50, W. Germany
e-mail: michael@ls5.informatik.uni-dortmund.de

This paper presents a typed language with a precisely defined semantics that integrates functional and logic programming styles. To detect programming errors at compile time, the language has a polymorphic type system. The type system restricts the possible use of functions and predicates and ensures that a function or predicate is only called with appropriate arguments at run time. In contrast to many other type systems for logic programming, this type system has a model-theoretic semantics (types are subsets of domains of interpretations) and allows the use of typical logic programming techniques. For instance, it is possible to add specialized clauses and to apply higher-order programming techniques.

1 Introduction

During recent years, various attempts have been made to amalgamate functional and logic programming languages (see [DL86] for a collection of proposals). A lot of these integrations are based on Horn clause logic with equality which consists of predicates and Horn clauses for logic programming and functions and equations for functional programming. Because of the underlying single-sorted logic, a lot of these languages are untyped which is a disadvantage from a practical point of view. Modern functional programming languages, like ML [HMM86], Miranda [Tur85] and HOPE [BMS80], have polymorphic type systems to detect specific programming errors at compile time. Due to the polymorphism, the restrictions of the type system are acceptable for practical programming.

The lack of types for logic programming was recognized by several people, and therefore a lot of research for types in logic programming languages have been done. In contrast to functional languages, where types have a declarative meaning (e.g., [DM82], but a model-theoretic semantics for types in functional languages is also an area of ongoing research), many proposals for type systems for logic languages have a notion of a type that is not directly related to the declarative semantics. For instance, the polymorphic type systems in [MO84] and [DH88] require declarations for types and predicates but do not define the semantics of such declarations. Declarations are not required in the proposals [Mis84], [Klu87] and [Zob87] but are inferred by a type checking algorithm. These and other type systems have only a syntactic notion of a type, i.e., types are sets of terms rather than subsets of carrier sets of interpretations. Moreover, the inference of types from a completely untyped program yields only in a few cases the types expected by the programmer. For instance, assume *list* denotes the set of all terms of the form `[]` or `[E|L]` where `L` is a term from *list*. Then the inferred type for the predicate `append` defined by

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

may be "$list \times \alpha \times \alpha \,\cup\, list \times \beta \times list$" [XW88], where α and β denote arbitrary types. The problem is the first clause which defines `append` to be true not only for lists but also for other terms. E.g., `append([],3,3)` is true but usually considered as an ill-typed goal, i.e., the expected type is "$list \times list \times list$". This example shows that inferring types from a completely untyped program is generally not acceptable since an untyped logic program does not contain the type information expected by the programmer (see also [Nai87]). A type system should *allow user declarations for types.* Such declarations frequently documents the expected meaning of predicates and improves the readability of large programs. Therefore the language presented in this paper is *explicitly typed*, i.e., each variable, function and predicate has an explicitly given type. A program where the types of some variables are not declared is viewed as a short-hand for an explicitly typed program. A type inference procedure can be used to insert the missing type declarations for variables.

Since pure functional and logic languages are declarative languages, an integration of these languages should be also declarative. Hence each declaration in the language should have a clearly defined model-theoretic semantics. If such a language is enriched by a type structure, the semantics of the type structure must be defined. But this have been done only in a few proposals. For instance, Padawitz [Pad88] has proposed a Horn clause logic with equality with a many-sorted type structure. Each type corresponds to a non-empty subset of the carrier set of the interpretation. Goguen and Meseguer [GM84] have shown that the common deduction rules for untyped equational logic become unsound if types denote empty sets. Hence the definition of the model-theoretic semantics of types is necessary to establish exact soundness and completeness results.

The simple many-sorted type structure was extended in several directions: Goguen, Meseguer [GM86] and Smolka [Smo86b] have proposed order-sorted type systems for Horn clause logic with equality where the ordering of types implies a subset relation on the corresponding sets. Aït-Kaci and Nasr [AN86] have proposed a logic language with a type system that includes subtypes and inheritance based on a similar semantics. From an operational point of view the approaches with subtypes need a unification procedure that considers types, i.e., types are present at run time. Smolka [Smo88b] has proposed an equational logic language with a polymorphic type system [Smo88a]. Since his language also includes an order-sorted type system, he has some restrictions on his type system and therefore the application of higher-order programming techniques, well-known from functional languages, is not possible (see below). A polymorphic type system for pure logic programming, based on a model-theoretic semantics, has been proposed in [Han89a]. Since it is rather general, it allows the application of higher-order programming techniques [Han89b]. Therefore it is the basis of the type system of the language presented in this paper.

Higher-order programming is a useful feature in functional languages. Therefore we are also interested in the application of higher-order programming techniques in our (first-order) framework. Warren [War82] has shown that higher-order programming techniques can be simulated in first-order logic if higher-order predicates are specified by clauses for an `apply` predicate. For instance, the following clauses define two binary predicates `not` and `inc` and the predicate `map` which applies a binary predicate to corresponding elements of two lists:

```
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(not,B1,B2) ← not(B1,B2)
apply2(inc,I1,I2) ← inc(I1,I2)
```

The following goal shows an application of the `map` predicate:

```
?- map(not,[true,false,true],L).
L = [false,true,false]
```

Since such a specification is first-order, the result is not a higher-order logic, but predicate variables are universally quantified over all predicates defined in the program. One reason not to leave first-order logic in contrast to [MN86] is efficiency: The higher-order unification problem is undecidable in general [Gol81] and a complete unification procedure is a very complex task. But there is still another reason: Higher-order unification means searching new functions that satisfy a given set of equations. This feature is 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.

Since Warren's proposal is concerned with Prolog and its untyped logic, it is not type secure and there is no clear distinction between first-order and higher-order objects (predicate variables). Therefore a type system is needed. But the proposed polymorphic type systems for logic programming [MO84] [DH88]

[Smo88a] are not applicable because of the clauses for the `apply2` predicate: `apply2` has the polymorphic type "$pred2(\alpha,\beta),\alpha,\beta$" (where $pred2(\ldots,\ldots)$ denotes the type of binary predicate abstractions), but the type of `apply2` in the clause

    apply2(inc,I1,I2) ← inc(I1,I2)

is the specialized type "$pred2(nat,nat),nat,nat$". Hence this clause is ill-typed w.r.t. these type systems.[1] If we want to avoid any type-checking at run time, then it is not possible to construct a type system that can type the above program. For instance, consider the following goal (the constant `0` is of type *nat*):

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

If the Prolog computation rule is used and no type-checking is made during the resolution process, then `P`, `N1` and `N2` are bound to `not`, `true` and `false`, respectively, and the goal

    ?- apply2(not,0,N3)

is derived after four resolution steps. The last goal is ill-typed, because the predicate `not` is defined on Booleans, but not on naturals. If we use type information at run time, then this resolution can be avoided, because variable `P` has type $pred2(nat,nat)$ and may not be bound to `not` which is of type $pred2(bool,bool)$. This example shows that the translation of higher-order programs into first-order programs (as proposed for IDEAL [BG86]) is generally not type secure if the target language does not perform any type-checking at run time. Since type security is the main purpose of our language proposal, our operational semantics includes type information whenever it is necessary.

The rest of this paper presents the syntactical part of our language and an outline of the declarative and operational semantics. Let us summarize the main features of our language:

- The language combines functional programming with equations and logic programming with Horn clauses in a type secure way. The type system has parametric polymorphism, i.e., types may contain type variables that are universally quantified over all types [DM82].
- The language has a declarative semantics that is based on Horn clause logic with equality. The types have a declarative meaning, i.e., types are subsets of carriers sets of interpretations.
- Only well-typed programs have a declarative meaning. Therefore all programs are explicitly typed. A program with some missing type annotations is viewed as a short-hand for an explicitly typed program and the missing type annotations have to be inserted by a type inference procedure.
- The operational semantics is based on resolution for predicates and narrowing for functions. Because of the general type system the unification must be replaced by the typed unification of [Han89a]. The typed unification procedure is more expensive than the untyped one but may help to reduce the search space in the resolution process.
- Each variable, function and predicate has a distinct (polymorphic) type. The function and predicate types must be declared by the programmer and all clauses have to meet the declared type requirements. Thus our language is explicitly typed. But the implementation of the language has a type inference system based on [DM82] that deduces the types of variables in the given untyped clauses.
- The type system is rather general and allows the application of higher-order programming techniques in a type secure way. Some restrictions of the polymorphic type systems in [MO84], [DH88] and [Smo88a] are dropped.

2 The polymorphic equational logic language

The typing rules of our language are quite simple. We start by defining the language of types which must be specified for a program. 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). Basic types can be seen as type constructors of arity 0. We assume

[1] 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.

$$\frac{}{V \vdash x{:}\tau} \qquad (x{:}\tau \in V)$$

$$\frac{V \vdash t_1{:}\tau_1, \ldots, V \vdash t_n{:}\tau_n}{V \vdash f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau} \qquad (f{:}\tau_1, \ldots, \tau_n \rightarrow \tau \text{ is a generic instance of a function declaration, } n \geq 0)$$

$$\frac{V \vdash t_1{:}\tau_1, \ldots, V \vdash t_n{:}\tau_n}{V \vdash p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)} \qquad (p{:}\tau_1, \ldots, \tau_n \text{ is a generic instance of a predicate declaration, } n \geq 0)$$

$$\frac{V \vdash L_0, \ldots, V \vdash L_n}{V \vdash L_0 \leftarrow L_1, \ldots, L_n} \qquad (\text{each } L_i \text{ has the form } p(\cdots),\ i = 0, \ldots, n)$$

Figure 1: Typing rules for the polymorphic equational logic language

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 [HO80] for the notion of term). A **monomorphic type** is a type without type variables. For instance, $list(int)$ and $list(\alpha)$ are types which denote lists of integers and lists of elements of an arbitrary type, respectively.

Next the argument and result types of functions and predicates occurring in the program must be declared. 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 the 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 $n = 0$. For simplicity we have no distinction between constructors for data types and functions defined on data types [HH82], but in practice this distinction can be made by the form of equations (constructors do not occur as the top symbol of the left-hand side of an equation). This distinction is unnecessary from a declarative point of view. 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 the predicate `p` has n arguments of types $\tau_1, \ldots, \tau_n$. The **equality predicate** is always defined by

pred $\equiv$:α, α

In order to compute the most general type of a term automatically, we forbid overloading: For each function and predicate symbol there is only one type declaration.

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/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. [DM82]). For instance, `length`:$list(nat) \rightarrow nat$ is a generic instance of the function declaration

func `length`:$list(\alpha) \rightarrow nat$

We embed types 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 figure 1 (analogously for **terms**, **atoms** and **goals**). Note that we have no restrictions on the use of types and type variables in a clause in contrast to [MO84], [Smo88a] and similar type systems. For instance, facts with specialized types are allowed in our type system. A **polymorphic equational logic program** is a finite set of polymorphic program clauses. If the left-hand side of a clause has the form $t_1{:}\tau_1 \equiv t_2{:}\tau_2$

(we use the infix notation for '≡'), we call the clause **conditional equation**. Conditional equations define the semantics of functions, whereas other clauses define the semantics of predicates. Since there are no restrictions on the use of predicates and functions in clauses, our language is a genuine amalgamation of a functional and a logic language. The declarative semantics of the language is Horn clause logic with equality, where types are interpreted as subsets of carrier sets. Before we present more details, we give some examples for polymorphic equational logic programs in the next section.

3 Examples

For the examples in this section it is sufficient to have an intuitive idea of the semantics of polymorphic equational logic programs: The equality symbol denotes identity, clauses are implications and type variables are universally quantified over all types.

The first example defines some operations on lists of any type. 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. [CM87]). The function `append` that concatenates both argument lists is defined in a functional style by two equations (with empty conditions). The predicate `member` is defined in a logic programming style. We omit type annotations in program clauses because they can be simply produced by an ML-like type checker [DM82] since the types of all functions and predicates are explicitly given.

type $list/1$
func `[]`: $\rightarrow\ list(\alpha)$
func • : $\alpha,\ list(\alpha)\ \rightarrow\ list(\alpha)$
func `append`: $list(\alpha),\ list(\alpha)\ \rightarrow\ list(\alpha)$
pred `member`: $\alpha,\ list(\alpha)$

clauses:
`append([],L)` ≡ `L` ←
`append([E|R],L)` ≡ `[E|append(R,L)]` ←
`member(E,[E|L])` ←
`member(E,[F|L])` ← `member(E,L)`

Next we show the application of higher-order programming techniques in our framework. As stated above we do not want to leave first-order logic for the sake of efficiency. But we can apply Warren's technique [War82] in our typed framework because we have an unrestricted mechanism of polymorphic types. As shown in the first chapter and in [Han89b], this is *not* possible in other polymorphic type systems for logic programming, e.g., [MO84].

The following example defines the well-known function `map`. The higher-order function `map` takes a function and a list as arguments and applies the function to each element of the list. In order to define the type of `map` we introduce a type constructor '⇒' of arity 2 that denotes the types of functional expressions. The type of `map` is

func `map`: $(\alpha \Rightarrow \beta),\ list(\alpha)\ \rightarrow\ list(\beta)$

(for convenience we use the infix notation for the type constructor '⇒'). The symbols '⇒' and '→' have different meanings: '→' has a fixed interpretation (first-order function, see next section), but '⇒' is a user-defined type and the meaning of operations on objects of this type must be explicitly specified by program clauses. Since we want to give '⇒' a similar meaning as '→', we introduce for each function f of type "$\tau_1 \rightarrow \tau_2$" a corresponding functional constant λf of type "$(\tau_1 \Rightarrow \tau_2)$". The relation between each function f and the functional constant λf is defined by clauses for the function `apply`. Hence we get the following example program for the function `map` (we assume that the basic type int of integers is predefined with operation `+` and the type constructor $list$ is predefined as in the preceding example):

type ⇒/2
func `map` : $(\alpha \Rightarrow \beta),\ list(\alpha)\ \rightarrow\ list(\beta)$
func `double` : $int\ \rightarrow\ int$
func `inc` : $int\ \rightarrow\ int$

```
func λdouble:   →  (int ⇒ int)
func λinc    :   →  (int ⇒ int)
func apply   :  (α ⇒ β),  α  →  β

clauses:
map(F,[]) ≡ [] ←
map(F,[E|L]) ≡ [apply(F,E)|map(F,L)] ←
double(N) ≡ N + N ←
inc(N) ≡ N + 1 ←
apply(λdouble,N) ≡ double(N) ←
apply(λinc,N) ≡ inc(N) ←
```

If we run the program with the initial goal

```
map(λdouble,[1,3,5]) ≡ L
```

(see below for operational semantics), then we get the answer substitution

```
L = [2,4,6]
```

Since our language is a logic language, too, we may ask for functions that map one list into another: For the goal

```
map(F,[2,3,4]) ≡ [3,4,5]
```

we get the only answer substitution

```
F = λinc
```

Since our foundation is first-order logic, the function symbol `map` is semantically not interpreted as a higher-order function. The constant `λinc` is also interpreted as a value and not as a function. But the last clause ensures that in any interpretation which satisfies all program clauses the constant `λinc` and the function `inc` 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 functional language.

A similar solution to logic programming with higher-order functions was proposed by Smolka [Smo86a]. In his language Fresh higher-order functional programming is combined with unification. To avoid the difficulties with higher-order unification, he associates a name with each function and defines equality between functions 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 [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). As shown in the first section, the untyped target program may deduce ill-typed goals. Since we have translated higher-order objects into polymorphic equational 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 the operational semantics ensures that a function or predicate is only called with appropriate arguments at run time ("well-typed programs do not go wrong").

4 Declarative Semantics

We want to give an outline of the declarative semantics. More details can be found in [Han88]. Polymorphic equational logic programs will be interpreted by algebraic structures with a particular sort structure [Poi86]. If there are no type constructors in a program, i.e., all types are monomorphic, then the notions of interpretation and validity for polymorphic programs are equivalent to untyped [Llo87] or many-sorted [GM84] logic.

Variables in untyped logic vary over the carrier set of the interpretation. Consequently, type variables in polymorphic programs vary over all types of the interpretation and typed variables vary over appropriate carrier sets. Hence an interpretation of a polymorphic equational logic program consists of a single-sorted algebra that describes all types in the interpretation and a structure for the derived polymorphic signature. A structure is an interpretation of types as sets, function symbols as operations on these sets and predicate symbols as predicates on these sets where the equality predicate '$\equiv$' is always interpreted as identity on the carrier sets [Pad88]. In the following we will give more precise definitions of these notions. We assume familiarity with notions from algebraic specifications [EM85].

In order to give a precise definition of "universal quantification over all types", we view a specification of basic types and type constructors as a single-sorted signature H (where *type* is the only sort): Each basic type is a constant and each type constructor of arity n is an n-ary function in the signature H. If we denote by X the infinite set of all type variables, the term algebra over H and X, $T_H(X)$, contains all polymorphic type expressions. Hence we can use the notion of an H-algebra for the interpretation of types and the notion of H-homomorphisms from $T_H(X)$ into A to formalize "universal quantification over all types":

Let $\Sigma = (H, Func, Pred)$ be the type declarations of a polymorphic equational logic program, where H is the signature that specifies the basic types and type constructors and $Func$ and $Pred$ are the sets of function and predicate declarations, respectively. Σ is called a **polymorphic signature**. Let A be an H-algebra with carrier set Ty_A. The **polymorphic signature** $\Sigma(A) = (Ty_A, Func_A, Pred_A)$ **derived from** Σ **and** A is defined by

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

This is well-defined since each assignment for variables from X can be uniquely extended to an H-homomorphism from $T_H(X)$ into A and H-homomorphisms are extended to tuples over $T_H(X)$ by componentwise application. An **interpretation** of a polymorphic signature Σ is an H-algebra 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.
3. If '$\equiv$'$:\tau,\tau \in Pred_A$, then $\delta_{\equiv:\tau,\tau} = \{(a,a) \mid a \in S_\tau\}$

If A and A' are H-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 homomorphism between $\Sigma(A)$-structures. 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.

In the following we assume that X is the set of all type variables and V is a set of typed variables. The notion of a **term interpretation** can be defined as usual, where '$\equiv$' denotes the identity relation on terms and all other predicate symbols denote empty relations. By $\boldsymbol{T_\Sigma(X, V)}$ we denote the free term interpretation over X and V where the carrier is the set of all well-typed terms that contain only symbols from Σ, X and 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 $I = (A, S, \delta)$ 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 $x{:}\tau \in V$. Similarly to the many-sorted case, any variable assignment can be uniquely extended to a Σ-homomorphism [Poi86]:

Theorem 1 (Free term structure) *Let I be a Σ-interpretation and v be an assignment for (X, V) in I. There exists a unique Σ-homomorphism from $T_\Sigma(X, V)$ into I that extends v.*

In the following we denote the Σ-homomorphism that extends an assignment v again by v. We are not interested in all interpretations of a polymorphic signature but only in those interpretations that satisfy all clauses of a given polymorphic equational logic program. In order to formalize that we define validity of atoms, goals and clauses relative to a given Σ-interpretation $I = (A, S, \delta)$ (we assume that the polymorphic atoms, goals and clauses contain only type variables from X and typed variables from V):

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

  $\boldsymbol{I, v} \models \boldsymbol{L}$ if $L = p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)$ is a polymorphic 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 $U_\mu((S, \delta)) = (S', \delta')$

  $\boldsymbol{I, v} \models \boldsymbol{G}$ if G is a polymorphic goal with $I, v \models L$ for all $L \in G$

  $\boldsymbol{I, v} \models \boldsymbol{L \leftarrow G}$ if $L \leftarrow G$ is a polymorphic clause where $I, v \models G$ implies $I, v \models L$

- $\boldsymbol{I, V} \models \boldsymbol{\mathcal{F}}$ if $\mathcal{F}$ is a polymorphic atom, goal or clause with $I, v \models \mathcal{F}$ for all variable assignments v for (X, V) in I

Let C be a set of polymorphic program clauses. A Σ-interpretation I is called **model** for (Σ, C) if $I, V_0 \models L \leftarrow G$ for all clauses $L \leftarrow G \in C$, where V_0 is the set of all typed variables occurring in $L \leftarrow G$. A polymorphic 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: $(\boldsymbol{\Sigma}, \boldsymbol{C}, \boldsymbol{V}) \models \boldsymbol{G}$.

This notion of validity is the extension of validity in untyped or many-sorted equational Horn clause logic [Pad88] to the polymorphic case: In untyped or many-sorted 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. We have defined validity relative to a set of variables because 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. An example for such a difference can be found in [Han89a], p. 231. Validity in our sense is equivalent to validity in the sense of untyped logic if the types of variables denote non-empty sets in all interpretations. But a requirement for non-empty carrier sets is not reasonable in our polymorphic framework.

"Typed substitutions" are a combination of substitutions on types and substitutions on well-typed terms: If V, V' are 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: $\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$.

A polymorphic term t' with typed variables V' is called an **instance** of a term t with typed variables V if there is a typed substitution σ from $T_\Sigma(X, V)$ into $T_\Sigma(X, V')$ with $t' = \sigma(t)$. The definition of instances can be extended on atoms, goals and clauses. We omit the simple definitions here.

It can be shown that there exists an initial model for each polymorphic equational logic program. A particular initial model is the quotient of the term interpretation $T_\Sigma(\emptyset, \emptyset)$ and the least congruence generated by the axioms of equality [Han88]. Note that this is a set-theoretic model for logic programs with polymorphic types. This is not a contradiction to the result of Reynolds [Rey84], because Reynolds has a different semantic basis. Reynolds' foundation is the polymorphic typed lambda calculus which allows computations with higher-order functions. Our basis is not the lambda calculus but we use algebraic interpretations. All functions in our specifications are first-order and therefore we can construct a set-theoretic model in contrast to Reynolds. But Reynolds' result shows that the extension of typed logic programs to higher-order functions or predicates is a difficult task. Nevertheless we have seen in section 3 that it is possible to simulate some features of higher-order programming in our framework.

5 Operational semantics

There exists an efficient proof procedure that computes a correct answer if the initial goal is valid w.r.t. the given program. The proof procedure is similar to other equational logic languages, e.g., EQLOG [GM86]: It is the combination of resolution for predicates [Llo87] and conditional narrowing [Hus85] for the computation of functions but with a particular unification procedure for polymorphic terms: In [Han89a] it is shown that the unification of polymorphic terms can be reduced to common first-order unification [Rob65] 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.

Similarly to [HO80], we denote *positions* or *occurrences* in polymorphic terms or goals by sequences of naturals. If π is an occurrence in a polymorphic goal G, then $G[\pi]$ denotes the subterm of G at position π and $G[\pi \leftarrow t]$ denotes the result of replacing the subterm $G[\pi]$ by t. We specify the *basic operational semantics* of our language by describing a computation step from one goal to the next goal. Let $L_1, \ldots, L_n$ be a polymorphic goal.

1. (**Narrowing rule**) If there exist a non-variable occurrence π in L_1, a conditional equation $v \equiv w \leftarrow R_1, \ldots, R_m$ and a most general unifier σ for $L_1[\pi]$ and v, then

$$\sigma(R_1), \ldots, \sigma(R_m), \sigma(L_1)[\pi \leftarrow \sigma(w)], \sigma(L_2), \ldots, \sigma(L_n)$$

is the next goal, otherwise:

2. (**Unification rule**) If $L_1 = v \equiv w$ and there exists a most general unifier σ for v and w, then

$$\sigma(L_2), \ldots, \sigma(L_n)$$

is the next goal, otherwise:

3. (**Restricted resolution rule**) If there exist a program clause $L \leftarrow R_1, \ldots, R_m$ which is not a conditional equation and a most general unifier σ for L_1 and L, then

$$\sigma(R_1), \ldots, \sigma(R_m), \sigma(L_2), \ldots, \sigma(L_n)$$

is the next goal, otherwise: fail.

The soundness and completeness of the operational semantics is proved in [Han88], where the completeness is only stated under the following assumptions: The polymorphic equational program is Church-Rosser (i.e., for each proof with applications of equations in both directions there exists a proof with applications of equations from left to right), the computed answer is irreducible (i.e., the solutions are in normal form) and there are no extra-variables in clauses (i.e., the left-hand-side of a conditional equation contains all variables occurring in the conditional equation). The first two restrictions are standard in equational logic programming [Pad88], but the last restriction is not acceptable from a logic programmer's point of view. Since we are interested in the influence of types into equational logic programming, we have not developed better results for this problem but we may adopt results from other work done in this area. For instance, Bertling and Ganzinger [BG89] have investigated the problem of extra-variables in conditions of conditional equations. They have no restrictions on the variables in conditional equations and give sufficient criteria for the completeness of the narrowing procedure. In some cases it is possible to transform a given program into another one which satisfies the completeness criterion. A particular variant of the Knuth-Bendix completion procedure which considers the particularly chosen rewrite relation is used to transform the programs.

The actual implementation of the language is done in a more efficient way than described above. For instance, the set of all function symbols are partitioned into a set of constructors and a set of defined operators [HH82], where constructors are function symbols that do not appear as the top symbol of the left-hand side in any conditional equation. The operational semantics considers the constructors so that equations between constructors are flattened and an innermost position is chosen in a narrowing step (this strategy is only complete under additional restrictions, see [Fri85]).

Since we have an unrestricted type system, the unification procedure has to consider the types of the polymorphic terms. Hence the polymorphic unification is more complex and less efficient than the unification in untyped logic languages. But it can be shown [Han88] that it is possible to omit type annotations at run time if the program satisfies particular syntactic conditions (compare [Han89a] and [Han89b] for the case of polymorphic logic programs without equality). For instance, if the left-hand sides of all clauses have a most general type w.r.t. the type declarations, then all types can be omitted at run time. Better optimization techniques are a topic for future research. On the other hand, type information may be useful to reduce the search space in the computation process [HV87]. Therefore the overall goal is not to omit all type annotations at run time but only those annotations that are unnecessary for correct computation. We have seen in the introduction that some logic programming techniques cannot be applied in a type secure way if all type annotations are omitted at run time.

Resolution combined with conditional narrowing is only a complete proof method for validity if all possible derivations are simultaneously computed. This is very expensive and therefore it is recommendable to use a backtracking strategy as in Prolog with the consequence that completeness is lost in general. The interpreter of our language is implemented in Prolog and uses the Prolog backtracking strategy for different derivations. Polymorphic terms are mapped into Prolog terms and therefore the built-in unification can be used to unify polymorphic terms. A lot of equational logic programs have been successfully executed by the interpreter.

6 Conclusions

We have presented an equational logic language with a polymorphic type system. The language integrates two interesting programming styles in a type secure way: functional and logic programming. In contrast to other proposals for polymorphically typed logic programming languages, it is possible to apply higher-order programming techniques in our framework. The language has a well-defined semantics and programs can be efficiently executed if the same drawbacks as in Prolog (incompleteness because of infinite derivations) are accepted. The language is explicitly typed, i.e., each syntactic unit is annotated with a type expression, but the implementation contains a type inference procedure so that the programmer can omit all type annotations in clauses. Further work remains to be done in the area of run-time optimizations with the aim to reduce the cases where type information is required for correct unification.

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.

[BG86] P.G. 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.

[BG89] H. Bertling and H. Ganzinger. Completion-Time Optimization of Rewrite-Time Goal Solving. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 45–58. Springer LNCS 355, 1989.

[BMS80] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. HOPE: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pp. 136–143. ACM, 1980.

[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.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

[DM82] 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.

[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.

[Fri85] L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 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.

[Han88] M. Hanus. *Horn Clause Specifications with Polymorphic Types.* Dissertation, FB Informatik, Univ. Dortmund, 1988.

[Han89a] 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.*

[Han89b] M. Hanus. Polymorphic Higher-Order Programming in Prolog. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 382–397. MIT Press, 1989.

[HH82] G. Huet and J.-M. Hullot. Proofs by Induction of Equational Theories with Constructors. *Journal of Computer and System Sciences 25*, pp. 239–266, 1982.

[HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, 1986.

[HO80] 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.

[Hus85] H. Hussmann. Unification in Conditional-Equational Theories. In *Proc. EUROCAL '85*, pp. 543–553. Springer LNCS 204, 1985.

[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.

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

[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.

[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.

[Nai87] L. Naish. Specification = Program + Types. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, pp. 326–339. Springer LNCS 287, 1987.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1988.

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

[Rey84] J.C. Reynolds. Polymorphism is not set-theoretic. In *Proc. of the Int. Symp. on the Semantics of Data Types, Sophia-Antipolis*, pp. 145–156. Springer LNCS 173, 1984.

[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.

[Smo86a] 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.

[Smo86b] G. Smolka. Order-Sorted Horn Logic: Semantics and Deduction. SEKI Report SR-86-17, FB Informatik, Univ. Kaiserslautern, 1986.

[Smo88a] 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. Springer LNCS 343, 1988.

[Smo88b] G. Smolka. TEL (Version 0.9) Report and User Manual. SEKI Report SR-87-11, FB Informatik, Univ. Kaiserslautern, 1988.

[Tur85] D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, pp. 1–16. Springer LNCS 201, 1985.

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

[XW88] 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.

[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.