Theoretical Computer Science 89 (1991) 63-106
Elsevier

Horn clause programs with polymorphic types: semantics and resolution

Michael Hanus

Fachbereich Informatik, Universität Dortmund, W-4600 Dortmund 50, Germany

Abstract

Hanus, M., Horn clause programs with polymorphic types: semantics and resolution, Theoretical Computer Science 89 (1991) 63-106.

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 [24]. 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 nonempty 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 a type, e.g., types represent subsets of carrier sets of interpretations. Goguen, Meseguer [11] and Smolka [34] have proposed order-sorted type systems for Horn clause logic (with equality). Each type represents a subset of the carrier set in the interpretation, and the ordering of types implies a subset relation on the corresponding sets. Aït-Kaci and Nasr [1] have proposed a logic language with subtypes and inheritance 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 a type. Mishra [25] and Zobel [41] have presented type inference systems for detecting programming errors in a given Prolog program. Kanamori, Horiuchi [21] and Kluźniak [22] have developed algorithms for inferring types of variables in a Prolog program. Yardeni and Shapiro [40] have presented a type-checking algorithm where types are regular sets of ground atoms.

Since pure Prolog is a declarative language, each extension should have a declarative meaning. Hence we will define a typed Horn clause logic in a model-theoretic way and then we investigate the operational mechanisms for this kind of logic.

The important question is: What is an adequate type system for logic programming? As shown above, there are several proposals for type systems for logic programming, and these type systems offer different flexibilities from a programmer's point of view. For instance, the Pascal-like type system of Turbo-Prolog is comparable to many-sorted Horn logic [30], but this type system is too restricted for many applications [14]. Prolog is a very flexible language because the programmer can simply define predicates (e.g., see the definition of append) which are applicable

to a number of different types, i.e., classes of objects like lists of integers, lists of characters etc. Therefore we are interested in a *polymorphic type system* where type declarations may contain type variables that are universally quantified over all types [8]. Mycroft and O'Keefe [28] 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 [7] 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 [35] 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 [28], 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 [28] 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

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

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 • that concatenates an element with a list of the same type (throughout this paper we use the Prolog notation for lists [6]). The type declarations for these two functions are

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

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)
```

With these type declarations the goal `append([1,2],[3,4],[1,2,3,4])` is well-typed and can be proved to be true, whereas the goal `append([],3,3)` is rejected since the second and third arguments are not lists. In contrast to other polymorphic type systems for logic programming our type system allows a useful logic programming technique: optimization of the resolution process by *lemma generation*. In untyped logic programming it is possible to add a new fact L to a program without changing the program semantics if L is a logical consequence of the program. The new fact L can be used to obtain shorter proofs for subsequent goals that include L. For instance, the specialized clause

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

can be added to the above append-program. This is also possible in our typed logic language, but other polymorphic type systems for logic programming reject this clause because they require the argument types in clause heads to be equivalent (equal up to type variable renaming) to the type declaration of the predicate [28]. But the arguments of the head of the last clause have type $list(int)$. Hence this is not a well-typed clause in the sense of [28] 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.

2. Polymorphic logic programs

We use notions from algebraic specifications [13] 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 $O = (O_{w,s} \mid w \in S^*, s \in S)$. 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 $S_A = (S_{A,s} \mid s \in S)$ and an operation $o_A : S_{A,s_1}, \ldots, S_{A,s_n} \to S_{A,s} \in O_A$ for any $o : s_1, \ldots, s_n \to s \in O$. A set of *$\Sigma$-variables* is an S-sorted set $V = (V_s \mid s \in S)$. 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 $c : \to s \in O$, and $o(t_1, \ldots, t_n) \in T_{\Sigma,s}(V)$ for all $o : s_1, \ldots, s_n \to s \in O$ $(n > 0)$ 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 $a : V \to S_A$ with $a(x) \in S_{A,s}$ for all variables $x \in V_s$ (more precisely, it is a family of mappings $(a_s : V_s \to S_{A,s} \mid s \in S)$). 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) $h : S_A \to S_B$ with the properties $h_s(c_A) = c_B$ for all $c : \to s \in O$ 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 $o : s_1, \ldots, s_n \to s \in O$ $(n > 0)$ 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 $h : \to type$ are called *basic types* (with arity 0), whereas operators of the form $h : type^n \to type$ 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 $\sigma : T_H(X) \to T_H(X)$. $TS(H, X)$ 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 $(H, Func, Pred)$ 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., [28, 35].

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, *pred*2/2
func z : $\to$ *nat*
func s : *nat* $\to$ *nat*
func [ ] : $\to$ *list*(α)
func • : α, *list*(α) $\to$ *list*(α)

func pred_inc: → *pred*2(*nat*, *nat*)
pred inc : *nat*, *nat*
pred map : *pred*2(α, β), *list*(α), *list*(β)
pred apply2 : *pred*2(α, β), α, β

The predicate apply2 will be interpreted like call in Prolog: If the first argument has type *pred*2(α, β) 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 *pred*2(*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 *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 [5], 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 (Σ, 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 $(i = 1, \ldots, n)$. $Term_\Sigma(X, V)$ denotes the $T_H(X)$-sorted set of all (Σ, X, V)-terms. A *ground term* is a term from the set $Term_\Sigma(X, \emptyset)$.

Different occurrences of a function in a term may have different types which shows the polymorphism in our framework. 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* → *bool* **var** X: α

then the terms f(X:α, X:α):*bool* and f(X:*int*, X:*bool*):*bool* are 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* $\in Term_{\Sigma,bool}(\{\alpha\}, \emptyset)$ 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. For convenience we sometimes omit the curly brackets around a goal and we identify a goal containing only one atom with that atom. 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 in practice the user may omit the type annotations in the clauses of a polymorphic logic program and the most general type of a term that satisfies the uniqueness requirement can be automatically computed. 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 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 [28] since apply2 has the declared type "*pred*2(α, β), α, β" but is used in the clause head with the specialized type "*pred*2(*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` or `or`, or 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`: → *bool*
func `false`: → *bool*
func `and`: *bool*, *bool* → *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 [28] because of the type of the function `equal` (in their type system each type variable occurring in the argument type of a function must also occur in the result type [29]).

3. Semantics of polymorphic logic programs

3.1. Validity and models

We use algebraic structures for the interpretation of polymorphic logic programs [31]. Variables in untyped logic vary over the carrier set of the interpretation. Consequently, type variables in polymorphic logic programs 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

$$Func_A := \{f:\sigma(\tau_f) \mid f:\tau_f \in Func,\ \sigma : X \to Ty_A \text{ is a type variable assignment}\},$$

$$Pred_A := \{p:\sigma(\tau_p) \mid p:\tau_p \in Pred,\ \sigma : X \to Ty_A \text{ is a type variable assignment}\}.$$

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.

Hence polymorphic functions and predicates are interpreted as families of functions and predicates on the given types. In order to compare different interpretations, we define homomorphisms between them. At first, we define $\Sigma(A)$-homomorphisms to compare different $\Sigma(A)$-structures: Let $A=(Ty_A, Ht_A)$ be an H-type algebra and (S, δ), (S', δ') be $\Sigma(A)$-structures. A $\Sigma(A)$-*homomorphism* h from (S, δ) into (S', δ') is a family of functions $(h_\tau \mid \tau \in Ty_A)$ with

(1) $h_\tau: S_\tau \to S'_\tau$.

(2) If $f:\tau_f \in Func_A$ with $\tau_f = \tau_1, \ldots, \tau_n \to \tau$ $(n \geq 0)$ and $a_i \in S_{\tau_i}$ $(i=1, \ldots, n)$, then $h_\tau(\delta_{f:\tau_f}(a_1, \ldots, a_n)) = \delta'_{f:\tau_f}(h_{\tau_1}(a_1), \ldots, h_{\tau_n}(a_n))$.

(3) If $p:\tau_p \in Pred_A$ with $\tau_p = \tau_1, \ldots, \tau_n$ $(n \geq 0)$ and $(a_1, \ldots, a_n) \in \delta_{p:\tau_p}$, then $(h_{\tau_1}(a_1), \ldots, h_{\tau_n}(a_n)) \in \delta'_{p:\tau_p}$.

If it is clear from the context we omit the indices τ in the functions h_τ. Note that the composition of two $\Sigma(A)$-homomorphisms is again a $\Sigma(A)$-homomorphism. The class of all $\Sigma(A)$-structures together with the $\Sigma(A)$-homomorphisms is a category [9]. We denote this category by $Cat_{\Sigma(A)}$.

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* $U_\sigma: Cat_{\Sigma(A')} \to Cat_{\Sigma(A)}$ (for details, see [9]). 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* in a class of Σ-interpretations $\mathscr{C}$ iff for all Σ-interpretations $(A', S', \delta') \in \mathscr{C}$ 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 $Term_\Sigma(X, V)$ and all predicate symbols are interpreted as empty sets. 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 $x:\tau \in V$.

In many-sorted algebra any variable assignment can be uniquely extended to a homomorphism. This is also true in the polymorphic case [31].

Lemma 3.1 (Free term structure). *Let (A, S, δ) be a Σ-interpretation and (μ, val) be an assignmentt for (X, V) in (A, S, δ). There exists a unique Σ-homomorphism (σ, h) from $T_\Sigma(X, V)$ into (A, S, δ) with the properties $\sigma(\alpha) = \mu(\alpha)$ for all $\alpha \in X$ and $h(v) = val(v)$ for all $v \in V$.*

As a special case ($X = V = \emptyset$) the lemma shows that every ground term with monomorphic types corresponds to a unique value in a given Σ-interpretation. Generally, any variable assignment (μ, val) can be extended to a Σ-homomorphism in a unique way. In the following we denote that Σ-homomorphism again by (μ, val).

We are not interested in all interpretations of a polymorphic signature but only in those interpretations that satisfy 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.
  $I, v \models L$ 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 $U_\mu((S, \delta)) = (S', \delta')$.
  $I, v \models G$ if G is a (Σ, X, V)-goal with $I, v \models L$ for all $L \in G$.
  $I, v \models L \leftarrow G$ if $L \leftarrow G$ is a (Σ, X, V)-clause where $I, v \models G$ implies $I, v \models L$.
- $I, V \models \mathcal{F}$ if $\mathcal{F}$ is a (Σ, X, V)-atom, -goal or -clause with $I, v \models \mathcal{F}$ 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 $(\Sigma, C, V) \models G$.

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 [10]. Validity relative to variables is different from validity in the sense of untyped logic. The following example shows such a difference.

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

$$\mathrm{p}(\mathrm{x}: void) \leftarrow$$
$$\mathrm{q}(0: zero) \leftarrow \mathrm{p}(\mathrm{x}: void)$$

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

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

Hence $\mathrm{q}(0: zero)$ is valid in M relative to $\{\mathrm{x}: void\}$, but $\mathrm{q}(0: zero)$ 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 denote nonempty sets in all interpretations. But a requirement for nonempty carrier sets is not reasonable. For a more detailed discussion of this subject compare [10].

"Typed substitutions" are a combination of type substitutions and substitutions on well-typed terms: If $V, V' \subseteq Var_{\Sigma,X}$ 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 $\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$. $Sub_\Sigma(X, V, V')$ denotes the class of all typed substitutions from $T_\Sigma(X, V)$ into $T_\Sigma(X, V')$ and $id_{X,V} \in Sub_\Sigma(X, V, V)$ denotes the identity on $T_\Sigma(X, V)$. $\iota dom(\sigma) := \{\alpha \in X \mid \sigma(\alpha) \neq \alpha\}$ is the *type domain* of a typed substitution σ. A typed substitution keeps the set of type variables X but may change the set of typed variables because the types of the variables influence validity (see above). Sometimes we represent typed substitutions by sets: The set

$$\sigma = \{\alpha / nat, x{:}\alpha / 0{:}nat\}$$

represents a typed substitution that replaces the type variable α by the monomorphic type *nat* and the typed variable $x{:}\alpha$ by the ground term $0{:}nat$. Hence the result of applying σ to the atom $p(x{:}\alpha, y{:}\alpha)$ is the atom $p(0{:}nat, y{:}nat)$. The following lemma shows a relationship between variable assignments and typed substitutions w.r.t. validity.

Lemma 3.2. *Let I be a Σ-interpretation, G be a (Σ, X, V)-goal, $\sigma \in Sub_\Sigma(X, V, V')$ and v be a variable assignment for (X, V') in I. Then $I, v \models \sigma(G)$ iff $I, v \circ \sigma \models G$.*

Proof. Let $G, \sigma, v = (\mu, val)$ and $I = (A, S, \delta)$ be given. The composition $v' := v \circ \sigma$ is defined by $v' = (\mu', val')$ with $\mu'(\alpha) = \mu(\sigma(\alpha))$ for all $\alpha \in X$ and

$$val'_\tau(x{:}\tau) = (U_\sigma(val) \circ \sigma)_\tau(x{:}\tau) = val_{\sigma(\tau)}(\sigma(x{:}\tau))$$

for all $x{:}\tau \in V$. Thus v' is a variable assignment for (X, V) in I. Let $p(\ldots t_i{:}\tau_i \ldots) \in G$. Then

$$\begin{aligned} I, v \models \sigma(p(\ldots t_i{:}\tau_i \ldots)) &\Leftrightarrow I, v \models p(\ldots \sigma(t_i{:}\tau_i) \ldots) \\ &\Leftrightarrow (\ldots val_{\sigma(\tau_i)}(\sigma(t_i{:}\tau_i)) \ldots) \in \delta_{p:\ldots\mu(\sigma(\tau_i))\ldots} \\ &\Leftrightarrow (\ldots val'_{\tau_i}(t_i{:}\tau_i) \ldots) \in \delta_{p:\ldots\mu'(\tau_i)\ldots} \\ &\Leftrightarrow I, v' \models p(\ldots t_i{:}\tau_i \ldots). \end{aligned}$$

This proves the lemma. □

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

$$I, V \models L \leftarrow G \Leftrightarrow I, V' \models \sigma(L) \leftarrow \sigma(G) \text{ for all } \sigma \in Sub_\Sigma(X, V, V').$$

Proof. The direction "$\Leftarrow$" is trivial if we use the identity on $T_\Sigma(X, V)$ for the typed substitution σ. Let $I, V \models L \leftarrow G$ and $\sigma \in Sub_\Sigma(X, V, V')$ be a typed substitution. We have to show $I, V' \models \sigma(L) \leftarrow \sigma(G)$. Let v be a variable assignment for (X, V') in I with $I, v \models \sigma(G)$ (if there exists no such variable assignment, $I, V' \models \sigma(L) \leftarrow \sigma(G)$ is trivially true). Lemma 3.2 yields $I, v \circ \sigma \models G$. This implies $I, v \circ \sigma \models L$ since $I, V \models L \leftarrow G$. Again by Lemma 3.2, it follows $I, v \models \sigma(L)$. □

Along with a set of Σ-clauses C we define the *set of instantiated clauses* $\hat{C}$ as follows:

$$\hat{C} := \{L \leftarrow G \mid L \leftarrow G \text{ is an instance of a clause from } C\}.$$

The set $\hat{C}$ contains all clauses which are obtained from clauses in C by substituting type expressions for type variables and well-typed terms for typed variables.

Corollary 3.4. *A Σ-interpretation is a model for (Σ, C) iff it is a model for $(\Sigma, \hat{C})$.*

Proof. The theorem follows by definition of $\hat{C}$ and Lemma 3.3. □

3.2. Construction of an initial model

In this section we show the existence of an initial model for every polymorphic logic program. The construction is very similar to the untyped case [24]. A *Herbrand interpretation* (model) for a polymorphic logic program (Σ, C) is an interpretation (model) where the carrier is a term interpretation with ground terms and monomorphic types. Hence different Herbrand interpretations only differ in the denotation of the predicate symbols. Therefore any Herbrand interpretation $\mathcal{H} = (T_H, Term_\Sigma(\emptyset, \emptyset), \delta)$ can be characterized by the set

$$M_\mathcal{H} = \{p(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in \delta_{p:\tau_p}, p:\tau_p \in Pred_{T_H}\}.$$

Lemma 3.5. *Let (Σ, C) be a polymorphic logic program and (A, S, δ) be a model for (Σ, C). Then there exists a Herbrand model for (Σ, C).*

Proof. By Lemma 3.1 (free term structure), there exists a unique Σ-homomorphism (σ, h) from $T_\Sigma(\emptyset, \emptyset)$ into (A, S, δ). We define the following Herbrand interpretation:

$$M = \{p(t_1:\tau_1, \ldots, t_n:\tau_n) \mid p:\tau_1, \ldots, \tau_n \in Pred_{T_H}, t_i:\tau_i \in Term_\Sigma(\emptyset, \emptyset) \\ (i = 1, \ldots, n), (h_{\tau_1}(t_1:\tau_1), \ldots, h_{\tau_n}(t_n:\tau_n)) \in \delta_{p:\sigma(\tau_1),\ldots,\sigma(\tau_n)}\}.$$

It is straightforward to show that M is a model for (Σ, C). □

Next we show that Herbrand models are sufficient for proving the validity of monomorphic ground atoms.

Lemma 3.6. *Let (Σ, C) be a polymorphic logic program and L be a $(\Sigma, \emptyset, \emptyset)$-atom. If L is valid in every Herbrand model, then L is valid in any model.*

Proof. Let $M = (A, S, \delta)$ be a model for (Σ, C). By Lemma 3.5, there exists a Herbrand model $M_{\mathscr{H}}$ for (Σ, C). By Lemma 3.1, there exists a unique Σ-homomorphism (σ, h) from $T_\Sigma(\emptyset, \emptyset)$ into M. $L = p(t_1 : \tau_1, \ldots, t_n : \tau_n)$ is valid in $M_{\mathscr{H}}$. Therefore $p(t_1 : \tau_1, \ldots, t_n : \tau_n) \in M_{\mathscr{H}}$. By construction in the proof of Lemma 3.5, $(h_{\tau_1}(t_1 : \tau_1), \ldots, h_{\tau_n}(t_n : \tau_n)) \in \delta_{p:\sigma(\tau_1),\ldots,\sigma(\tau_n)}$. Thus $M, (\sigma, h) \models L$. Since (σ, h) is the unique variable assignment, L is valid in M. □

It is straightforward to show that the intersection of a nonempty set of Herbrand models is again a Herbrand model. Hence the set

$$M_{\mathscr{I}} := \bigcap \{M_i \mid M_i \text{ is a Herbrand model}\}$$

is a Herbrand model, because every polymorphic logic program (Σ, C) has at least one model

$$M_\Sigma = \{p(t_1 : \tau_1, \ldots, t_n : \tau_n) \mid p : \tau_1, \ldots, \tau_n \in \mathit{Pred}_{T_H}, t_i : \tau_i \in \mathit{Term}_\Sigma(\emptyset, \emptyset)\ (i = 1, \ldots, n)\}.$$

The model $M_{\mathscr{I}}$ is called the *least Herbrand model*. It is an initial model for (Σ, C).

Theorem 3.7 (Initial model). *Let (Σ, C) be a polymorphic logic program. Then the least Herbrand model $M_{\mathscr{I}}$ is an initial model for (Σ, C), i.e., for each model M for (Σ, C) there exists a unique Σ-homomorphism from $M_{\mathscr{I}}$ into M.*

Proof. Let $M = (A, S, \delta)$ be a model for (Σ, C) and $M_{\mathscr{I}} = (T_H, \mathit{Term}_\Sigma(\emptyset, \emptyset), \delta')$. By Lemma 3.1, there exists a unique Σ-homomorphism (σ, h) from $T_\Sigma(\emptyset, \emptyset)$ into M. In order to show that (σ, h) is a Σ-homomorphism from $M_{\mathscr{I}}$ into M, we have to prove the following condition for Σ-homomorphisms (we assume $\tau_p = \tau_1, \ldots, \tau_n$):

$$(t_1 : \tau_1, \ldots, t_n : \tau_n) \in \delta'_{p:\tau_p} \Rightarrow (h_{\tau_1}(t_1 : \tau_1), \ldots, h_{\tau_n}(t_n : \tau_n)) \in \delta_{p:\sigma(\tau_p)}.$$

Let $(t_1 : \tau_1, \ldots, t_n : \tau_n) \in \delta'_{p:\tau_p}$. Then $M_{\mathscr{I}}, \emptyset \models L$, where $L = p(t_1 : \tau_1, \ldots, t_n : \tau_n)$. Therefore L is valid in all Herbrand models. By Lemma 3.6, L is valid in all models and in particular, L is valid in M. Hence $M, (\sigma, h) \models L$ and $(h_{\tau_1}(t_1 : \tau_1), \ldots, h_{\tau_n}(t_n : \tau_n)) \subset \delta_{p:\sigma(\tau_p)}$. □

3.3. Fixpoint characterization of the least Herbrand model

We want to characterize the least Herbrand model by a fixpoint of a monotonic function, which will be used for proving a completeness theorem for our polymorphic logic. For this purpose we need some results about fixpoints in complete lattices. We skip the necessary definitions here (keywords: partial order, least upper bound *lub*, greatest lower bound *glb*, complete lattice, monotonic and continuous mappings, directed subsets and $f \uparrow \omega$) and refer to [24] for details. We only cite two important results.

Theorem 3.8 (Knaster-Tarski). *Let S be a complete lattice and $f: S \to S$ be a monotonic mapping. Then f has a least fixpoint $lfp(f)$ with $lfp(f) = glb(\{x \mid f(x) = x\}) = glb(\{x \mid f(x) \leq x\})$.*

Theorem 3.9 (Kleene). *Let S be a complete lattice and $f: S \to S$ be a continuous mapping. Then $f\uparrow\omega = lfp(f)$.*

In the following we apply these results to Herbrand interpretations. The next lemma is straightforward to show.

Lemma 3.10. *Let Σ be a polymorphic signature. The set 2^{M_Σ} of all Herbrand interpretations of Σ is a complete lattice with the set inclusion $\subseteq$ as a partial order. The bottom element is $\emptyset$, the top element is M_Σ. For a subset $M \subseteq 2^{M_\Sigma}$ the least upper bound is $lub(M) = \bigcup \{M_i \mid M_i \in M\}$ and the greatest lower bound is $glb(M) = \bigcap \{M_i \mid M_i \in M\}$.*

The mappping $T_{\Sigma,C}$ is a transformation on Herbrand interpretations and was defined for the untyped case in [37]: For each polymorphic logic program (Σ, C) we define a mapping $T_{\Sigma,C}: 2^{M_\Sigma} \to 2^{M_\Sigma}$ on Herbrand interpretations as follows:

$$T_{\Sigma,C}(M) := \{L \in M_\Sigma \mid \exists \text{ an instance } L \leftarrow G \text{ of a clause from } C \text{ with } G \subseteq M\}$$

for all $M \in 2^{M_\Sigma}$. We will give a characterization of the least Herbrand model by the mapping $T_{\Sigma,C}$. The next lemma can be proved in the same way as in untyped Horn logic.

Lemma 3.11. *Let (Σ, C) be a polymorphic logic program. Then $T_{\Sigma,C}$ is continuous (and monotonic).*

Lemma 3.12. *Let (Σ, C) be a polymorphic logic program and I be a Herbrand interpretation of Σ. Then I is a model for (Σ, C) iff $T_{\Sigma,C}(I) \subseteq I$.*

Proof. "$\Rightarrow$" Let I be a model for (Σ, C) and $L \in T_{\Sigma,C}(I)$. Then there exists an instance $L \leftarrow G$ of a clause from C with $G \subseteq I$. By Corollary 3.4, I is a model for $(\Sigma, \hat{C})$ and therefore $L \in I$.

"$\Leftarrow$" Let $T_{\Sigma,C}(I) \subseteq I$, $L \leftarrow G \in C$, $V = var(L \leftarrow G)$ and $v = (\mu, val)$ be an arbitrary variable assignment for (X, V) in $T_\Sigma(\emptyset, \emptyset)$ (if there exists no such variable assignment, then $I, V \models L \leftarrow G$ is trivially true). If $I, v \models G$, then $val(G) \subseteq I$, and therefore (because $val(L) \leftarrow val(G) \in \hat{C}$) $val(L) \in T_{\Sigma,C}(I)$, i.e., $I, v \models L$. Thus $L \leftarrow G$ is valid in I. □

Theorem 3.13 (Fixpoint characterization of the least Herbrand model). *Let (Σ, C) be a polymorphic logic program. Then $M_\mathscr{P} = lfp(T_{\Sigma,C}) = T_{\Sigma,C}\uparrow\omega$.*

Proof.

$$\begin{aligned}
M_\mathscr{P} &= \bigcap \{M_j \mid M_j \text{ is a Herbrand model}\} && \text{(by definition)}\\
&= glb(\{M_j \mid M_j \text{ is a Herbrand model}\}) &&\\
&= glb(\{M_j \mid T_{\Sigma,C}(M_j) \subseteq M_j\}) && \text{(by Lemma 3.12)}\\
&= lfp(T_{\Sigma,C}) && \text{(by Theorem 3.8)}\\
&= T_{\Sigma,C}\uparrow\omega && \text{(by Theorem 3.9).}
\end{aligned}$$

□

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 3.1 is given, then the following sequence is a deduction in the polymorphic Horn clause calculus:

$$(\Sigma, C, \{\mathsf{x}:\mathit{void}\}) \vdash \mathsf{p}(\mathsf{x}:\mathit{void}) \leftarrow$$
$$(\Sigma, C, \{\mathsf{x}:\mathit{void}\}) \vdash \mathsf{q}(\mathsf{0}:\mathit{zero}) \leftarrow \mathsf{p}(\mathsf{x}:\mathit{void})$$
$$(\Sigma, C, \{\mathsf{x}:\mathit{void}\}) \vdash \mathsf{q}(\mathsf{0}:\mathit{zero}) \leftarrow$$

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

The soundness of the polymorphic Horn clause calculus can be shown by proving the soundness of each inference rule.

Theorem 4.1 (Soundness of deduction). *Let C be a set of Σ-clauses, $V \subseteq Var_{\Sigma,X}$ and L be a (Σ, X, V)-atom. If $(\Sigma, C, V) \vdash L \leftarrow \emptyset$, then $(\Sigma, C, V) \models L$.*

Proof. Let M be a model for (Σ, C). By induction on the length of a deduction we show that $M, V_i \models L_i \leftarrow G_i$ for each element $(\Sigma, C, V_i) \vdash L_i \leftarrow G_i$ in a deduction for $L \leftarrow \emptyset$.

(1) Axioms: If $L_i \leftarrow G_i \in C$, then $M, var(L_i \leftarrow G_i) \models L_i \leftarrow G_i$. Let $v = (\mu, val)$ be a variable assignment for (X, V_i) in M (if there exists no such variable assignment, then $M, V_i \models L_i \leftarrow G_i$ is trivially true). Let $v' = (\mu, val|_{var(L_i \leftarrow G_i)})$ be the restriction of v to $(X, var(L_i \leftarrow G_i))$. Then $M, v' \models L_i \leftarrow G_i$ is true and therefore $M, v \models L_i \leftarrow G_i$ is also true.

(2) Substitution rule: Let $\sigma \in Sub_\Sigma(X, V_i, V_i')$ be a typed substitution and v' be a variable assignment for (X, V_i') in M (if there exists no such variable assignment, then $M, V_i' \models \sigma(L_i) \leftarrow \sigma(G_i)$ is trivially true). $v := v' \circ \sigma$ is a variable assignment for (X, V_i) in M. By induction hypothesis, $M, v \models L_i \leftarrow G_i$. Suppose now that $M, v' \models \sigma(G_i)$. Lemma 3.2 yields $M, v \models G_i$. This implies $M, v \models L_i$ and, again by Lemma 3.2, $M, v' \models \sigma(L_i)$. Therefore, $M, v' \models \sigma(L_i) \leftarrow \sigma(G_i)$.

(3) Cut rule: Let $(\Sigma, C, V_i) \vdash L_i \leftarrow G_i \cup \{L_j\}$ and $(\Sigma, C, V_j) \vdash L_j \leftarrow G_j$ be elements of the deduction with $V_i = V_j$. Let v be a variable assignment for (X, V_i) in M with

$M, v \models G_i \cup G_j$ (if there exists no such variable assignment, then $M, V_i \models L_i \leftarrow G_i \cup G_j$ is trivially true). By induction hypothesis, $M, v \models L_i \leftarrow G_i \cup \{L_j\}$ and $M, v \models L_j \leftarrow G_j$. Since $M, v \models G_j$, we obtain $M, v \models L_j$. On the other hand, $M, v \models G_i$. Hence $M, v \models G_i \cup \{L_j\}$ and $M, v \models L_i$. Therefore, $M, v \models L_i \leftarrow G_i \cup G_j$, as required. □

Similarly to [2], we prove the completeness of deduction by using the fixpoint characterization of the least Herbrand model. At first, we state a completeness result for monomorphic ground atoms.

Lemma 4.2. *Let C be a set of Σ-clauses and L be a $(\Sigma, \emptyset, \emptyset)$-atom. If $(\Sigma, C, \emptyset) \models L$, then $(\Sigma, C, \emptyset) \vdash L \leftarrow \emptyset$.*

Proof. If $(\Sigma, C, \emptyset) \models L$, then L is valid in every model for (Σ, C), in particular, $L \in M_\emptyset$. Theorem 3.13 yields $L \in T_{\Sigma,C} \uparrow \omega$ and therefore $L \in T_{\Sigma,C} \uparrow n$ for some finite n. We prove the lemma by induction on n.

$n = 1$: By definition of $T_{\Sigma,C}$, there exist $L' \leftarrow \emptyset \in C$ and $\sigma \in Sub_\Sigma(X, V, \emptyset)$ with $L = \sigma(L')$. By an application of an axiom and the substitution rule, we obtain $(\Sigma, C, \emptyset) \vdash L \leftarrow \emptyset$.

$n > 1$: By definition of $T_{\Sigma,C}$, there exist $L' \leftarrow G' \in C$ and $\sigma \in Sub_\Sigma(X, V, \emptyset)$ with $L = \sigma(L')$ and $\sigma(G') \subseteq T_{\Sigma,C} \uparrow n - 1$. By an application of an axiom and the substitution rule, we obtain $(\Sigma, C, \emptyset) \vdash L \leftarrow \sigma(G')$. Let $\sigma(G') = \{L_1, \ldots, L_k\}$. By induction hypothesis, $(\Sigma, C, \emptyset) \vdash L_i \leftarrow \emptyset$ for $i = 1, \ldots, k$. By k applications of the cut rule, we obtain $(\Sigma, C, \emptyset) \vdash L \leftarrow \emptyset$. □

To extend the completeness result to Σ-atoms with type variables and typed variables, we need the following lemma which states that validity is invariant under the extension of signatures.

Lemma 4.3 (Extended signatures). *Let C be a set of Σ-clauses, $\tau_1, \ldots, \tau_k$ be new basic types or type constructors and $\Sigma' = (H', Func', Pred)$ be an extended polymorphic signature with $H' = H \cup \{\tau_1, \ldots, \tau_k\}$ and $Func \subseteq Func'$. If $V \subseteq Var_{\Sigma,X}$, then the following implication is true for any (Σ, X, V)-clause $L \leftarrow G$:*

$$(\Sigma, C, V) \models L \leftarrow G \Rightarrow (\Sigma', C, V) \models L \leftarrow G.$$

Proof. We assume $(\Sigma, C, V) \models L \leftarrow G$. Let $M' = (A', S', \delta')$ be a model for (Σ', C) with $(A' = Ty_{A'}, Ht_{A'})$. Let $A := (Ty_A, Ht_A)$ with $Ty_A := Ty_{A'}$ and $Ht_A := \{h_{A'} \mid h \in Ht\}$. A is an H-type algebra. Let $S := S'$, $\delta_{f:\tau_f} := \delta'_{f:\tau_f}$ for all $f:\tau_f \in Func_A$ and $\delta_{p:\tau_p} := \delta'_{p:\tau_p}$ for all $p:\tau_p \in Pred_A$. (S, δ) is a $\Sigma(A)$-structure. $M = (A, S, \delta)$ is a Σ-interpretation and all clauses from C are valid in M. Therefore M is a model for (Σ, C) and $M, V \models L \leftarrow G$ is true. Let v be a variable assignment for (X, V) in M'. Since $V \subseteq Var_{\Sigma,X}$, $Ty_A = Ty_{A'}$ and $S = S'$, v is also a variable assignment for (X, V) in M. Therefore $M, v \models L \leftarrow G$. Since $\delta_{p:\tau_p} = \delta'_{p:\tau_p}$ for all $p:\tau_p \in Pred_A$, it follows $M', v \models L \leftarrow G$. Hence $M', V \models L \leftarrow G$ is true. □

Now we can state the completeness of the polymorphic Horn clause calculus.

Theorem 4.4 (Completeness of deduction). *Let C be a set of Σ-clauses, $V \subseteq Var_{\Sigma,X}$ be a finite set of typed variables and L be a (Σ, X, V)-atom. If $(\Sigma, C, V) \models L$, then $(\Sigma, C, V) \vdash L \leftarrow \emptyset$.*

Proof. Let $tvar(L) \cup tvar(V) = \{\alpha_1, \ldots, \alpha_m\}$ and $V = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. Let $\gamma_1, \ldots, \gamma_m$ be new basic types and $c_1, \ldots, c_n$ be new constant symbols. Let $\Sigma' = (H \cup \{\gamma_1, \ldots, \gamma_m\}, Func \cup \{c_i : \to \sigma(\tau_i) \mid i = 1, \ldots, n\}, Pred)$ be an extended polymorphic signature, where $\sigma \in Sub_{\Sigma}(X, V, \emptyset)$ is a typed substitution with $\sigma(\alpha_i) = \gamma_i$ $(i = 1, \ldots, m)$, $\sigma(\alpha) = \alpha$ for all other type variables α, and $\sigma(x_j : \tau_j) = c_j : \sigma(\tau_j)$ $(j = 1, \ldots, n)$. If $(\Sigma, C, V) \models L$, then $(\Sigma', C, V) \models L$ by Lemma 4.3. By Lemma 3.3, $(\Sigma', C, \emptyset) \models \sigma(L)$. By Lemma 4.2, $(\Sigma', C, \emptyset) \vdash \sigma(L) \leftarrow \emptyset$. Since the basic types γ_i and the constants $c_j : \sigma(\tau_j)$ do not appear in the clauses C, we can replace γ_i by α_i $(i = 1, \ldots, m)$ and $c_j : \sigma(\tau_j)$ by $x_j : \tau_j$ $(j = 1, \ldots, n)$ in the last deduction. Hence we obtain a deduction for $(\Sigma, C, V) \vdash L \leftarrow \emptyset$. □

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 [33] with SLD-refutation [2] 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 [23].

Example. Let a polymorphic signature contain the declarations $\mathtt{p}:\alpha \in Pred$, $\mathtt{q}:int \in Pred$ and $\mathtt{r}:\alpha \in Pred$ (α is a type variable). $\mathtt{X}, \mathtt{Y}, \mathtt{Z} \in Var$ are variable names and assume the following two clauses to be given:

$$\mathtt{p}(\mathtt{X}:int) \leftarrow \mathtt{q}(\mathtt{X}:int)$$
$$\mathtt{p}(\mathtt{Y}:\alpha) \leftarrow \mathtt{r}(\mathtt{Y}:\alpha)$$

The first clause is not allowed for proving the goal $\mathtt{p}(\mathtt{Z}:bool)$. We can use the second clause and have to prove in the next step the goal $\mathtt{r}(\mathtt{Z}:bool)$.

For proving the goal $\mathtt{p}(\mathtt{Z}:int)$ the first clause can be used. In this case we are left with the goal $\mathtt{q}(\mathtt{Z}:int)$ 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 3.1 *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 $\sigma \leqslant \sigma'$, 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 $\sigma \leqslant \sigma'$ 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 $\Sigma^u = (Term, Op)$, is defined as follows:

- $Term = \{term\}$.
- $\underbrace{h{:}term, \ldots, term}_{n} \to term \in Op$ for all $h \in H$ with arity n $(n \geqslant 0)$.
- $\underbrace{f{:}term, \ldots, term}_{n} \to term \in Op$ for all $f{:}\tau_1, \ldots, \tau_n \to \tau \in Func$ $(n \geqslant 0)$.
- $":"{:}term, term \to term \in Op$.

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 from *Func* and the symbol ":" with arity 2. It is easy to show that $Term_\Sigma(X, V') \subseteq T_{\Sigma^u}(X \cup V)$, where $V = uvar(V')$, 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 $\texttt{equal}(\texttt{1}{:}int, \texttt{true}{:}bool){:}bool$ is a Σ^u-term, but not a Σ-term if $\texttt{equal}{:}\alpha, \alpha \to bool \in Func$.

The notions of "substitution" and "unifier" for the algebra $T_{\Sigma^u}(X \cup V)$ are defined as usual (e.g., [24]) and we omit the details here. [33] 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 $\texttt{v}{:}list(int)$ is $\sigma(\alpha) = int$, $\sigma(\iota) = [\ ]$. 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 $\texttt{v}{:}list(int)$ in $Term_\Sigma(X, \{\texttt{v}{:}list(int)\})$. Generally, we can compute a most general unifier from a most general unifier in $T_{\Sigma^u}(X \cup V)$. In order to prove this proposition, we present the algorithm and the result of Robinson. The algorithm of Robinson uses *disagreement sets* to specify the differences of terms. For our purpose it is

important to inspect the differences in type expressions first. Therefore we define for $t_0, t_1 \in T_{\Sigma^u}(X \cup V)$ the *disagreement set* of t_0 and t_1, $ds(t_0, t_1)$, as follows:

- If $t_0 = t_1$ then $ds(t_0, t_1) := \emptyset$ else
- if $t_0 = t:\tau$ and $t_1 = t':\tau'$ then

$$ds(t_0, t_1) := \begin{cases} ds(t, t') & \text{if } \tau = \tau', \\ ds(\tau, \tau') & \text{otherwise,} \end{cases} \quad \text{else}$$

- if $t_0 \in X \cup V$ or $t_1 \in X \cup V$ then $ds(t_0, t_1) := \{t_0, t_1\}$ else
- if $t_0 = f(r_1, \ldots, r_m)$ and $t_1 = g(s_1, \ldots, s_n)$ $(m, n \geqslant 0)$ then
  if $f \neq g$ or $m \neq n$ then $ds(t_0, t_1) := \{t_0, t_1\}$ else
  if $r_i = s_i (i = 1, \ldots, j-1)$ and $r_j \neq s_j$ then $ds(t_0, t_1) := ds(r_j, s_j)$.

If σ is a substitution in $T_{\Sigma^u}(X \cup V)$ and the set $\{x \in X \cup V \mid \sigma(x) \neq x\}$ is finite, we denote σ by the set

$$\{x/\sigma(x) \mid x \in X \cup V \text{ and } \sigma(x) \neq x\}.$$

Then the following algorithm computes a most general unifier in $T_{\Sigma^u}(X \cup V)$.

Algorithm mgu

Input: $t_0, t_1 \in T_{\Sigma^u}(X \cup V)$.

Output: An mgu σ for t_0 and t_1 in $T_{\Sigma^u}(X \cup V)$ or *fail*, if t_0 and t_1 are not unifiable in $T_{\Sigma^u}(X \cup V)$.

(1) $k := 0$; $\sigma_0 := \{\}$,

(2) *If* $\sigma_k(t_0) = \sigma_k(t_1)$ *then* stop "σ_k is the mgu",

(3) *If* $\{x, t\} \subseteq ds(\sigma_k(t_0), \sigma_k(t_1))$ *and* $x \in X \cup V$ *and* x does not occur in t
then $\sigma_{k+1} := \{x/t\} \circ \sigma_k$; $k := k+1$; go to (2).
else stop "*fail*: t_0 and t_1 are not unifiable",

The following theorem is due to Robinson [33].

Theorem 5.1. *If* $t_0, t_1 \in T_{\Sigma^u}(x \cup V)$ *are unifiable in* $T_{\Sigma^u}(X \cup V)$, *then the algorithm "mgu" terminates and gives an mgu for* t_0 *and* t_1, *otherwise the algorithm "mgu" terminates and reports "fail*: t_0 *and* t_1 *are not unifiable".*

In the following we assume that a set $V \subseteq_U Var_{\Sigma,X}$ of typed variables with unique types is given and $V_0 := uvar(V)$.

Lemma 5.2. *Let* t_0 *and* t_1 *be unifiable* (Σ, X, V)*-terms and* σ *be a unifier for* t_0 *and* t_1. *Let* σ' *be a substitution in* $T_{\Sigma^u}(X \cup V_0)$ *with* $\sigma'(\alpha) := \sigma(\alpha)$ *for any* $\alpha \in X$ *and* $\sigma'(x) := t$ *if* $x:\tau \in V$ *and* $\sigma(x:\tau) = t:\sigma(\tau)$ *for any* $x \in V_0$. *Then* σ' *is a unifier for* t_0 *and* t_1 *in* $T_{\Sigma^u}(X \cup V_0)$.

Proof. It is straightforward to show (by induction on the size of terms) that $\sigma|_{T_H(X)} = \sigma'|_{T_H(X)}$ and $\sigma|_{Term_\Sigma(X,V)} = \sigma'|_{Term_\Sigma(X,V)}$. Therefore $\sigma'(t_0) = \sigma(t_0) = \sigma(t_1) = \sigma'(t_1)$. □

Hence each unifier corresponds to a unifier in $T_{\Sigma^u}(X \cup V_0)$. The converse is only true for most general unifiers in $T_{\Sigma^u}(X \cup V_0)$. The following lemma is due to [23].

Lemma 5.3. *Let t_0 and t_1 be two (Σ, X, V)-terms unifiable in $T_{\Sigma^u}(X \cup V_0)$ with σ a most general unifier in $T_{\Sigma^u}(X \cup V_0)$. Then there exists a typed substitution $\sigma' \in Sub_\Sigma(X, V, V')$ such that $\sigma'(\alpha) = \sigma(\alpha)$ for any $\alpha \in X$, $\sigma'(x:\tau) = \sigma(x):\sigma(\tau)$ for any $x:\tau \in V$ and $V' := \bigcup_{x:\tau \in V} var(\sigma(x:\tau))$. Moreover, σ' is a most general unifier for t_0 and t_1.*

Proof. At first we show $\sigma(x):\sigma(\tau) \in Term_\Sigma(X, V')$ for all $x:\tau \in V$. By Theorem 5.1, an mgu σ_k in $T_{\Sigma^u}(X \cup V_0)$ can be computed by the algorithm "mgu" presented above. We show by induction on the computation steps the following property of the computed substitutions σ_i in the algorithm "mgu": *Let* $W_i := \{x:\sigma_i(\tau) \mid x:\tau \in V\}$, $t \in Term_\Sigma(X, V)$. *Then* $\sigma_i(t) \in Term_\Sigma(X, W_i)$.

For $i = 0$ we have $W_0 = V$ and $\sigma_0(t) = t$. Let $i > 0$ and $\sigma_{i-1}(t) \in Term_\Sigma(X, W_{i-1})$ for all $t \in Term_\Sigma(X, V)$. By the algorithm "mgu", $\sigma_i = \{v/u\} \circ \sigma_{i-1}$ for a $v \in X \cup V_0$ and $u \in T_{\Sigma^u}(X \cup V_0)$.

(a) $v \in X$: Since $\sigma_{i-1}(t_0), \sigma_{i-1}(t_1) \in Term_\Sigma(X, W_{i-1})$, it must be $u \in T_H(X)$. It is straightforward to show that $\{v/u\}(t) \in Term_\Sigma(X, W_i)$ for all $t \in Term_\Sigma(X, W_{i-1})$.

(b) $v \in V_0$: Since $\sigma_{i-1}(t_0), \sigma_{i-1}(t_1) \in Term_\Sigma(X, W_{i-1})$, $v:\tau_v$ (for a $\tau_v \in T_H(X)$) must occur in $\sigma_{i-1}(t_0)$ or $\sigma_{i-1}(t_1)$ and therefore $u:\tau_v \in Term_\Sigma(X, W_{i-1})$ (otherwise v and u are not in the disagreement set). It is straightforward to show that $\{v/u\}(t) \in Term_\Sigma(X, W_i)$ for all $t \in Term_\Sigma(X, W_{i-1})$ since $W_i = W_{i-1}$ and $V \subseteq_U Var_{\Sigma,X}$ is a set of typed variables with *unique* types.

By induction hypothesis, it follows $\sigma_i(t) \in Term_\Sigma(X, W_i)$ for all $t \in Term_\Sigma(X, V)$.

Since t_0 and t_1 are unifiable in $T_{\Sigma^u}(X \cup V_0)$, the algorithm "mgu" stops with an mgu σ_k and $\sigma_k(t) \in Term_\Sigma(X, W_k)$ for all $t \in Term_\Sigma(X, V)$. If σ is another mgu in $T_{\Sigma^u}(X \cup V_0)$, then $\sigma(\tau)$ and $\sigma_k(\tau)$ are equivalent types for all $\tau \in T_H(X)$. Therefore $\sigma(t) \in Term_\Sigma(X, V')$ for all $t \in Term_\Sigma(X, V)$, in particular $\sigma(x):\sigma(\tau) = \sigma(x:\tau) \in Term_\Sigma(X, V')$ for all $x:\tau \in V$.

It can be shown in a similar way that $\sigma(\alpha) \in T_H(X)$ for all $\alpha \in X$ since $t_0, t_1 \in Term_\Sigma(X, V)$. By Lemma 3.1, there exists a typed substitution σ' with the conditions described in the lemma. It is straightforwad to show (by induction on the size of terms) that $\sigma|_{T_H(X)} = \sigma'|_{T_H(X)}$ and $\sigma|_{Term_\Sigma(X,V)} = \sigma'|_{Term_\Sigma(X,V)}$. Therefore $\sigma'(t_0) = \sigma(t_0) = \sigma(t_1) = \sigma'(t_1)$, i.e., σ' is a unifier for t_0 and t_1. By Lemma 5.2, σ' is a most general unifier. □

The requirement for a most general unifier in the last lemma is essential. If the unifier in $T_{\Sigma^u}(X \cup V_0)$ is not most general, then the proposition does not hold: If Σ is a polymorphic signature with basic types *bool* and *int*, and $V = (x:bool, y:bool\}$,

then the substitution $\sigma = \{x/1, y/1\}$ is a unifier for $x{:}bool$ and $y{:}bool$ in $T_{\Sigma^u}(X \cup V_0)$, but $\sigma(x{:}bool) = 1{:}bool \notin Term_\Sigma(X, V')$ is an ill-typed term.

The requirement for typed variables with unique types is also essential for the correspondence of most general unifiers in $T_{\Sigma^u}(X \cup V_0)$ to unifiers in $Term_\Sigma(X, V)$: Let $\mathtt{X}, \mathtt{Y} \in Var$ and Σ be a polymorphic signature with basic types *bool* and *int* and a function declaration

func f: *int*, *bool* → *bool*

Then the terms $\mathtt{f}(\mathtt{X}{:}int, \mathtt{X}{:}bool){:}bool$ and $\mathtt{f}(\mathtt{0}{:}int, \mathtt{Y}{:}bool){:}bool$ are unifiable in $T_{\Sigma^u}(\{\mathtt{X}, \mathtt{Y}\})$ and $\sigma = \{\mathtt{X/0},\ \mathtt{Y/0}\}$ is an mgu in $T_{\Sigma^u}(\{\mathtt{X}, \mathtt{Y}\})$. But $\sigma(\mathtt{Y}{:}bool) = \mathtt{0}{:}bool$ is an ill-typed term and therefore the theorem does not hold for this case. The following theorem shows that the polymorphic unification problem can be reduced to the unification problem in $T_{\Sigma^u}(X \cup V)$.

Theorem 5.4 (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 two (Σ, X, V)-terms are not unifiable, then they are not unifiable in $T_{\Sigma^u}(X \cup V_0)$ by Lemma 5.3. If two (Σ, X, V)-terms are unifiable, then (by Lemma 5.2) they are unifiable in $T_{\Sigma^u}(X \cup V_0)$. Theorem 5.1 yields a most general unifier in $T_{\Sigma^u}(X \cup V_0)$, and Lemma 5.3 converts the mgu in $T_{\Sigma^u}(X \cup V_0)$ into a most general unifier in $Term_\Sigma(X, V)$. □

The unification problem in the polymorphic case is solved by this theorem. There exist more efficient unification algorithms ([26, 3, 32]) that can also be used instead of the classical one presented above. We only require the following technical restriction that will be needed for later proofs:

If $V \subseteq Var_{\Sigma,X}$, t and t' are (Σ, X, V)-terms and σ is a most general unifier for t and t', then $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ and the following conditions hold:

(1) $uvar(t_i) \subseteq uvar(t) \cup uvar(t')$ and $tvar(t_i) \subseteq tvar(t) \cup tvar(t')$ for $i = 1, \ldots, n$.

(2) $x_i \notin var(t_j) \cup tvar(t_j)$ for all $i, j \in \{1, \ldots, n\}$, i.e., the most general unifier is an idempotent substitution.

(3) If $V \subseteq_U Var_{\Sigma,X}$, then $\bigcup_{x:\tau \in V} var(\sigma(x{:}\tau)) \subseteq_U Var_{\Sigma,X}$.

The classical unification algorithm meets these requirements.

6. Resolution

In this section we will show that the resolution principle in untyped Horn logic (see [24]) 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. We call a Σ-clause a *variant* of another Σ-clause if it is obtained by replacing type variables and typed variables by other type variables and typed variables, respectively, such that different variables are replaced by new different variables. Let (Σ, C) be a polymorphic logic program.

(a) Let $V, V' \subseteq_U Var_{\Sigma,X}$, $G \cup \{L\}$ be a (Σ, X, V)-goal and the (Σ, X, V)-clause $L' \leftarrow G'$ be a variant of a clause from C with $tvar(G \cup \{L\}) \cap tvar(L' \leftarrow G') = \emptyset$ and $var(G \cup \{L\}) \cap var(L' \leftarrow G') = \emptyset$. If there exists a most general unifier $\sigma \in Sub_\Sigma(X, V, V')$ for L and L', then $\sigma(G) \cup \sigma(G')$ is said to be *derived by resolution* from $G \cup \{L\}$ relative to σ and $L' \leftarrow G'$. Notation:

$$(\Sigma, C, V)\ G \cup \{L\} \vdash_R \sigma\ \sigma(G) \cup \sigma(G').$$

(b) Let $V \subseteq_U Var_{\Sigma,X}$ and G be a (Σ, X, V)-goal. A (Σ, C, V)-*resolution* of G is a sequence of the form

$$(\Sigma, C, V_0)\ G_0 \vdash_R \sigma_1\ G_1 \vdash_R \sigma_2\ G_2 \vdash_R \cdots \vdash_R \sigma_n\ G_n$$

where $G_0 = G$, $V_0 = V$ and $(\Sigma, C, V_i)\ G_i \vdash_R \sigma_{i+1}\ G_{i+1}$ with $\sigma_{i+1} \in Sub_\Sigma(X, V_i, V_{i+1})$ for $i = 0, 1, 2, \ldots, n-1$. The (Σ, C, V)-resolution is called *successful* if $G_n = \emptyset$. In this case n is called the length of the (Σ, C, V)-resolution, and $\sigma := \sigma_n \circ \cdots \circ \sigma_1$ is called a *computed answer*. Notation:

$$(\Sigma, C, V) \vdash_R \sigma\ G.$$

We remark that $V_i \subseteq_U Var_{\Sigma,X}$ for $i = 0, 1, 2, \ldots, n-1$ since $V \subseteq_U Var_{\Sigma,X}$. If $V \subseteq Var_{\Sigma,X}$ rather than $V \subseteq_U Var_{\Sigma,X}$, the unifier in a resolution step is not a most general one, the type variables and typed variables in a clause applied in a resolution step are not disjoint from those in the Σ-goal, or C is only a set of Σ-clauses rather than $var(c) \subseteq_U Var_{\Sigma,X}$ for all $c \in C$, then the resolution is called an *unrestricted* (Σ, C, V)-*resolution* and the symbol $\vdash_R$ is replaced by $\vdash_{UR}$.

The soundness of resolution can be shown by simulating a resolution sequence by a derivation in the polymorphic Horn clause calculus.

Theorem 6.1 (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 is a successful resolution $(\Sigma, C, V) \vdash_R \sigma\ G$ with computed answer $\sigma \in Sub_\Sigma(X, V, V')$, then $(\Sigma, C, V') \models \sigma(G)$.*

Proof. We assume that there is a successful resolution $(\Sigma, C, V) \vdash_R \sigma\ G$. Thus there is a (Σ, C, V)-resolution of the form

$$(\Sigma, C, V)\ G_0 \vdash_R \sigma_1\ G_1 \vdash_R \sigma_2\ G_2 \vdash_R \cdots \vdash_R \sigma_n\ \emptyset$$

with $G_0 = G$ and $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. We show the following proposition by induction on the resolution steps: *For* $i = 0, \ldots, n$: If $P \in G_{n-i}$, *then* $(\Sigma, C, V') \vdash \sigma_n \circ \cdots \circ \sigma_{n-i+1}(P)$.

This is true for $i=0$ since $G_n=\emptyset$. For the induction step we assume $(\Sigma, C, V')\vdash \sigma_n\circ\cdots\circ\sigma_{n-i+1}(P)$ for all $P\in G_{n-i}$. The $(n-i)$th resolution step has the form

$$(\Sigma, C, V_{n-i-1})\ G_{n-i-1}\underset{R}{\vdash}\sigma_{n-i}\ G_{n-i}$$

and there is a variant $L'\leftarrow G'$ of a clause from C and a Σ-atom $L\in G_{n-i-1}$ with

$$\sigma_{n-i}(L)=\sigma_{n-i}(L') \quad \text{and} \quad G_{n-i}=\sigma_{n-i}((G_{n-i-1}-\{L\})\cup G').$$

We have to show that $(\Sigma, C, V')\vdash\sigma_n\circ\cdots\circ\sigma_{n-i}(P)$ is true for all $P\in G_{n-i-1}$. As the (Σ, X, V_{n-i-1})-clause $L'\leftarrow G'$ is a variant of a clause from C, there exist $V''\subseteq Var_{\Sigma,X}$, $L''\leftarrow G''\in C$ and

$$\sigma''\in Sub_\Sigma(C, V'', V_{n-i-1}) \quad \text{with } \sigma''(L''\leftarrow G'')=L'\leftarrow G'.$$

Therefore,

$$(\Sigma, C, V'')\vdash L''\leftarrow G'' \quad \text{and} \quad (\Sigma, C, V_{n-i-1})\vdash L'\leftarrow G'$$

by the substitution rule. If we apply the substitution rule with typed substitution $\sigma_n\circ\cdots\circ\sigma_{n-i}$, we obtain

$$(\Sigma, C, V')\vdash\sigma_n\circ\cdots\circ\sigma_{n-i}(L'\leftarrow G').$$

By induction hypothesis, $(\Sigma, C, V')\vdash\sigma_n\circ\cdots\circ\sigma_{n-i+1}(P)$, for all $P\in G_{n-i}$. Since $\sigma_{n-i}(G')\subseteq G_{n-i}$, we get from multiple applications of the cut rule

$$(\Sigma, C, V')\vdash\sigma_n\circ\cdots\circ\sigma_{n-i}(L').$$

If $P\in G_{n-i-1}-\{L\}$, then $\sigma_{n-i}(P)\in G_{n-i}$, and, therefore,

$$(\Sigma, C, V')\vdash\sigma_n\circ\cdots\circ\sigma_{n-i}(P)$$

by induction hypothesis. This completes the induction step. We obtain the following proposition for $i=n$: *For all* $P\in G$, $(\Sigma, C, V')\vdash\sigma(P)$.

Let M be a model for (Σ, C) and v be a variable assignment for (X, V') in M (if there exists no such variable assignment, then $M, V'\models\sigma(G)$ is trivially true). By Theorem 4.1 (soundness of deduction), we obtain $M, v\models\sigma(P)$ for all $P\in G$. This implies $M, v\models\sigma(G)$. Therefore, $(\Sigma, C, V')\models\sigma(G)$. □

The completeness of resolution in untyped Horn logic can be proved by a fixpoint theorem using a transformation on Herbrand interpretations [37, 24]. In [16] this proof method is adapted to the polymorphic case. In this paper we will show the completeness of resolution for polymorphic logic programs by simulating each deduction in the polymorphic Horn clause calculus by resolution. Padawitz [30] has presented such a proof for many-sorted Horn clause logic with equality. However, he has required that all types are interpreted as nonempty sets, which simplifies the proof but is not reasonable in our context.

The simulation of deduction by resolution is more difficult than the simulation of resolution by deduction. A few technical lemmas will help to structure the completeness proof. The first lemma shows that the substitution rule is not necessary if $\hat{C}$ (the set of instantiated clauses) is used in a deduction.

Lemma 6.2. *Let C be a set of Σ-clauses, $V, V' \subseteq Var_{\Sigma,X}$ and $(\Sigma, C, V) \vdash L \leftarrow G$. For any typed substitution $\sigma \in Sub_\Sigma(X, V, V')$ there exists a deduction $(\Sigma, \hat{C}, V') \vdash \sigma(L \leftarrow G)$ where only axioms and cut rules are applied.*

Proof. We prove the lemma by induction on the number n of cut rule applications in a shortest deduction of $(\Sigma, C, V) \vdash L \leftarrow G$. The case $n = 0$ is trivial since $\sigma_0(L_0) \leftarrow \sigma_0(G_0) \in \hat{C}$ for all $L_0 \leftarrow G_0 \in C$ and all appropriate typed substitutions σ_0. Otherwise there is a last application of the cut rule in the deduction, say

$$(\Sigma, C, V_i) \vdash L_i \leftarrow G_i \cup \{L_j\} \quad \text{and} \quad (\Sigma, C, V_i) \vdash L_j \leftarrow G_j$$

occur in the deduction before the last application of the cut rule. Let $\sigma_1 \in Sub_\Sigma(X, V_i, V_i')$. We have to show that $(\Sigma, \hat{C}, V_i') \vdash \sigma_1(L_i) \leftarrow \sigma_1(G_i \cup G_j)$ can be deduced without an application of the substitution rule. The number of cut rule applications in shortest derivations of

$$(\Sigma, C, V_i) \vdash L_i \leftarrow G_i \cup \{L_j\} \quad \text{and} \quad (\Sigma, C, V_i) \vdash L_j \leftarrow G_j$$

is less than n. By induction hypothesis,

$$(\Sigma, \hat{C}, V_i') \vdash \sigma_1(L_i) \leftarrow \sigma_1(G_i \cup \{L_j\}) \quad \text{and} \quad (\Sigma, \hat{C}, V_i') \vdash \sigma_1(L_j) \leftarrow \sigma_1(G_j)$$

can be deduced without an application of the substitution rule. By an application of the cut rule, we obtain

$$(\Sigma, \hat{C}, V_i') \vdash \sigma_1(L_i) \leftarrow \sigma_1(G_i \cup G_j).$$

This proves the lemma. □

Lemma 6.3. *Let C be a set of Σ-clauses and $V \subseteq Var_{\Sigma,X}$. If $(\Sigma, C, V) \vdash L \leftarrow G$ where only axioms and cut rules are applied, then $(\Sigma, C', V) \vdash_{\overline{UR}} id_{X,V}\ L$ with $C' = C \cup \{P \leftarrow \mid P \in G\}$, and each substitution in the unrestricted resolution is equal to $id_{X,V}$.*

Proof. The lemma is proved by induction on the length of the deduction. Let $d_1, \ldots, d_n$ be a deduction for $(\Sigma, C, V) \vdash L \leftarrow G$ where only axioms and cut rules are applied. If $L \leftarrow G \in C$, then $(\Sigma, C', V)\ L \vdash_{\overline{UR}} id_{X,V}\ G$ is an unrestricted resolution step. If G consists of k Σ-atoms, then we achieve the empty goal with k further unrestricted resolution steps with substitutions $id_{X,V}$.

If $L \leftarrow G \notin C$, then the clause must be derived by an application of the cut rule, i.e., there are

$$d_i = (\Sigma, C, V) \vdash L \leftarrow G_0 \cup \{L_1\}, \qquad d_j = (\Sigma, C, V) \vdash L_1 \leftarrow G_1$$

with $G = G_0 \cup G_1$ and $i, j < n$. By induction hypothesis,

$$(\Sigma, C' \cup \{L_1 \leftarrow\}, V) \vdash_{\overline{\cup}R} id_{X,V}\ L \tag{1}$$

and

$$(\Sigma, C', V) \vdash_{\overline{\cup}R} id_{X,V}\ L_1 \tag{2}$$

since $G = G_0 \cup G_1$. If the clause $L_1 \leftarrow$ is used in resolution (1), then, by (2), it is possible to replace the resolution step by a sequence of resolution steps that derives L_1 to the empty goal using clauses from C'. Thus $(\Sigma, C', V) \vdash_{\overline{\cup}R} id_{X,V}\ L$ and each substitution in this unrestricted resolution is equal to $id_{X,V}$. □

Now we can prove the completeness of unrestricted resolution.

Theorem 6.4 (Completeness of unrestricted resolution for atoms). *Let C be a set of Σ-clauses, V, $V' \subseteq Var_{\Sigma,X}$ be finite and A be a (Σ, X, V)-atom. If $\sigma \in Sub_\Sigma(X, V, V')$ is a typed substitution with $(\Sigma, C, V') \models \sigma(A)$, then there exists a set $V_0 \subseteq Var_{\Sigma,X}$ and a typed substitution $\sigma_0 \in Sub_\Sigma(X, V_0, V')$ with $(\Sigma, C, V_0) \vdash_{\overline{\cup}R} \sigma_0\ A$ and $\sigma_0(A) = \sigma(A)$.*

Proof. W.l.o.g. we assume that σ affects only a finite number of type variables since V is finite, i.e., $tdom(\sigma)$ is finite. Let $(\Sigma, C, V') \models \sigma(A)$. Theorem 4.4 yields $(\Sigma, C, V') \vdash \sigma(A)$. By Lemma 6.2 and Lemma 6.3, there exists a successful unrestricted resolution of the form

$$(\Sigma, \hat{C}, V')\ \sigma(A) \vdash_{\overline{\cup}R} id_{X,V'}\ G_1 \vdash_{\overline{\cup}R} id_{X,V'} \cdots \vdash_{\overline{\cup}R} id_{X,V'}\ \emptyset.$$

In the first resolution step there exist $L_0 \leftarrow R_0 \in C$, $V_0' \subseteq Var_{\Sigma,X}$ and $\sigma_0 \in Sub_\Sigma(X, V_0', V')$ with $\sigma_0(L_0) = \sigma(A)$ and $\sigma_0(R_0) = G_1$.

W.l.o.g. we assume $tdom(\sigma) \cap tdom(\sigma_0) = \emptyset$ and $uvar(V) \cap uvar(V_0') = \emptyset$ (otherwise we choose an appropriate variant of $L_0 \leftarrow R_0$ and an appropriate typed substitution σ_0). We define $V_0 := V \cup var(L_0 \leftarrow R_0)$ and combine σ and σ_0 into a typed substitution $\sigma_1 \in Sub_\Sigma(X, V_0, V')$ with

$$\sigma_1(\alpha) = \begin{cases} \sigma(\alpha) & \text{if } \alpha \in tdom(\sigma), \\ \sigma_0(\alpha) & \text{otherwise,} \end{cases}$$

and

$$\sigma_1(x{:}\tau) = \begin{cases} \sigma(x{:}\tau) & \text{if } x{:}\tau \in V, \\ \sigma_0(x{:}\tau) & \text{if } x{:}\tau \in var(L_0 \leftarrow R_0). \end{cases}$$

Then $\sigma_1(A) = \sigma(A) = \sigma_0(L_0) = \sigma_1(L_0)$ and $\sigma_1(R_0) = \sigma_0(R_0) = G_1$. Therefore

$$(\Sigma, C, V_0)\ A \vdash_{\overline{\cup}R} \sigma_1\ G_1$$

is an unrestricted resolution step. If $G_1 = \emptyset$, then the proof is finished, otherwise there is a second resolution step

$$(\Sigma, \hat{C}, V')\ G_1 \vdash_{\overline{\cup}R} id_{X,V'}\ G_2.$$

Let $L_1' \leftarrow R_1' \in \hat{C}$ be the clause used in this resolution step, i.e., there exist $L_1 \leftarrow R_1 \in C$, $V_1' \subseteq Var_{\Sigma,X}$ and $\sigma_1' \in Sub_\Sigma(X, V_1', V')$ with $\sigma_1'(L_1 \leftarrow R_1) = L_1' \leftarrow R_1'$. Similarly to the

first resolution step, we combine σ_1' and $id_{X,V'}$ into a typed substitution $\sigma_2 \in Sub_\Sigma(X, V_1, V')$, where $V_1 := V' \cup var(L_1 \leftarrow R_1)$, such that

$$(\Sigma, C, V_1)\ G_1 \vdash_{\overline{U}R} \sigma_2\ G_2$$

is an unrestricted resolution step. Since $V' \subseteq V_1$, we can extend σ_1 to a typed substitution $\sigma_1 \in Sub_\Sigma(X, V_0, V_1)$. Hence we get the unrestricted resolution

$$(\Sigma, C, V_0)\ A \vdash_{\overline{U}R} \sigma_1\ G_1 \vdash_{\overline{U}R} \sigma_2\ G_2$$

with $\sigma_2(\sigma_1(A)) = \sigma_2(\sigma(A)) = \sigma(A)$ and $\sigma_2 \circ \sigma_1 \in Sub_\Sigma(X, V_0, V')$. If we apply the transformation of the second resolution step in the same way to the remaining resolution steps, we obtain an unrestricted resolution

$$(\Sigma, C, V_0)\ A \vdash_{\overline{U}R} \sigma_1\ G_1 \vdash_{\overline{U}R} \sigma_2 \cdots \vdash_{\overline{U}R} \sigma_n\ \emptyset$$

with $\sigma_n \circ \cdots \circ \sigma_1(A) = \sigma(A)$ and $\sigma_n \circ \cdots \circ \sigma_1 \in Sub_\Sigma(X, V_0, V')$. □

We need the next lemma to prove the completeness of unrestricted resolution for general goals.

Lemma 6.5. *Let C be a set of Σ-clauses, $V = \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\} \subseteq Var_{\Sigma,X}$ and G be a (Σ, X, V)-goal with $tvar(G) \subseteq tvar(V)$. Let p be a new symbol that does not occur in Σ, $\Sigma' := (H, Func, Pred \cup \{p{:}\tau_1, \ldots, \tau_n\})$, $L := p(x_1{:}\tau_1, \ldots, x_n{:}\tau_n)$ and $C' := C \cup \{L \leftarrow G\}$. Then*

$$(\Sigma, C, V') \models \sigma(G) \Rightarrow (\Sigma', C', V') \models \sigma(L)$$

for all $\sigma \in Sub_\Sigma(X, V, V')$.

Proof. Let $(\Sigma, C, V') \models \sigma(G)$ and M' be a model for (Σ', C'). Then M' is also a model for (Σ', C) and $M', V \models L \leftarrow G$. By Lemma 3.3, $M', V' \models \sigma(L) \leftarrow \sigma(G)$. Suppose v is a variable assignment for (X, V') in M'. M' is also a model for (Σ, C) if we omit the interpretation of the predicate symbol p in M'. Therefore $M', v \models \sigma(G)$. $M', v \models \sigma(L) \leftarrow \sigma(G)$ implies $M', v \models \sigma(L)$. Hence we obtain $M', V' \models \sigma(L)$. □

Theorem 6.6 (Completeness of unrestricted resolution). *Let C be a set of Σ-clauses, $V \subseteq Var_{\Sigma,X}$ 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 $V_0 \subseteq Var_{\Sigma,X}$ and a typed substitution $\sigma_0 \in Sub_\Sigma(X, V_0, V')$ with $(\Sigma, C, V_0) \vdash_{\overline{U}R} \sigma_0\ G$ and $\sigma_0(G) = \sigma(G)$.*

Proof. Let $tvar(G) \subseteq tvar(V)$, otherwise add new variables with types from $tvar(G) - tvar(V)$ to V and extend σ to these variables so that this condition holds and σ does not alter the new variables. Then we define p, L, Σ' and C' as in the last lemma. $(\Sigma, C, V') \models \sigma(G)$ implies $(\Sigma', C', V') \models \sigma(L)$. By Theorem 6.4, there exist $V_0 \subseteq Var_{\Sigma,X}$ and a typed substitution $\sigma_0 \in Sub_\Sigma(X, V_0, V')$ with

$(\Sigma', C', V_0) \vdash_{\overline{U}R} \sigma_0\ L$ and $\sigma_0(L) = \sigma(L)$. Since the only clause for the elimination of an atom with predicate symbol p is $L \leftarrow G$, there is a resolution

$$(\Sigma, C', V_0)\ L \vdash_{\overline{U}R} \sigma_1\ \sigma_1(G) \vdash_{\overline{U}R} \sigma_2\ G_2 \cdots \vdash_{\overline{U}R} \sigma_n\ \emptyset$$

with $\sigma_0 = \sigma_n \circ \cdots \circ \sigma_1$. We can combine the typed substitution σ_1 with the typed substitution σ_2 in the second resolution step and obtain an unrestricted (Σ, C, V_0)-resolution for G with the same computed answer. □

To prove completeness of resolution with most general unifiers we need the following lemma.

Lemma 6.7 (mgu-Lemma). *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 an unrestricted resolution*

$$(\Sigma, C, V)\ G \vdash_{\overline{U}R} \sigma_1\ G_1 \vdash_{\overline{U}R} \sigma_2\ G_2 \vdash_{\overline{U}R} \cdots \vdash_{\overline{U}R} \sigma_n\ \emptyset$$

for G, then there exists an unrestricted resolution

$$(\Sigma, C, V)\ G \vdash_{\overline{U}R} \sigma_1'\ G' \vdash_{\overline{U}R} \sigma_2'\ G_2' \vdash_{\overline{U}R} \cdots \vdash_{\overline{U}R} \sigma_n'\ \emptyset$$

where each σ_i' is a most general unifier and $\sigma_n' \circ \cdots \circ \sigma_1' \in Sub_\Sigma(X, V, V')$. Furthermore, there exists a typed substitution $\phi \in Sub_\Sigma(X, V', V'')$ with $\phi \circ \sigma_n' \circ \cdots \circ \sigma_1' = \sigma_n \circ \cdots \circ \sigma_1$.

Proof. By induction on the length n of the resolution: If $n = 1$, then (Σ, C, V) $G \vdash_{\overline{U}R} \sigma_1\ \emptyset$. Hence there exists a variant $L \leftarrow \emptyset$ of a clause from C with $\sigma_1(G) = \sigma_1(L)$. By Unification Theorem 5.4, there exists a most general unifier $\sigma_1' \in Sub_\Sigma(X, V, V')$ for G and L and therefore there is a typed substitution $\phi \in Sub_\Sigma(X, V', V'')$ with $\phi \circ \sigma_1' = \sigma_1$. Thus (Σ, C, V) $G \vdash_{\overline{U}R} \sigma_1'\ \emptyset$ is a resolution for G.

If $n > 1$, then there is a resolution

$$(\Sigma, C, V)\ G \vdash_{\overline{U}R} \sigma_1\ G_1 \vdash_{\overline{U}R} \sigma_2\ G_2 \vdash_{\overline{U}R} \cdots \vdash_{\overline{U}R} \sigma_n\ \emptyset.$$

Hence there exists a variant $L' \leftarrow G'$ of a clause from C with $\sigma_1(L') = \sigma_1(L)$ where $G = G_0 \cup \{L\}$. By Unification Theorem 5.4, there exists a most general unifier $\sigma_1' \in Sub_\Sigma(X, V, V')$ for L' and L and therefore there is a typed substitution $\phi \in Sub_\Sigma(X, V', V'')$ with $\phi \circ \sigma_1' = \sigma_1$. If $G_1' := \sigma_1'(G_0 \cup G')$, then

$$(\Sigma, C, V)\ G \vdash_{\overline{U}R} \sigma_1'\ G_1' \vdash_{\overline{U}R} \sigma_2 \circ \phi\ G_2 \vdash_{\overline{U}R} \cdots \vdash_{\overline{U}R} \sigma_m\ \emptyset$$

is an unrestricted resolution for G (w.l.o.g. we assume that ϕ does not alter any type variables or typed variables from the clause used in the second resolution step) and

$$(\Sigma, C, V')\ G_1' \vdash_{\overline{U}R} \sigma_2 \circ \phi\ G_2 \vdash_{\overline{U}R} \cdots \vdash_{\overline{U}R} \sigma_n\ \emptyset$$

is an unrestricted resolution for G_1' of length $n-1$. Since $V' \subseteq_U Var_{\Sigma,X}$ and by induction hypothesis, there exists an unrestricted resolution

$$(\Sigma, C, V')\ G_1' \vdash_{\overline{U}R} \sigma_2'\ G_2' \vdash_{\overline{U}R} \cdots \vdash_{\overline{U}R} \sigma_n'\ \emptyset$$

where each σ'_i is a most general unifier, $\sigma'_n \circ \cdots \circ \sigma'_2 \in Sub_\Sigma(X, V', V_1)$, and there exists a typed substitution $\rho \in Sub_\Sigma(X, V_1, V_2)$ with $\rho \circ \sigma'_n \circ \cdots \circ \sigma'_2 = \sigma_n \circ \cdots \circ \sigma_2 \circ \phi$. Hence we obtain an unrestricted resolution

$$(\Sigma, C, V)\ G \vdash_{UR} \sigma'_1\ G'_1 \vdash_{UR} \sigma'_2\ G'_2 \vdash_{UR} \cdots \vdash_{UR} \sigma'_n\ \emptyset$$

where each σ'_i is a most general unifier, $\sigma'_n \circ \cdots \circ \sigma'_1 \in Sub_\Sigma(X, V, V_1)$ and $\rho \in Sub_\Sigma(X, V_1, V_2)$ is a typed substitution with $\rho \circ \sigma'_n \circ \cdots \circ \sigma'_1 = \sigma_n \circ \cdots \circ \sigma_2 \circ \phi \circ \sigma'_1 = \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1$. □

The completeness of resolution follows from completeness of unrestricted resolution and mgu-Lemma 6.7:

Theorem 6.8 (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 $(\Sigma, C, V_0) \vdash_R \sigma_0\ G$, and there is a typed substitution $\phi \in Sub_\Sigma(X, V_1, V')$ with $\phi(\sigma_0(G)) = \sigma(G)$.*

Proof. By completeness Theorem 6.6, there exist $V_2 \subseteq Var_{\Sigma,X}$ and an unrestricted resolution of the form

$$(\Sigma, C, V_2)\ G \vdash_{UR} \sigma_1\ G_1 \vdash_{UR} \sigma_2\ G_2 \vdash_{UR} \cdots \vdash_{UR} \sigma_n\ \emptyset \tag{1}$$

with $\sigma_n \circ \cdots \circ \sigma_1 \in Sub_\Sigma(X, V_2, V')$ and $\sigma_n \circ \cdots \circ \sigma_1(G) = \sigma(G)$. W.l.o.g. we assume that V_2 is finite. $V_2 \subseteq_U Var_{\Sigma,X}$ is not true in general. Hence we construct a set $V_0 \subseteq_U Var_{\Sigma,X}$ corresponding to V_2: If $V_2 = var(G) \cup \{x_1 : \tau_1, \ldots, x_m : \tau_m\}$, then we define $V_0 := var(G) \cup \{y_1 : \tau_1, \ldots, y_m : \tau_m\}$ where y_i are pairwise distinct new variable names from *Var*. Let $\rho \in Sub_\Sigma(X, V_0, V_2)$ with $\rho(\alpha) = \alpha$ for all $\alpha \in X$, $\rho(x:\tau) = x:\tau$ for all $x:\tau \in var(G)$ and $\rho(y_i : \tau_i) = x_i : \tau_i$ for $i = 1, \ldots, m$. $V_0 \subseteq_U Var_{\Sigma,X}$ and ρ is invertible so that $\rho \circ \rho^{-1} = id_{X,V_2}$. We show that

$$(\Sigma, C, V_0)\ G \vdash_{UR} \sigma_1 \circ \rho\ G_1 \vdash_{UR} \sigma_2\ G_2 \cdots \vdash_{UR} \sigma_n\ \emptyset \tag{2}$$

is an unrestricted resolution for G: Let $L' \leftarrow G'$ be the clause used in the first resolution step in (1). Therefore $G = G_0 \cup \{L_0\}$ with $\sigma_1(L_0) = \sigma_1(L')$ and $G_1 = \sigma_1(G_0 \cup G')$. $\rho^{-1}(L' \leftarrow G')$ is also a variant of a clause from C. $\rho(G) = G$ since $\rho(x:\tau) = x:\tau$ for all $x:\tau \in var(G)$. Thus $\sigma_1(\rho(L_0)) = \sigma_1(L_0) = \sigma_1(L') = \sigma_1(\rho(\rho^{-1}(L')))$ and $\sigma_1(\rho(G_0 \cup \rho^{-1}(G'))) = \sigma_1(G_0 \cup G') = G_1$. Therefore (2) is indeed an unrestricted resolution for G.

We assume for the resolution (2) that $tvar(G_i) \cap tvar(c_i) = \emptyset$ and $var(G_i) \cap var(c_i) = \emptyset$ where c_i is the clause used in the ith resolution step. If this is not the case then we choose an appropriate variant of c_i and extend V_0 and the preceding substitutions as in the proof of Theorem 6.4. By the mgu-Lemma 6.7, we obtain from (2) a resolution

$$(\Sigma, C, V_0)\ G \vdash_R \sigma'_1\ G_1 \vdash_R \sigma'_2\ G_2 \cdots \vdash_R \sigma'_n\ \emptyset$$

and a typed substitution $\phi \in Sub_\Sigma(X, V_1, V')$ (where $\sigma_0 := \sigma'_n \circ \cdots \circ \sigma'_1 \in Sub_\Sigma(X, V_0, V_1)$) with $\phi \circ \sigma_0 = \sigma_n \circ \cdots \circ \sigma_1 \circ \rho$. Hence $\phi(\sigma_0(G)) = \sigma_n \circ \cdots \circ \sigma_1 \circ \rho(G) = \sigma_n \circ \cdots \circ \sigma_1(G) = \sigma(G)$. □

Soundness Theorem 6.1 and completeness Theorem 6.8 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 (cf. Section 9).

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 [28] 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. We present some optimizations for the resolution of polymorphic programs.

7.1. Type preserving functions

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 •: α, $list(\alpha) \to list(\alpha)$

are type preserving functions, whereas

func `equal`: α, $\alpha \to bool$

is not a type preserving function. We will see that in the case of type preserving functions type annotations in 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

$$\Phi(\texttt{equal}(1{:}int, 2{:}int){:}bool) = \texttt{equal}(1{:}int, 2{:}int){:}bool.$$

Formally, Φ can be defined as a mapping $\Phi: Term_\Sigma(X, V) \to T_{\Sigma^u}(X \cup V_0)$, where $V \subseteq_U Var_{\Sigma,X}$ and $V_0 = uvar(V)$.

- $\Phi(x:\tau) := x:\tau$ for all $x:\tau \in V$,
- $\Phi(c:\tau) := c:\tau$ for all constants $c:\tau$,
- $\Phi(f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau) := f(t_1', \ldots, t_n'):\tau$ where $\Phi(t_i:\tau_i) = t_i':\tau_i$ $(i = 1, \ldots, n)$ for all composite terms with a type preserving function f,
- $\Phi(g(t_1:\tau_1, \ldots, t_n:\tau_n):\tau) := g(\Phi(t_1:\tau_1), \ldots, \Phi(t_n:\tau_n)):\tau$ for all composite terms where g is not type preserving

The next proposition states an important property of Φ.

Proposition 7.1. *The mapping Φ is injective.*

Proof. Φ deletes only type annotations in arguments of type preserving functions. Such type annotations can be computed from the type declaration of the function and the actual result type in a unique way (here it is essential that we have no overloading!). Hence the proposition can be shown by a simple induction on the size of (Σ, X, V)-terms. □

Lemma 7.2. *Let $V \subseteq_U Var_{\Sigma,X}$ and $V_0 := uvar(V)$. If $t_0, t_1 \in Term_\Sigma(X, V)$ are unifiable, then $\Phi(t_0)$ and $\Phi(t_1)$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$.*

Proof. Let $\sigma \in Sub_\Sigma(X, V, V')$ be a unifier for t_0 and t_1. Let σ' be a substitution in $T_{\Sigma^u}(X \cup V_0)$ with $\sigma'(\alpha) := \sigma(\alpha)$ for all $\alpha \in X$ and $\sigma'(x) := t$ for all $x:\tau \in V$ with $\Phi(\sigma(x:\tau)) = t:\sigma(\tau)$. It is straightforward to show (by induction on the size of terms) that $\sigma|_{T_H(X)} = \sigma'|_{T_H(X)}$ and $\sigma'(\Phi(t)) = \Phi(\sigma(t))$ for all $t \in Term_\Sigma(X, V)$. Therefore, $\sigma'(\Phi(t_0)) = \Phi(\sigma(t_0)) = \Phi(\sigma(t_1)) = \sigma'(\Phi(t_1))$. □

Lemma 7.3. *Let $V \subseteq_U Var_{\Sigma,X}$, $V_0 := uvar(V)$ and $t_0, t_1 \in Term_\Sigma(X, V)$. If $\Phi(t_0)$ and $\Phi(t_1)$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$, then t_0 and t_1 are unifiable.*

Proof. Let $\Phi(t_0)$ and $\Phi(t_1)$ be unifiable in $T_{\Sigma^u}(X \cup V_0)$. By Theorem 5.1, a most general unifier in $T_{\Sigma^u}(X \cup V_0)$ can be computed by the algorithm "mgu". We show by induction on the computation steps the following property of the computed substitutions σ_i in the algorithm "mgu": *Let* $W_i := \{x:\sigma_i(\tau) \mid x:\tau \in V\}$, $t \in Term_\Sigma(X, V)$. *Then* $\sigma_i(\Phi(t)) \in \Phi(Term_\Sigma(X, W_i))$.

For $i = 0$, we have $W_0 = V$ and $\sigma_0(\Phi(t)) = \Phi(t)$. Let $i > 0$ and $\sigma_{i-1}(\Phi(t)) \in \Phi(Term_\Sigma(X, W_{i-1}))$ for all $t \in Term_\Sigma(X, V)$. By the algorithm "mgu", $\sigma_i = \{v/u\} \circ \sigma_{i-1}$ for a variable $v \in V_0 \cup X$ and $u \in T_{\Sigma^u}(X \cup V_0)$.

(a) $v \in X$: Since $\sigma_{i-1}(\Phi(t_0)), \sigma_{i-1}(\Phi(t_1)) \in \Phi(Term_\Sigma(X, W_{i-1}))$, it must be $u \in T_H(X)$ since in $\Phi(Term_\Sigma(X, W_{i-1}))$ type expressions occur always as the second argument of a ":"-term. It is straightforward to show that $\{v/u\}(\Phi(t)) \in \Phi(Term_\Sigma(X, W_i))$ for all $t \in Term_\Sigma(X, W_{i-1})$.

(b) $v \in V_0$: Since $\sigma_{i-1}(\Phi(t_0)), \sigma_{i-1}(\Phi(t_1)) \in \Phi(Term_\Sigma(X, W_{i-1}))$, $v:\tau_v$ (for a $\tau_v \in T_H(X)$) and $u:\tau_v \in \Phi(Term_\Sigma(X, W_{i-1}))$ or the subterms $f(\ldots, v, \ldots):\tau$ and $f(\ldots, u, \ldots):\tau$ must occur at the same position in $\sigma_{i-1}(\Phi(t_0))$ and $\sigma_{i-1}(\Phi(t_1))$. In the latter case f is a type preserving function with $f:\tau_f \in Func$ and there exists $\sigma, \sigma' \in TS(H, X)$ with $\sigma(\tau_f) = \tau_1, \ldots, \tau_n \to \tau$ and $\sigma'(\tau_f) = \tau'_1, \ldots, \tau'_n \to \tau$, whereas $f(\ldots, v:\tau_j, \ldots)$ and $f(\ldots, u':\tau'_j, \ldots)$ are the corresponding subterms in $\sigma_{i-1}(t_0)$ and $\sigma_{i-1}(t_1)$. Let $\tau_f = \rho_1, \ldots, \rho_n \to \rho$ and $X' = tvar(\rho)$. Since $\sigma(\rho) = \tau = \sigma'(\rho)$, it follows $\sigma|_{T_H(X')} = \sigma'|_{T_H(X')}$ and $\sigma(\rho_j) = \sigma'(\rho_j)$ (since $tvar(\rho_j) \subseteq tvar(\rho)$). Thus $\tau_j = \tau'_j$ and $f(\ldots, v:\tau_j, \ldots)$ and $f(\ldots, u':\tau_j, \ldots)$ are the corresponding subterms in $\sigma_{i-1}(t_0)$ and $\sigma_{i-1}(t_1)$ with $\Phi(u':\tau_j) = u:\tau_j$. Therefore $v:\tau_j \in W_{i-1}$ and $u:\tau_j \in \Phi(Term_\Sigma(X, W_{i-1}))$. Now it is straightforward to show that $\{v/u\}(\Phi(t)) \in \Phi(Term_\Sigma(X, W_i))$ for all $t \in Term_\Sigma(X, W_{i-1})$ since $W_i = W_{i-1}$.

By induction hypothesis and (a) and (b), it follows $\sigma_i(\Phi(t)) \in \Phi(Term_\Sigma(X, W_i))$ for all $t \in Term_\Sigma(X, V)$.

Since $\Phi(t_0)$ and $\Phi(t_1)$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$, the algorithm "mgu" stops with an mgu σ_k and $\sigma_k(\Phi(t)) \in \Phi(Term_\Sigma(X, W_k))$ for all $t \in Term_\Sigma(X, V)$. By Proposition 7.1, for each $t \in Term_\Sigma(X, V)$ there exists a unique term $t' \in Term_\Sigma(X, W_k)$ with $\Phi(t') = \sigma_k(\Phi(t))$. Hence we can define a typed substitution $\sigma \in Sub_\Sigma(X, V, V')$ with $\sigma(\alpha) = \sigma_k(\alpha)$ for all $\alpha \in X$ and $\sigma(x:\tau) = t'$ for all $x:\tau \in V$, whereas $t' \in Term_\Sigma(X, W_k)$ with $\Phi(t') = \sigma_k(\Phi(x:\tau))$. It is easy to show (by induction on the size of terms) that $\Phi(\sigma(t)) = \sigma_k(\Phi(t))$ for all $t \in Term_\Sigma(X, V)$. Therefore $\Phi(\sigma(t_0)) = \sigma_k(\Phi(t_0)) = \sigma_k(\Phi(t_1)) = \Phi(\sigma(t_1))$. Proposition 7.1 yields $\sigma(t_0) = \sigma(t_1)$, i.e., σ is a unifier for t_0 and t_1. □

Theorem 7.4 (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 $T_{\Sigma^u}(X \cup V_0)$.*

Proof. If t_0 and t_1 are not unifiable, then $\Phi(t_0)$ and $\Phi(t_1)$ are not unifiable in $T_{\Sigma^u}(X \cup V_0)$ by Lemma 7.3. If t_0 and t_1 are unifiable, then $\Phi(t_0)$ and $\Phi(t_1)$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$ by Lemma 7.2. By Theorem 5.1, a most general unifier for $\Phi(t_0)$ and $\Phi(t_1)$ in $T_{\Sigma^u}(X \cup V_0)$ can be computed, which can be transformed into a unifier for t_0 and t_1 (see proof of Lemma 7.3). By the proof of Lemma 7.2, this corresponds to a most general unifier for t_0 and t_1. □

The optimized unification can be extended to 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. For the case of monomorphic signatures we obtain the following result.

Corollary 7.5. *If the signature is monomorphic, i.e., all function and predicate declarations do not contain any type variables, then type annotations are unnecessary for the unification of atoms.*

This corollary shows that in many-sorted Horn clause programs the resolution procedure has the same efficiency as in untyped programs since types are not needed at run time.

7.2. Type-generally defined predicates

There is another possibility for optimization if a predicate is defined with *most general types*, i.e., in each clause for the predicate the head has a most general type and the predicates in the body are also defined with most general types. In the following we develop the necessary definitions and results to prove this idea.

We assume that $V \subseteq_U Var_{\Sigma,X}$ is a set of typed variables with *unique* types. A (Σ, X, V)-term $t:\tau$ is called *type general* if for any (Σ, X, V)-term $t':\tau'$ with $\phi(\tau) = \tau'$ and $\phi \in TS(H, X)$ which is unifiable with $t:\tau$ and which has no type variables in common with $t:\tau$, there exists a typed substitution $\sigma \in Sub_\Sigma(X, V, V')$ with $\sigma(t:\tau) = \sigma(t':\tau')$ and $\sigma(\alpha) = \alpha$ for all $\alpha \in tvar(t':\tau')$. The property *type general* can be simply extended to atoms, if we treat predicates as Boolean functions.

For instance, if there is a declaration $g:\alpha, \beta \to bool$, then $g(X:\alpha, Y:\beta):bool$ is a type-general term, but neither $g(X:\alpha, I:int):bool$ nor $g(X:\alpha, Z:\alpha):bool$ is a type-general term. Note that variables and constants are always type general. For type-general terms we do not require the result type to be most general as otherwise type-general terms may not have type-general subterms. But this is important for further results.

We will show that in case of type-general terms type annotations may be omitted in the unification. First we have to prove some properties of type-general terms.

Lemma 7.6. *Let $t = f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau$ and $t' = f(t'_1:\tau'_1, \ldots, t'_n:\tau'_n):\tau'$ be (Σ, X, V)-terms with $\phi(\tau) = \tau'$ for a type substitution $\phi \in TS(H, X)$. If t is type general, then there exists a type substitution $\sigma \in TS(H, X)$ with $\sigma(\tau_1, \ldots, \tau_n \to \tau) = \tau'_1, \ldots, \tau'_n \to \tau'$.*

Proof. Let $f:\tau_f \in Func$ with $\tau_f = \cdots \to \tau_0$. There exists $\phi' \in TS(H, X)$ with $\phi'(\tau_f) = \tau_1, \ldots, \tau_n \to \tau$. Let $\phi'' \in TS(H, X)$ with $\phi''(\alpha) = \phi'(\alpha)$ for all $\alpha \in tvar(\tau_0)$ and $\phi''(\alpha) = \alpha$ for all other $\alpha \in X$. Let $r = f(x_1:\rho_1, \ldots, x_n:\rho_n):\rho$, where $\phi''(\tau_f)$ and $\rho_1, \ldots, \rho_n \to \rho$ are equivalent and $x_i \in Var$ with $x_i \neq x_j$ for $i \neq j$. We assume that type variables and typed variables in r and t are disjoint (otherwise rename them). By construction of r, there exist $\sigma, \sigma' \in TS(H, X)$ with

$$\sigma(\rho_1, \ldots, \rho_n \to \rho) = \tau_1, \ldots, \tau_n \to \tau \quad \text{and} \quad \sigma'(\rho_1, \ldots, \rho_n \to \rho) = \tau'_1, \ldots, \tau'_n \to \tau'.$$

Let $V_0 := var(r) \cup var(t)$. We define $\theta \in Sub_\Sigma(X, V_0, V_1)$ by

$$\theta(\alpha) = \begin{cases} \sigma(\alpha) & \text{if } \alpha \in tvar(\rho_1, \ldots, \rho_n \to \rho), \\ \alpha & \text{otherwise,} \end{cases}$$

and

$$\theta(x:\tau) = \begin{cases} t_i:\tau_i & \text{if } x:\tau = x_i:\rho_i, \\ x:\tau & \text{otherwise.} \end{cases}$$

Such a typed substitution exists since $\theta(\rho_i) = \sigma(\rho_i) = \tau_i$.

Now we have $\theta(r) = f(\theta(x_1:\rho_1), \ldots, \theta(x_n:\rho_n)):\theta(\rho) = t = \theta(t)$, i.e., r and t are unifiable. By definition of "type general", there exists a type substitution $\theta' \in TS(H, X)$ with $\rho_1, \ldots, \rho_n \to \rho = \theta'(\rho_1, \ldots, \rho_n \to \rho) = \theta'(\tau_1, \ldots, \tau_n \to \tau)$. Therefore $\sigma'(\theta'(\tau_1, \ldots, \tau_n \to \tau)) = \tau_1', \ldots, \tau_n' \to \tau'$. □

Lemma 7.7. *Let $t = f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau$ be a type-general (Σ, X, V)-term. Then $t_i:\tau_i$ is a type-general (Σ, X, V)-term and $(tvar(\tau) \cap tvar(t_i:\tau_i)) - tvar(\tau_i) = \emptyset$ for $i = 1, \ldots, n$. Furthermore, $(tvar(t_i:\tau_i) - tvar(\tau_i)) \cap (tvar(t_j:\tau_j) - tvar(\tau_j)) = \emptyset$ for $i \neq j$.*

Proof. We prove the case $i = 1$. First we show that $t_1:\tau_1$ is type general. Let $r_1:\rho_1$ be a (Σ, X, V)-term with $\phi(\tau_1) = \rho_1$ for a $\phi \in TS(H, X)$, $tvar(t) \cap tvar(r_1:\rho_1) = \emptyset$ and $t_1:\tau_1$ is unifiable with $r_1:\rho_1$. We assume that $\phi(\alpha) = \alpha$ for all $\alpha \in X - tvar(\tau_1)$. Let $x_2, \ldots, x_n$ be pairwise distinct variable names not occurring in V and $V_0 := V \cup \{x_2:\phi(\tau_2), \ldots, x_n:\phi(\tau_n)\}$. Then $r := f(r_1:\rho_1, x_2:\phi(\tau_2), \ldots, x_n:\phi(\tau_n)):\phi(\tau)$ is a (Σ, X, V_0)-term unifiable with t. Since t is type general, there exists a unifier $\sigma \in Sub_\Sigma(X, V_0, V_0')$ for t and r with $\sigma(\alpha) = \alpha$ for all $\alpha \in tvar(r)$. Thus $\sigma(t_1:\tau_1) = \sigma(r_1:\rho_1)$ and $\sigma(\alpha) = \alpha$ for all $\alpha \in tvar(r_1:\rho_1)$. Hence $t_1:\tau_1$ is type general.

Assumption: There exists $\alpha \in (tvar(\tau) \cap tvar(t_1:\tau_1)) - tvar(\tau_1)$. Then α occurs in the subterm $t_1:\tau_1$ but not in τ_1. Therefore all occurrences of α in $t_1:\tau_1$ can be replaced by a new type variable β and the resulting term $t_1':\tau_1$ remains also well-typed and has the same result type. Clearly, the term $t' = f(t_1':\tau_1, \ldots, t_n:\tau_n):\tau$ is unifiable with t (for convenience we do not rename the type variables in t' which formally must be done). But each unifier for t and t' must identify the type variable β in $t_1':\tau_1$ with the type variable α in τ because these are identical in t. Hence t is not type general in contrast to our assumption.

The last proposition in the lemma can be proved in the same way. □

For a precise definition of "omitting all type annotations in a term" we define a mapping Ψ that deletes all type annotations in a term. Formally, Ψ can be defined as a mapping $\Psi: Term_\Sigma(X, V) \to T_{\Sigma^u}(X \cup V_0)$, where $V_0 = uvar(V)$.

- $\Psi(x:\tau) := x$ for all $x:\tau \in V$,
- $\Psi(c:\tau) := c$ for all constants $c:\tau$,
- $\Psi(f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau) := f(\Psi(t_1:\tau_1), \ldots, \Psi(t_n:\tau_n))$ for all composite terms $f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau \in Term_\Sigma(X, V)$.

The definitions of Ψ can be simply extended to Σ-atoms.

Theorem 7.8 (Unification with type-general terms). *Let $t:\tau$, $t':\tau'$ be (Σ, X, V)-terms with $tvar(t:\tau) \cap tvar(t':\tau') = \emptyset$, $var(t:\tau) \cap var(t':\tau') = \emptyset$ and $t:\tau$ be type general with $\phi(\tau) = \tau'$ for a type substitution $\phi \in TS(H, X)$. $t:\tau$ and $t':\tau'$ are unifiable iff $\Psi(t:\tau)$ and $\Psi(t':\tau')$ are unifiable in $T_{\Sigma^u}(X \cup uvar(V))$.*

Proof. Let $t:\tau$ and $t':\tau'$ be unifiable, $\sigma \in Sub_\Sigma(X, V, V')$ be a unifier for t_0 and t_1 and $V_0 := uvar(V)$. Let σ' be a substitution in $T_{\Sigma^u}(X \cup V_0)$ with $\sigma'(\alpha) = \sigma(\alpha)$ for all $\alpha \in X$ and $\sigma'(x) = \Psi(\sigma(x:\tau))$ for all $x:\tau \in V$. It is straightforward to show (by induction on the size of terms) that $\sigma|_{T_H(X)} = \sigma'|_{T_H(X)}$ and $\sigma'(\Psi(t)) = \Psi(\sigma(t))$ for all $t \in Term_\Sigma(X, V)$. Therefore $\sigma'(\Psi(t:\tau)) = \Psi(\sigma(t:\tau)) = \Psi(\sigma(t':\tau')) = \sigma'(\Psi(t':\tau'))$.

Conversely, let $\Psi(t:\tau)$ and $\Psi(t':\tau')$ be unifiable in $T_{\Sigma^u}(X \cup V_0)$. We assume $\phi(\alpha) = \alpha$ for all $\alpha \in X - tvar(\tau)$ and prove the proposition by induction on the size of the term $t:\tau$.

$t:\tau \in V$. Let $\sigma \in Sub_\Sigma(X, V, V')$ with $\sigma|_{T_H(X)} = \phi$, $\sigma(x:\rho) = x:\phi(\rho)$ for all $x:\rho \in V - \{t:\tau\}$ and $\sigma(t:\tau) = \sigma(t':\tau')$. Then σ is a unifier for $t:\tau$ and $t':\tau'$.

$t:\tau = c:\tau$ with $c:\to\tau_c \in Func$. Since $\phi(\tau) = \tau'$ and c and t' are unifiable in $T_{\Sigma^u}(X \cup V_0)$, $t:\tau$ and $t':\tau'$ are unifiable.

$t:\tau = f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau$ $(n > 0)$. The case $t':\tau' \in V$ is the same as $t:\tau \in V$. Therefore we assume $t':\tau' = f(t'_1:\tau'_1, \ldots, t'_n:\tau'_n):\tau'$. By Lemma 7.7, each $t_i:\tau_i$ is type general. By Lemma 7.6, there exists $\phi' \in TS(H, X)$ with $\phi'(\tau_1, \ldots, \tau_n \to \tau) = \tau'_1, \ldots, \tau'_n \to \tau'$. $\Psi(t_i:\tau_i)$ and $\Psi(t'_i:\tau'_i)$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$ since $\Psi(t:\tau)$ and $\Psi(t':\tau')$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$. Thus we can apply the induction hypothesis and infer that $t_i:\tau_i$ and $t'_i:\tau'_i$ are unifiable, i.e., there exist unifiers σ_i for $t_i:\tau_i$ and $t'_i:\tau'_i$ with $\sigma_i(\alpha) = \alpha$ for all $\alpha \in tvar(t'_i:\tau'_i)$ (by definition of "type general"). Let $\phi_i := \sigma_i|_{T_H(tvar(t_i:\tau_i))}$ be type substitutions $(i = 1, \ldots, n)$. Then $\phi_i(\tau_i) = \sigma_i(\tau_i) = \tau'_i = \phi'(\tau_i)$. This implies $\phi_i(\alpha) = \phi'(\alpha)$ for all $\alpha \in tvar(\tau_i)$ $(i = 1), \ldots, n)$. By Lemma 7.7, the type substitutions ϕ_i can be combined into a type substitution θ with

$$\theta(\alpha) = \begin{cases} \phi_i(\alpha) & \text{for all } \alpha \in tvar(t_i:\tau_i), \\ \phi'(\alpha) & \text{for all } \alpha \in tvar(\tau), \\ \alpha & \text{for all other } \alpha \in X. \end{cases}$$

We extend θ to a typed substitution by $\theta(x:\tau) := x:\theta(\tau)$ for all $x:\tau \in V$. Then in $\theta(t:\tau)$ and $\theta(t':\tau') = t':\tau'$ subterms at same positions have identical types: If we start the algorithm "mgu" with $\theta(t:\tau)$ and $\theta(t':\tau')$ at no time there are type expressions in the disagreement set. The unification does not depend on the types, and therefore $\theta(t:\tau)$ and $\theta(t':\tau')$ are unifiable since $\Psi(t:\tau)$ and $\Psi(t':\tau')$ are unifiable in $T_{\Sigma^u}(X \cup V_0)$. □

The condition $\phi(\tau) = \tau'$ in the last theorem is necessary, otherwise the theorem does not hold. For example, if there is a declaration

func id: $\alpha \to \alpha$

then $id(I:int):int$ is type general but not unifiable with the term $id(B:bool):bool$. $\Psi(id(I:int):int) = id(I)$ and $\Psi(id(B:bool):bool) = id(B)$ are unifiable in $T_{\Sigma^u}(X \cup \{I, B\})$.

If we want to omit type annotations, it is not sufficient that the clause heads are type-general atoms. It is necessary that all predicates in the clause bodies are

type-generally defined. Therefore we define: Let (Σ, C) be a polymorphic logic program and $p:\tau_p \in Pred$.

(a) The predicate p is type-generally defined in (Σ, C) relative to a set of predicates P if for every clause

$$p(t_1:\tau_1, \ldots, t_n:\tau_n) \leftarrow L_1, \ldots, L_k \in C$$

the following conditions hold:

(1) $p(t_1:\tau_1, \ldots, t_n:\tau_n)$ is type general.

(2) If $L_i = q_i(\cdots)$ and $q_i \notin P \cup \{p\}$, then q_i is type-generally defined in (Σ, C) relative to $P \cup \{p\}$ (for $i = 1, \ldots, k$).

(b) The predicate p is *type-generally defined in* (Σ, C) if p is type-generally defined in (Σ, C) relative to $\emptyset$.

The next lemma shows that type variables are never instantiated in a resolution step if the predicates are type-generally defined. Type variables can only be renamed depending on the computation method for the most general unifier.

Lemma 7.9. *Let (Σ, C) be a polymorphic logic program and G be a (Σ, X, V)-goal. If (Σ, C, V) $G \vdash_R \sigma$ G' is a resolution step and all predicates occurring in G are type-generally defined in (Σ, C), then all predicates occurring in G' are type-generally defined in (Σ, C) and $\sigma|_{T_H(tvar(G))}$ is bijective, i.e., the types τ and $\sigma(\tau)$ are equivalent for each type τ in G.*

Proof. Let (Σ, C, V) $G \vdash_R \sigma$ G' be a resolution step. Then $G = G_0 \cup \{L_0\}$, the (Σ, X, V)-clause $L_1 \leftarrow G_1$ is a variant of a clause from C, $tvar(G) \cap tvar(L_1 \leftarrow G_1) = \emptyset$, $var(G) \cap var(L_1 \leftarrow G_1) = \emptyset$, $\sigma \in Sub_\Sigma(X, V, V_1)$ is an mgu for L_0 and L_1, and $G' = \sigma(G_0 \cup G_1)$. Hence $L_0 = p(\cdots)$ and p is type-generally defined in (Σ, C). By definition of "type-generally defined", all predicates occurring in G_1 are type-generally defined in (Σ, C). Thus all predicates occurring in G' are type-generally defined in (Σ, C) and L_1 is type general. By definition of "type general", there exists a typed substitution $\sigma' \in Sub_\Sigma(X, V, V')$ with $\sigma'(L_0) = \sigma'(L_1)$ and $\sigma'(\alpha) = \alpha$ for all $\alpha \in tvar(L_0)$. Hence σ' is a unifier for L_0 and L_1 and there exists $\phi \in Sub_\Sigma(X, V_1, V')$ with $\phi \circ \sigma = \sigma'$ since σ is an mgu for L_0 and L_1. For all $\alpha \in tvar(L_0)$ we have $\alpha = \sigma'(\alpha) = \phi(\sigma(\alpha))$, which implies $\sigma(\alpha) \in X$. Hence $\sigma(\alpha) \in X$ for all $\alpha \in tvar(G)$ because σ is an mgu for L_0 and L_1 and $tvar(G) \cap tvar(L_1) = \emptyset$.

We have to show that σ is injective on $tvar(G)$. σ is injective on $tvar(L_0)$ because $\alpha_1, \alpha_2 \in tvar(L_0)$ with $\sigma(\alpha_1) = \sigma(\alpha_2)$ implies $\alpha_1 = \phi(\sigma(\alpha_1)) = \phi(\sigma(\alpha_2)) = \alpha_2$. Since σ is an mgu for L_0 and L_1, we may assume

$$\sigma(\alpha) = \alpha \quad \forall \alpha \notin tvar(L_0) \cup tvar(L_1) \tag{1}$$

and

$$tvar(\sigma(\alpha)) \in tvar(L_0) \cup tvar(L_1) \quad \forall \alpha \in tvar(L_0) \cup tvar(L_1). \tag{2}$$

Suppose there are $\alpha_1 \in tvar(L_0)$ and $\alpha_2 \in tvar(G_0) - tvar(L_0)$ with $\sigma(\alpha_1) = \sigma(\alpha_2)$. $tvar(G_0) \cap tvar(L_1) = \emptyset$ and (1) implies $\sigma(\alpha_2) = \alpha_2$, i.e., $\sigma(\alpha_1) = \alpha_2$. (2) implies $\alpha_2 = \sigma(\alpha_1) \in tvar(L_0) \cup tvar(L_1)$. This contradicts our assumption.

Hence σ is injective on $tvar(G)$, i.e., $\sigma|_{T_H(tvar(G))}$ is bijective. □

The next corollary extends the result of the last lemma to resolution derivations.

Corollary 7.10. *Let (Σ, C) be a polymorphic logic program, G be a (Σ, X, V)-goal and all predicates occurring in G be type-generally defined in (Σ, C). If*

$$(\Sigma, C, V)\ G \underset{R}{\vdash} \sigma_1\ G_1 \underset{R}{\vdash} \cdots \underset{R}{\vdash} \sigma_n\ G_n$$

is a sequence of resolution steps, then $\sigma_n \circ \cdots \circ \sigma_1|_{T_H(tvar(G))}$ is bijective.

Proof. Let $G_0 = G$ and $X_i := tvar(G_i) \cup \cdots \cup tvar(G_0)$. If c_i is the clause used in the ith resolution step, we can assume $tvar(c_i) \cap X_{i-1} = \emptyset$. Since each σ_i is an mgu, we may assume that $\sigma_i(\alpha) = \alpha$ for all $\alpha \in X_{i-1} - tvar(G_{i-1})$. By induction on the resolution steps and Lemma 7.9, it follows that $\sigma_i \circ \cdots \circ \sigma_1|_{T_H(X_{i-1})}$ is bijective. □

Theorem 7.11 (Optimized unification for type-generally defined predicates). *Let (Σ, C) be a polymorphic logic program. If G is a Σ-goal where all predicates in G are type-generally defined in (Σ, C), then type annotations are unnecessary during a resolution for G.*

Proof. Only type-general atoms are unified during a resolution for G. By Theorem 7.8, type annotations have no influence on success or failure of the unification. By Corollary 7.10, the types of G are not modified during a resolution for G. □

Next we want to develop an algorithm for deciding the property *type general*. For this purpose we need an alternative characterization of type-general terms. We call a (Σ, X, V)-term $f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau$ *directly type general* if the following conditions hold:

(1) $(tvar(\tau) \cap tvar(t_i:\tau_i)) - tvar(\tau_i) = \emptyset$ for $i = 1, \ldots, n$.

(2) $(tvar(t_i:\tau_i) - tvar(\tau_i)) \cap (tvar(t_j:\tau_j) - tvar(\tau_j)) = \emptyset$ for $i, j = 1, \ldots, n$ with $i \neq j$.

(3) $f:\rho_1, \ldots, \rho_n \to \rho \in Func$, $\sigma \in TS(H, X)$ with $\sigma(\rho) = \tau$ and $\sigma(\alpha) = \alpha$ for all $\alpha \in X - tvar(\rho)$ implies that $\sigma(\rho_1, \ldots, \rho_n \to \rho)$ and $\tau_1, \ldots, \tau_n \to \tau$ are equivalent.

The next lemma shows the relation between the properties *directly type general* and *type general*. The notions of "occurrences" and "subterms" are standard (see for example [17]) and we omit the definitions here.

Lemma 7.12. *Let $t:\tau$ be a (Σ, X, V)-term. If each subterm of $t:\tau$ is a variable, a constant, or a directly type-general term, then $t:\tau$ is type general.*

Proof. First we show by induction on the size of $t:\tau$ that for each $t':\tau' \in Term_\Sigma(X, V)$ with $\sigma_0(\tau) = \tau'$ (for a type substitution $\sigma_0 \in TS(H, X)$) that is unifiable with $t:\tau$ and that has no type variables in common with $t:\tau$, there exists a type substitution $\sigma \in TS(H, X)$ that affects only type variables from $tvar(t:\tau)$ so that subterms at

identical occurrences in $\sigma(t:\tau)$ and $t':\tau'$ have identical types (in this proof we extend each type substitution σ to a typed substitution by $\sigma(x:\tau) := x:\sigma(\tau)$).

If $t:\tau$ is a variable or a constant, then we define $\sigma(\alpha) = \sigma_0(\alpha)$ if $\alpha \in tvar(\tau)$ and $\sigma(\alpha) = \alpha$ otherwise.

Induction step. $t:\tau = f(t_1:\tau_1, \ldots, t_n:\tau_n):\tau$. If $t':\tau'$ is a variable, then we define $\sigma(\alpha) = \sigma_0(\alpha)$ for $\alpha \in tvar(\tau)$ and $\sigma(\alpha) = \alpha$ for all other $\alpha \in X$.

Otherwise, $t':\tau' = f(t_1':\tau_1', \ldots, t_n':\tau_n'):\tau'$. Let $f:\rho_1, \ldots, \rho_n \to \rho \in Func$ and $\sigma_1, \sigma_2 \in TS(H, X)$ with $\sigma_1(\rho_1, \ldots, \rho_n \to \rho) = \tau_1, \ldots, \tau_n \to \tau$ and $\sigma_2(\rho_1, \ldots, \rho_n \to \rho) = \tau_1', \ldots, \tau_n' \to \tau'$. $\sigma_2(\rho) = \tau' = \sigma_0(\tau) = \sigma_0(\sigma_1(\rho))$ and therefore $\sigma_2(\alpha) = \sigma_0(\sigma_1(\alpha))$ for all $\alpha \in tvar(\rho)$. Since $t:\tau$ is directly type general (condition (3)), there exists $\sigma_3 \in TS(H, X)$ with $\sigma_3(\tau_1, \ldots, \tau_n \to \tau) = \tau_1', \ldots, \tau_n' \to \tau'$. We assume that σ_3 alters only type variables from $tvar(\tau_1, \ldots, \tau_n \to \tau)$. By condition (1), each composite subterm of $\sigma_3(t_i:\tau_i)$ is directly type general. Thus we can apply the induction hypothesis and we get (for $i = 1, \ldots, n$) type substitutions $\phi_i \in TS(H, X)$ that alters only type variables from $tvar(t_i:\tau_i)$ so that subterms at identical occurrences in $\phi_i(t_i:\tau_i)$ and $t_i':\tau_i'$ have identical types and $\phi_i(\alpha) = \sigma_3(\alpha)$ for all $\alpha \in tvar(\tau_i)$. By conditions (2) and (1), we can combine the type substitutions $\phi_1, \ldots, \phi_n$ and σ_3 into one type substitution σ with the desired properties.

If $t':\tau'$ is a (Σ, X, V)-term with $\sigma_0(\tau) = \tau'$ that is unifiable with $t:\tau$ and that has no type variables in common with $t:\tau$, there exists a type substitution $\sigma \in TS(H, X)$ which alters only type variables from $tvar(t:\tau)$ so that subterms at identical occurrences in $\sigma(t:\tau)$ and $t':\tau'$ have identical types. If we compute a unifier σ' for $\sigma(t:\tau)$ and $t':\tau'$ with algorithm "mgu", we get $\sigma'(\alpha) = \alpha$ for all $\alpha \in X$. Therefore $\sigma' \circ \sigma$ is a unifier for $t:\tau$ and $t':\tau'$ with $\sigma' \circ \sigma(\alpha) = \alpha$ for all $\alpha \in tvar(t':\tau')$, i.e., $t:\tau$ is a type-general term. □

The next lemma is the justification for the following algorithm *type_general.*

Lemma 7.13 (Type general). *A (Σ, X, V)-term is type general iff each subterm is a variable, a constant, or a directly type-general term.*

Proof. "⇒": By Lemma 7.6 and Lemma 7.7.
"⇐": By Lemma 7.12. □

Now we are able to present the algorithm *type_general.* 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 substitution, if t is type general, 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* $f: \rho_1, \ldots, \rho_n \to \rho_0 \in Func$ *and* $\sigma = mgu(\rho_0, \rho') \neq fail$ *then* $\rho_1', \ldots, \rho_n' \to \rho_0' := skolemize(\sigma(\rho_1, \ldots, \rho_n \to \rho_0))$.

If $mgu(\rho_0', \tau) = \sigma_0 \neq fail$ *and* $type\_general(\sigma_0(t_1:\tau_1), \rho_1') = \sigma_1 \neq fail$ *and* $\cdots$ $type\_general(\sigma_{n-1}(\ldots(\sigma_0(t_n:\tau_n))\ldots), \rho_n') = \sigma_n \neq fail$

then stop with $\sigma_n \circ \cdots \circ \sigma_1 \circ \sigma_0$ *else* stop with *fail*.

(5) stop with *fail*.

7.3. Comparison to the type system of Mycroft-O'Keefe

The next proposition shows that the polymorphic logic programs of [28] can be executed without dynamic type checking since their result holds only if each function is type preserving [29].

Proposition 7.14 (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* $\tau_1, \ldots, \tau_n$ *are equivalent, then* L *is type general.*

Proof. Let $L' = p(r_1:\rho_1, \ldots, r_n:\rho_n)$ be a (Σ, X, V)-atom unifiable with L and $tvar(L) \cap tvar(L') = \emptyset$. Since τ_p and $\tau_1, \ldots, \tau_n$ are equivalent, there exists a type substitution $\phi \in TS(H, X)$ with $\phi(\tau_1, \ldots, \tau_n) = \tau_p$. There exists another type substitution $\phi' \in TS(H, X)$ with $\phi'(\tau_p) = \rho_1, \ldots, \rho_n$. Therefore $(\phi' \circ \phi)(\tau_1, \ldots, \tau_n) = \rho_1, \ldots, \rho_n$. We assume without loss of generality that $\phi' \circ \phi$ alters only type variables of $\tau_1, \ldots, \tau_n$. Then $\phi' \circ \phi$ is an mgu for $\rho_1, \ldots, \rho_n$ and $\tau_1, \ldots, \tau_n$. Let $\sigma \in Sub_\Sigma(X, V, V')$ with $\sigma|_{T_H(X)} = \phi' \circ \phi$ and $\sigma(x:\tau) = x:\sigma(\tau)$ for all $x:\tau \in V$. Then $\sigma(t_i:\tau_i) = t_i':\rho_i$, where t_i and t_i' differ only in their types. By Theorem 7.4, types are unnecessary for the unification of $t_i':\rho_i$ and $r_i:\rho_i$. Since the two terms have the same type, the computation of a most general unifier with the algorithm "mgu" has no influence on the type variables in ρ_i. Hence there exists a unifier σ' for L and L' with $\sigma'(\alpha) = \alpha$ for all $\alpha \in tvar(L')$, i.e., L is type general. □

By this proposition, all predicates in a polymorphic logic program with the restrictions of [28] are type-generally defined, i.e., type annotations are unnecessary during the resolution of a Σ-goal by Theorem 7.11. 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 [28].

Moreover, in our type system it is allowed to define clauses for special cases in contrast to Mycroft-O'Keefe's type system. Such clauses can be used to reduce the search space in the resolution process. Therefore resolution with types may be more efficient than in the untyped case. This is demonstrated by the following example:

type $\tau_1/0$, $\tau_2/0$

func $f_1, \ldots, f_m : \to \tau_1$ **func** $g_1, \ldots, g_n : \to \tau_2$

pred $\mathtt{t}:\alpha$ **pred** $\mathtt{t1}:\tau_1$ **pred** $\mathtt{t2}:\tau_2$ **pred** $=:\alpha, \alpha$

clauses:

$\mathtt{t}(\mathtt{X}:\tau_1) \leftarrow \mathtt{t1}(\mathtt{X}:\tau_1)$ $\mathtt{t}(\mathtt{Y}:\tau_2) \leftarrow \mathtt{t2}(\mathtt{Y}:\tau_2)$

$\mathtt{t1}(f_1:\tau_1) \leftarrow$ $\ldots \mathtt{t1}(f_m:\tau_1) \leftarrow$

$\mathtt{t2}(g_1:\tau_2) \leftarrow$ $\ldots \mathtt{t2}(g_n:\tau_2) \leftarrow$

$\mathtt{X}:\alpha = \mathtt{X}:\alpha \leftarrow$

We want to prove the goal

$$\mathtt{t}(\mathtt{Z}:\tau_2), \quad \mathtt{Z}:\tau_2 = g_i:\tau_2.$$

If we omit all type annotations and use the Prolog backtracking strategy, then the goal is proved in $m+i+2$ resolution steps. If the types are not omitted, i.e., the unification considers the types of terms, then the goal is proved in $i+1$ steps since the first clause of the predicate `t` cannot be applied.

Therefore, type information may be useful to reduce the search space in the resolution process. This is also true for order-sorted logic programs. E.g., Schmidt-Schauss [36] and Huber and Varsek [19] have shown examples in order-sorted logic where typed unification leads to more efficient proofs than in untyped logic.

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 [28] 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 [12]. Miller and Nadathur [27] 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 [18]. 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 [38] 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 [28]). 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 *pred*2 of arity 2 that denotes the type of binary predicate expressions. 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 [38] for more discussion). If the underlying system implements indexing on the first arguments of predicates (as done in most compilers for Prolog, cf. [39, 15]), 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 [38].

The compilation of higher-order functions into first-order logic was also proposed by Bosco and Giovannetti [4], 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 [28] 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 [8] 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 [8]. 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
func [ ]: $\rightarrow list(\alpha)$
func •: α, $list(\alpha) \rightarrow list(\alpha)$
pred append: $list(\alpha)$, $list(\alpha)$, $list(\alpha)$

clauses:

```
append([1,2], [3,4], [1,2,3,4])←
append([ ], L, L)←
append([E|R], L, [E|RL])←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" [20], 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 [28]. On the other hand, type information is useful to reduce the search space in the resolution process. Thus there are examples where the unification with types leads to a more efficient resolution than in the untyped case. 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 [7]. 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.

Acknowledgment

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

References

[1] H. Aït-Kaci and R. Nasr, LOGIN: A logic programming language with built-in inheritance, *J. Logic Programming* **3** (1986) 185–215.

[2] K.R. Apt and M.H. Van Emden, Contributions to the theory of logic programming, *J. ACM* **29** (3) (1982) 841–862.

[3] M. Bidoit and J. Corbin, A rehabilitation of Robinson's unification algorithm, in: *Proc. IFIP '83* (1983) 909–914.

[4] P.G. Bosco and E. Giovannetti, IDEAL: An ideal deductive applicative language, in: *Proc. IEEE Internat. Symp. on Logic Programming*, Salt Lake City (1986) 89–94.

[5] A. Church, A formulation of the simple theory of types, *J. Symbolic Logic* **5** (1940) 56–68.

[6] W.F. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer, Berlin, third rev. and ext. ed., 1987).

[7] R. Dietrich and F. Hagl, A polymorphic type system with subtypes for Prolog, in: *Proc. ESOP 88*, Nancy, Lecture Notes in Computer Science, Vol. 300 (Springer, Berlin, 1988) 79–93.

[8] L. Damas and R. Milner, Principal type-schemes for functional programs, in: *Proc. 9th Ann. Symp. on Principles of Programming Languages* (1982) 207–212.

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

[10] J.A. Goguen and J. Meseguer, Completeness of many-sorted equational logic, Report No. CSLI-84-15, Stanford University, 1984.

[11] J.A. Goguen and J. Meseguer, Eqlog: Equality, types, and generic modules for logic programming, in: D. De Groot and G. Lindstrom, eds, *Logic Programming, Functions, Relations, and Equations*, (Prentice-Hall, Englewood Cliffs, 1986) 295–363.

[12] W. Goldfarb, The Undecidability of the second-order unification problem, *Theoret. Comput. Sci.* **13** (1981) 225–230.

[13] 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, ed., *Current Trends in Programming Methodologyy*, Vol. 4 (Prentice-Hall, Englewood Cliffs, NJ, 1978) 80–149.

[14] W. Hankley, Feature analysis of Turbo Prolog, *SIGPLAN Notices* **22** (3) (1987) 111–118.

[15] M. Hanus, Formal specification of a Prolog compiler, in: *Proc. Workshop on Programming Language Implementation and Logic Programming*, Orléans, Lecture Notes in Computer Science, Vol. 348 (Springer, Berlin, 1988) 273–282.

[16] M. Hanus, Horn clause programs with polymorphic types, technical report 248, FB Informatik, Univ. Dortmund, 1988.

[17] G. Huet and D.C. Oppen, Equations and rewrite rules: a survey, in: R.V. Book, ed. *Formal Language Theory: Perspectives and Open Problems* (Academic Press, New York, 1980).
[18] G.P. Huet, A unification algorithm for typed λ-calculus, *Theoret. Comput. Sci.* **1** (1975) 27–57.
[19] M. Huber and I. Varsek, Extended Prolog with order-sorted resolution, in: *Proc. 4th IEEE Internat. Symp. on Logic Programming*, San Francisco (1987) 34–43.
[20] J. Jaffar and J.-L. Lassez, Constraint logic programming, in: *Proc. 14th ACM Symp. on Principles of Programming Languages*, Munich (1987) 111–119.
[21] T. Kanamori and K. Horiuchi, Type inference in Prolog and its application, in: *Proc. 9th IJCAI* (1985) 704–707.
[22] F. Kluźniak, Type synthesis for ground Prolog, in: *Proc. 4th Internat. Conf. on Logic Programming*, Melbourne (1987) 788–816.
[23] 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.
[24] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, second extended ed., 1987).
[25] P. Mishra, Towards a theory of types in Prolog, in: *Proc. IEEE Internat. Symp. on Logic Programming*, Atlantic City (1984) 289–298.
[26] A. Martelli and U. Montanari, An efficient unification algorithm, *ACM Trans. Programming Languages Systems* **4** (2) (1982) 258–282.
[27] D.A. Miller and G. Nadathur, Higher-order logic programming, in: *Proc. 3rd Internat. Conf. on Logic Programming*, London, Lecture Notes in Computer Science, Vol. 225 (Springer, Berlin, 1986) 448–462.
[28] A. Mycroft and R.A. O'Keefe, A polymorphic type system for Prolog. *Artificial Intelligence* **23** (1984) 295–307.
[29] A. Mycroft, private communication, 1987.
[30] P. Padawitz, *Computing in Horn Clause Theories* EATCS Monographs on Theoretical Computer Science, Vol. 16 (Springer, Berlin, 1988).
[31] A. Poigné, On specifications, theories and models with higher types, *Inform. and Control* **68** (1-3) (1986).
[32] M.S. Paterson and M.N. Wegman, Linear Unification, *J. Comput. System Sci.* **17** (1978) 348–375.
[33] J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM* **12** (1) (1965) 23–41.
[34] G. Smolka, Order-sorted Horn logic: semantics and deduction, SEKI Report SR-86-17, FB Informatik, Univ. Kaiserslautern, 1986.
[35] G. Smolka, TEL (Version 0.9) report and user manual, SEKI Report SR-87-11, FB Informatik, Univ. Kaiserslautern, 1988.
[36] M. Schmidt-Schauss, A many sorted calculus with polymorphic functions based on resolution and paramodulation, in: *Proc. 9th IJCAI* (1985).
[37] M.H. Van Emden and J.A. Kowalski, The semantics of predicate logic as a programming language, *J. ACM* **23** (4) (1976) 733–742.
[38] D.H.D. Warren, Higher-order extensions to Prolog: are they needed? in: *Machine Intelligence* **10** (1982) 441–454.
[39] D.H.D. Warren, An abstract Prolog instruction set, Technical Note 309, SRI International, Stanford, 1983.
[40] E. Yardeni and E. Shapiro, A type system for logic programs, Technical Report CS87-05, The Weizmann Institute of Science, 1987.
[41] J. Zobel, Derivation of polymorphic types for Prolog programs, in: *Proc. 4th Internat. Conf. on Logic Programming*, Melbourne (1987) 817–838.