In *Proc. Int. Conference on Algebraic and Logic Programming 1990*,
pp. 70–85, Springer LNCS 463

Logic Programs with Equational Type Specifications
(Extended Abstract)

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

This paper proposes a framework for logic programming with different type systems. In this framework a typed logic program consists of a type specification and a Horn clause program which is well-typed with respect to the type specification. The type specification defines all types which can be used in the logic program. Relations between types are expressed by equations on the level of types. This permits the specification of many-sorted, order-sorted, polymorphic and polymorphically order-sorted type systems.

We present the declarative semantics of our framework and a resolution procedure for typed logic programs. Resolution requires a unification procedure for typed terms which is based on a unification procedure for the type theory. An interesting application is a type system that combines parametric polymorphism with order-sorted typing and permits higher-order logic programming. Moreover, our framework sheds some new light on the rôle of types in logic programming.

1 Introduction

During recent years, various attempts have been made to integrate types into logic programming languages. The *inference-based approaches* ([Mis84] [XW88] among others) try to deduce the type information from an untyped logic program. But it has been argued that untyped logic programs do not contain the type information expected by the programmer [Nai87]. Therefore we are interested in *declaration-based approaches* where type declarations are added to the programs and a type checker verifies the consistence of the program w.r.t. the type declarations. A simple example for this approach is the type system of Turbo-Prolog which is comparable to many-sorted Horn logic [Pad88] and has no influence on the operational behaviour of the programs. A more flexible type system motivated from ML was proposed by Mycroft and O'Keefe [MO84]. It offers parametric polymorphism, and an extension of this type system [Han89a] allows the application of higher-order programming techniques. In general it is necessary to consider the types at run time, but it has been shown that for Prolog-like applications of higher-order programming all type information can be omitted at run time [Han89c]. This type system can also be applied to a language that combines functional and logic programming [Han90]. Another direction for typing logic programs are order-sorted type systems where different types may be related by an inclusion relation [SNGM87]. In order-sorted logic programming types are present at run time, but the type information can be used to avoid unnecessary computations and reduce the search space [HV87]. Smolka [Smo89] has proposed the combination of parametric polymorphism and order-sorted typing for a logic programming language. These different proposals raise the question: What is the influence of different type systems on the operational semantics of logic programs? We want to give an answer to this question.

For this purpose we propose a framework for logic programming with different type systems. We adopt some of the ideas of Poigné [Poi86] who has proposed a two-level approach for algebraic specifications with higher types. In his approach each level consists of an equational specification where the first-level describes a type structure and the second level is an equational specification with sort expressions from the first level. While he has used the approach for the specification of the typed λ-calculus, we will use a similar approach for a new framework for typed logic programming. In our two-level approach the first level consists of a specification of a type structure. It contains all types which will be used inside the logic program and some relations between types specified by equations. Hence the first level is an equational specification [EM85]. The second level is based on the specified type structure and consists of a specification of the types of all variables, constants, functions and predicates occurring in the logic program and a set of Horn clauses which must be well-typed with respect to the type specification. The operational semantics, which is resolution with a unification procedure on well-typed terms, ensures that type errors do not occur while executing well-typed programs. We give some examples to show the basic ideas.

Example 1.1 *Parametric polymorphism* is used for defining universal data structures which can be applied to different concrete types. A classical example are polymorphic lists which can be applied to integers giving lists of integers, to Booleans giving lists of Booleans, etc. The following signature specifies a type structure for a program which uses the basic types of integers and Booleans and the polymorphic types of lists and pairs of elements:

`TYPEOPS`	*int*:		$\rightarrow$	*type*
	bool:		$\rightarrow$	*type*
	list:	*type*	$\rightarrow$	*type*
	pair:	*type*, *type*	$\rightarrow$	*type*

This type structure has only a single sort *type*. Hence all types can be used as arguments for the polymorphic type constructors *list* and *pair*. The set of all types specified by this signature is the set of all well-formed terms which may contain some *type variables*. For instance, types w.r.t. the above specification are

$$int \qquad bool \qquad list(int) \qquad list(\alpha) \qquad pair(bool, \beta) \qquad pair(\alpha, list(\alpha)) \qquad \cdots$$

where α and β are type variables. A typed logic program consists of type declarations for variables, functions and predicates (constants are functions without arguments) and a set of well-typed Horn clauses. The following program defines the polymorphic predicate `member` (throughout this paper we use the Prolog notation for lists [SS86]):

func `[]`: $\rightarrow$ $list(\alpha)$
func `[..|..]`: α, $list(\alpha)$ $\rightarrow$ $list(\alpha)$

pred `member`: α, $list(\alpha)$

vars `L`:$list(\alpha)$, `E, E1`:α

`member(E,[E|L])` $\leftarrow$
`member(E,[E1|L])` $\leftarrow$ `member(E,L)`

The clauses for `member` are well-typed in our sense (cf. section 2) w.r.t. the type definitions.

We view *subtyping* as the possibility of applying a function or predicate to all types which are subtypes of the declared type of the function or predicate. Hence we specify a type that has some subtypes as a function which is the identity on the subtypes. A similar approach has been proposed by Conrad and Furbach [CF88] where sorts are viewed as functions on terms. They realize a sort s as a function s with the property $s(t) = t$ if t is a term of sort s. Since value functions like $+$ and sort functions are merged in their approach, they may obtain results which do not correspond to results in order-sorted logic.[1] Therefore we will have a clear distinction between types (sort functions) and value functions and predicates. Our method will be illustrated by the next example.

Example 1.2 We want to specify a type structure with types *nat*, *zero* and *posint* where *zero* and *posint* are subtypes of *nat*. Hence we specify *nat* as a function on types which is the identity on *zero* and *posint*:

[1] E.g., it is possible to derive the equation `0 = 0/0` if the division function is only specified for second arguments with non-zero values.

$$
\begin{array}{llcl}
\texttt{TYPEOPS} & \mathit{zero}: & \rightarrow & \mathit{type} \\
 & \mathit{posint}: & \rightarrow & \mathit{type} \\
 & \mathit{nat}: \quad \mathit{type} & \rightarrow & \mathit{type} \\
\texttt{TYPEAXIOMS} & \mathit{nat}(\mathit{zero}) & = & \mathit{zero} \\
 & \mathit{nat}(\mathit{posint}) & = & \mathit{posint}
\end{array}
$$

The type axioms state that *nat* is not a free type constructor like *list* but it is the identity on the subtypes of *nat*. It is possible to apply *nat* to other types than *zero* and *posint*, but our logic programs which are based on this specification do not contain any ground terms of type $nat(\tau)$ where $\tau \notin \{zero, posint\}$. Therefore the type $nat(\alpha)$ describes the union of *zero* and *posint* in the initial model of the following program:

func `0`: $\rightarrow \quad zero$
func `s`: $nat(\alpha) \quad \rightarrow \quad posint$

pred `plus`: $nat(\alpha), \quad nat(\beta), \quad nat(\gamma)$

vars `N, N1:`$nat(\alpha)$`, N2:`$nat(\beta)$`, N3:`$nat(\gamma)$

`plus(0,N,N)` $\leftarrow$
`plus(s(N1),N2,s(N3))` $\leftarrow$ `plus(N1,N2,N3)`

The clauses for `plus` are well-typed in our sense (cf. section 2) w.r.t. the type definitions (note that the type of the first argument of the clause head is "*zero*" in the first and "*posint*" in the second clause). Since the argument types of `plus` are defined to be arbitrary naturals, we can apply `plus` with an arbitrary subtype of the naturals. It is possible to build nonsensical types like *nat*(*bool*) (if the basic type *bool* is added to the type structure), but our program does not contain a ground term of this type and therefore such a type denotes an empty set in the initial model of this program. Moreover, our proof procedure (resolution with typed unifiers, cf. section 5) ensures that such types do not occur in the computation if they are not present in the initial goal.

Since order-sorted type structures are polymorphic type specifications with equational axioms which describe the subsort relationship, it is clear that polymorphic and order-sorted type structures can be combined in our framework. It is also possible to express subsort relationships between polymorphic types:

Example 1.3 We want to specifiy a type structure for polymorphic lists so that the polymorphic type *list* is the union of *elist* (empty lists) and *nelist* (non-empty lists). Therefore we have to express the subtype relationships $elist < list(\alpha)$ and $nelist(\alpha) < list(\alpha)$. As in the previous example, we add an additional argument to a type constructor having some subtypes and express the subtype relationship by type equations:

$$
\begin{array}{llcl}
\texttt{TYPEOPS} & \mathit{elist}: & \rightarrow & \mathit{type} \\
 & \mathit{nelist}: \quad \mathit{type} & \rightarrow & \mathit{type} \\
 & \mathit{list}: \quad \mathit{type}, \mathit{type} & \rightarrow & \mathit{type} \\
\texttt{TYPEAXIOMS} & \mathit{list}(\alpha, \mathit{elist}) & = & \mathit{elist} \\
 & \mathit{list}(\alpha, \mathit{nelist}(\alpha)) & = & \mathit{nelist}(\alpha)
\end{array}
$$

The `append`-program is specified w.r.t. this type structure as follows:

func `[]`: $\quad \rightarrow \quad elist$
func `[..|..]`: $\alpha, \quad list(\alpha, \beta) \quad \rightarrow \quad nelist(\alpha)$

pred `append`: $list(\alpha, \beta_1), \quad list(\alpha, \beta_2), \quad list(\alpha, \beta_3)$

vars `R:`$list(\alpha, \beta_1)$`,  L:`$list(\alpha, \beta_2)$`,  RL:`$list(\alpha, \beta_3)$`,  E:`α

`append([],L,L)` $\leftarrow$
`append([E|R],L,[E|RL])` $\leftarrow$ `append(R,L,RL)`

The type variable α in all argument types of `append` expresses that `append` concatenates lists of the same element type whereas the different type variables $\beta_1, \beta_2, \beta_3$ show that an arbitrary subtype of an α-list (empty or non-empty list) can be used in each argument.

The example shows that logic programs with a polymorphically order-sorted type structure are allowed in our framework. Moreover, in section 6 we will give an example of a logic program with higher-order predicates which is well-typed in our framework.

In the following we present our framework for typed logic programming in detail. The main topics of this paper are:

- In our two-level approach to typed logic programming the first level is a specification of the basic type structure, and the second level contains a well-typed logic program which is based on the given type structure. The type structure is specified by a many-sorted signature with equational axioms. In contrast to other approaches to polymorphic type systems for logic programming, we do not restrict the use of types inside program clauses.
- We present a sound and complete resolution method for typed logic programs. For the soundness of the resolution method it is necessary to define the unification procedure on well-typed terms which is based on a unification procedure for the equational type theory. This sheds some new light on the rôle of types in logic programming since the complexity of the type structure directly influences the complexity of the unification procedure. A powerful type structure (e.g., polymorphic types combined with subtypes) implies a complex unification procedure.
- We show that higher-order programming techniques can be applied in our general framework. We give an example of a typed logic program with higher-order predicates which is ill-typed in the sense of other polymorphic type systems for logic programming.
- The presented approach is a framework for the definition of different type structures for logic programs. The type structure influences only the unification procedure for the execution of the program. Therefore different type structures can be used for different applications where the specification of the type structure can be compiled into a specific unification procedure. It is not necessary to use a powerful order-sorted unification procedure for simple applications like those possible in Turbo-Prolog.

This paper is organized as follows. In the next section the basic notions and the syntax of typed logic programs are defined. Section 3 defines the semantics of typed logic programs which is based on interpretations in algebraic structures. Section 4 presents a solution to the unification problem of typed terms which is based on a given unification procedure for the type theory. The unification procedure on typed terms will be used for the resolution method presented in section 5. Section 6 concludes with an interesting application of our framework. Detailed definitions and the proofs of all theorems are contained in the full version [Han89b] of this paper and omitted from this extended abstract.

2 Logic programs with equational type specifications

Since we use (many-sorted) equational logic for the specification of type structures, we assume familiarity with basic notions from algebraic specifications as to be found in [EM85]. We call $\mathcal{T} = (Ts, Top, Tax)$ a **specification of types** if $\mathcal{T}$ is an equational specification, i.e., Ts is a set of sorts (in our examples we have only one sort *type*), Top is a set of operations on these sorts and Tax is a set of equations. Constants from $\mathcal{T}$ are called **basic types**. By X we denote a set of **type variables** and $T_{\mathcal{T}}(X)$ denotes the set (precisely: algebra) of all well-formed terms w.r.t. $\mathcal{T}$ and X. A **type expression** or **type** is a term from $T_{\mathcal{T}}(X)$. A **type substitution** σ is a $\mathcal{T}$-homomorphism $\sigma: T_{\mathcal{T}}(X) \rightarrow T_{\mathcal{T}}(X)$. Two types $\tau_1, \tau_2 \in T_{\mathcal{T}}(X)$ are called **$\mathcal{T}$-equal**, denoted $\tau_1 =_{\mathcal{T}} \tau_2$, if $\tau_1 = \tau_2$ is a consequence of the axioms in $\mathcal{T}$.

A **polymorphic signature** Σ for logic programs is a triple $(\boldsymbol{\mathcal{T}}, \boldsymbol{\mathcal{F}}, \boldsymbol{\mathcal{P}})$ with:

- $\mathcal{T}$ is a specification of types with $T_{\mathcal{T},s} \neq \emptyset$ for all $s \in Ts$.

Variable:	$\dfrac{}{V \Vdash x{:}\tau'}$	$(x{:}\tau \in V$ and $\tau =_{\mathcal{T}} \tau')$
Term:	$\dfrac{V \Vdash t_1{:}\tau_1, \ldots, V \Vdash t_n{:}\tau_n}{V \Vdash f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau'}$	$(f{:}\tau_f \in \mathcal{F}$ so that there exists a type substitution σ with $\sigma(\tau_f) = \tau_1, \ldots, \tau_n \to \tau$ and $\tau =_{\mathcal{T}} \tau'$, $n \geq 0)$
Atom:	$\dfrac{V \Vdash t_1{:}\tau_1, \ldots, V \Vdash t_n{:}\tau_n}{V \Vdash p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)}$	$(p{:}\tau_p \in \mathcal{P}$ so that there exists a type substitution σ with $\sigma(\tau_p) = \tau_1, \ldots, \tau_n$, $n \geq 0)$
Goal:	$\dfrac{V \Vdash L_1, \ldots, V \Vdash L_n}{V \Vdash L_1, \ldots, L_n}$	(each L_i is an atom, i.e., has the form $p(\cdots)$, $i = 1, \ldots, n$)
Clause:	$\dfrac{V \Vdash L,\ V \Vdash G}{V \Vdash L \leftarrow G}$	(L is an atom and G is a goal)

Figure 1: Typing rules for program clauses

- $\mathcal{F}$ is a set of **function declarations** of the form $f{:}\tau_1, \ldots, \tau_n \to \tau$ with $\tau_i, \tau \in T_{\mathcal{T}}(X)$, $n \geq 0$.
- $\mathcal{P}$ is a set of **predicate declarations** of the form $p{:}\tau_1, \ldots, \tau_n$ with $\tau_i \in T_{\mathcal{T}}(X)$ $(n \geq 0)$.

The type specifications together with the definitions of function and predicate types in example 1.1, 1.2 and 1.3 are examples for polymorphic signatures. In the rest of this paper we assume that $\Sigma = (\mathcal{T}, \mathcal{F}, \mathcal{P})$ is a polymorphic signature for logic programs. Similarly to other typed logics, variables in a typed 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: Let $\boldsymbol{Var}$ be an infinite set of variable names that are distinguishable from symbols in polymorphic signatures and type variables. Then a set V of elements of the form $x{:}\tau$ where $x \in Var$ and $\tau \in T_{\mathcal{T}}(X)$ is called a **set of typed variables** if $x{:}\tau, x{:}\tau' \in V$ implies $\tau = \tau'$. We only consider sets of typed variables with unique types so that type errors can be detected at compile time. For instance, if a variable in a clause occurs in two different contexts so that it has type "*int*" in one context and type "*list*(*int*)" in the other context, this indicates a type error if all variables in a clause are required to have unique types. In the rest of this paper we assume that $V, V', V_0, V_1, \ldots$ denote sets of typed variables.

We embed types in terms, i.e., each symbol in a term is annotated with an appropriate type expression. These annotations are useful for the unification of typed terms (see section 4.2). We call $L \leftarrow G$ a *typed program clause* if there is a set of typed variables V and $V \Vdash L \leftarrow G$ is derivable by the inference rules in figure 1. The typing rules show that both parametric polymorphism and subtype polymorphism are covered by our framework: If the declared type of a function or predicate contains type variables, then this function or predicate can be applied to any type which is the result of replacing the type variables by other types (parametric polymorphism). If the type specification contains subtype relations as in example 1.2, then a function or predicate with declared argument type $nat(\alpha)$ can also be applied to the subtypes $nat(zero)$ $(=_{\mathcal{T}} zero)$ and $nat(posint)$ $(=_{\mathcal{T}} posint)$.

Note that we have no restrictions on the use of types and type variables in the left-hand side of program clauses in contrast to [MO84] [Smo89] and similar polymorphic type systems.[2] For instance, it is allowed to add the clause

`member(2,[1,2,3])` $\leftarrow$

to the program in example 1.1. By dropping this restriction it is also possible to apply higher-order programming techniques in our framework (cf. section 6).

We call variables, constants and composite terms derivable by these inference rules $(\boldsymbol{\Sigma, X, V})$**-terms** or **well-typed terms**. $\boldsymbol{Term}_{\Sigma}(\boldsymbol{X, V})$ denotes the set of all (Σ, X, V)-terms. Well-typed or (Σ, X, V)-atoms, -goals and -clauses are similarly defined (a goal is a set of atoms, but for convenience we denote it without

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

curly brackets). A Σ-**term** (atom, goal, clause) is a (Σ, X, V)-term (atom, goal, clause) for some set of typed variables V.

In the following, if t is a syntactic construction (type, term, atom, ...), $tvar(t)$ and $var(t)$ will denote the set of type variables and typed variables that occur in t, respectively (i.e., $var(t)$ is a set of typed variables so that t is a $(\Sigma, X, var(t))$-term, atom, ...). For instance, if $Tax = \{s_1(s_3) = s_3, s_2(s_3) = s_3\}$ and $t = f(X{:}s_1(s_3), X{:}s_2(s_3)){:}s_3$, then both $\{X{:}s_3\}$ and $\{X{:}s_1(s_3)\}$ satisfy the definition of $var(t)$, but it is always the case that these different sets are $\mathcal{T}$-equal sets of typed variables. Therefore we can choose one of these sets as $var(t)$.

A **typed logic program** or **typed Horn clause program** $P = (\Sigma, C)$ consists of a polymorphic signature Σ and a set C of Σ-clauses.

Corollary 2.1 *If $t{:}\tau$ is a well-typed term and $\tau =_{\mathcal{T}} \tau'$, then $t{:}\tau'$ is also a well-typed term.*

Since it is clear form the context, we will omit the type annotations in the clauses of example programs. Therefore we have written the clauses of the examples in the first chapter without type annotations but we have defined the types of the variables. For instance, the clause

`plus(0,N,N)` $\leftarrow$

in example 1.2 denotes the fully typed clause

`plus(0:`$nat(zero)$`,N:`$nat(\alpha)$`,N:`$nat(\alpha)$`)` $\leftarrow$

This clause is well-typed because "$nat(zero), nat(\alpha), nat(\alpha)$" is an instance of the declared type "$nat(\alpha)$, $nat(\beta)$, $nat(\gamma)$" of the predicate `plus` and `0:`$nat(zero)$ is a well-typed term since $nat(zero) =_{\mathcal{T}} zero$ (where $\mathcal{T}$ is the type specification of example 1.2).

3 Semantics of typed logic programs

Typed logic programs are interpreted by algebraic structures similar to [Poi86]. An interpretation of a typed logic program consists of an algebra that satisfies the type specification 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 relations between these sets. Type variables vary over all types of the interpretation and typed variables vary over appropriate carrier sets. We outline the necessary notions.

If $\mathcal{T} = (Ts, Top, Tax)$ is a specification of types, a $\mathcal{T}$-algebra $A = (Ts_A, Top_A)$ which satisfies all equations from Tax is also called **$\mathcal{T}$-type algebra**. The **signature** $\Sigma(A) = (Ts_A, \mathcal{F}_A, \mathcal{P}_A)$ **derived from** Σ **and** A is defined by

$$\begin{aligned} \mathcal{F}_A &:= \{f{:}\sigma(\tau_f) \mid f{:}\tau_f \in \mathcal{F},\ \sigma{:}\, X \to Ts_A \text{ is a type variable assignment}\} \\ \mathcal{P}_A &:= \{p{:}\sigma(\tau_p) \mid p{:}\tau_p \in \mathcal{P},\ \sigma{:}\, X \to Ts_A \text{ is a type variable assignment}\} \end{aligned}$$

An **interpretation** of a polymorphic signature Σ is a $\mathcal{T}$-type algebra $A = (Ts_A, Top_A)$ together with a $\Sigma(A)$-structure (S, δ), which consists of a Ts_A-sorted set S (the **carrier** of the interpretation) and a denotation δ with:

1. If $f{:}\tau_1, \ldots, \tau_n \to \tau \in \mathcal{F}_A$, 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 $p{:}\tau_1, \ldots, \tau_n \in \mathcal{P}_A$, then $\delta_{p:\tau_1,\ldots,\tau_n} \subseteq S_{\tau_1} \times \cdots \times S_{\tau_n}$ is a relation.

If A and A' are $\mathcal{T}$-type algebras, then every $\mathcal{T}$-homomorphism $\sigma{:}\, A \to A'$ induces a **signature morphism** $\sigma{:}\, \Sigma(A) \to \Sigma(A')$ and a **forgetful functor** $U_\sigma{:}\, Cat_{\Sigma(A')} \to Cat_{\Sigma(A)}$ from the category of $\Sigma(A')$-structures into the category of $\Sigma(A)$-structures (for details, see [EM85]). Therefore we 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 a $\mathcal{T}$-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.

A homomorphism in our typed framework consists of a mapping between type algebras and a mapping between appropriate structures. Consequently, a variable assignment in the typed framework maps type variables into types and typed variables into objects of appropriate types: If $I = ((Ts_A, Top_A), S, \delta)$ is a Σ-interpretation, then a **variable assignment** for (X, V) in I is a pair of mappings $v = (v_X, v_V)$ where $v_X: X \to Ts_A$ is a type variable assignment and $v_V: V \to S'$ with $(S', \delta') := U_{v_X}((S, \delta))$ and $v_V(x{:}\tau) \in S'_\tau$ $(= S_{v_X(\tau)})$ for all $x{:}\tau \in V$.

In many-sorted logic, a canonical interpretation for a signature is the *term interpretation* where the carrier sets consist of well-typed terms. In a term interpretation every variable assignment can be uniquely extended to a homomorphism. In our typed framework the situation is more complicated because a variable may correspond to syntactically different terms. For instance, if $s_1 = s_2 \in Tax$, then the variable $x{:}s_1 \in V$ corresponds to the (Σ, X, V)-terms $x{:}s_1$ and $x{:}s_2$. In order to identify such syntactically different terms, we define *canonical terms* as terms where type annotations are replaced by equivalence classes of types. For this purpose we define a mapping $\mathcal{C}$ which replaces all type annotations in a typed term by equivalence classes of types ($[\tau]$ denotes the equivalence class of the type τ defined by $[\tau] = \{\tau' \mid \tau =_{\mathcal{T}} \tau'\}$):
$\mathcal{C}(x{:}\tau') := x{:}[\tau]$ for all $x{:}\tau \in V$ and $\tau' =_{\mathcal{T}} \tau$
$\mathcal{C}(f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau) := f(\mathcal{C}(t_1{:}\tau_1), \ldots, \mathcal{C}(t_n{:}\tau_n)){:}[\tau]$ for all $f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau \in Term_\Sigma(X, V)$ $(n \geq 0)$
$\boldsymbol{CTerm_\Sigma(X, V)} := \{\mathcal{C}(t{:}\tau) \mid t{:}\tau \in Term_\Sigma(X, V)\}$ is the set of **canonical terms**. Now we are able to define the **canonical term interpretation** $\boldsymbol{T_\Sigma(X, V)}$ over X and V: $T_\Sigma(X, V) := (T_{Tax}(X), S, \delta)$, where

1. $T_{Tax}(X) := T_{\mathcal{T}}(X)/\equiv_{Tax}$ is the quotient of the algebra of type expressions by the congruence relation $\equiv_{Tax}$ generated by the axioms in the type specification $\mathcal{T} = (Ts, Top, Tax)$ (the elements of the domain of $T_{Tax}(X)$ are equivalence classes of types).
2. $S_{[\tau]} := \{t{:}[\tau] \mid t{:}[\tau] \in CTerm_\Sigma(X, V)\}$ for all $[\tau] \in T_{Tax}(X)$,
3. If $f{:}[\tau_1], \ldots, [\tau_n] \to [\tau] \in \mathcal{F}_{T_{Tax}(X)}$ and $t_i{:}[\tau_i] \in S_{[\tau_i]}$ for $i = 1, \ldots, n$, then
$$\delta_{f:[\tau_1],\ldots,[\tau_n]\to[\tau]}(t_1{:}[\tau_1], \ldots, t_n{:}[\tau_n]) := f(t_1{:}[\tau_1], \ldots, t_n{:}[\tau_n]){:}[\tau]$$
4. $\delta_{p:[\tau_1],\ldots,[\tau_n]} := \emptyset$ for all $p{:}[\tau_1], \ldots, [\tau_n] \in \mathcal{P}_{T_{Tax}(X)}$.

The mappings $\delta_{f:[\tau_1],\ldots,[\tau_n]\to[\tau]}$ in the definition are well-defined by corollary 2.1. Now we are able to state that any variable assignment can be uniquely extended to a homomorphism:

Lemma 3.1 *Let (A, S, δ) be a Σ-interpretation and $v = (v_X, v_V)$ be an assignment 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 $\sigma([\alpha]) = v_X(\alpha)$ for all $\alpha \in X$ and $h(x{:}[\tau]) = v_V(x{:}\tau)$ for all $x{:}\tau \in V$.*

This lemma is only valid if $T_\Sigma(X, V)$ and the $\mathcal{T}$-algebra A satisfies all equations from Tax. If this is not the case, there exist several different Σ-homomorphisms which extend the variable assignment. For instance, if $s_1 = s_2 \in Tax$ and A has different interpretations of the sorts s_1 and s_2, then the terms $x{:}s_1$ and $x{:}s_2$ may be mapped into different values by different homomorphisms, provided that $x{:}s_1 \in V$.

As a special case $(X = V = \emptyset)$ the lemma shows that every ground term without type variables corresponds to a unique value in a given Σ-interpretation. Generally, any variable assignment v can be extended to a Σ-homomorphism in a unique way. In the following we denote this Σ-homomorphism again by v. Since v_X and v_V are only applied to equivalence classes of type expressions and canonical terms, respectively, we omit the indices X and V and write v for both v_X and v_V.

We are not interested in all interpretations of a polymorphic signature but only in those interpretations that satisfy the clauses of a given typed 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 be an assignment for (X, V) in I.

  $\boldsymbol{I, v \models L}$ if $L = p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)$ is a (Σ, X, V)-atom with $(v(\mathcal{C}(t_1{:}\tau_1)), \ldots, v(\mathcal{C}(t_n{:}\tau_n))) \in \delta'_{p:[\tau_1],\ldots,[\tau_n]}$ where $U_v((S, \delta)) = (S', \delta')$, i.e., $\delta'_{p:[\tau_1],\ldots,[\tau_n]} = \delta_{p:v([\tau_1]),\ldots,v([\tau_n])}$.

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

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

- $\boldsymbol{I, V} \models \boldsymbol{\mathcal{F}}$ 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 $I = (A, S, \delta)$ is called **model** for a typed logic program (Σ, C) if all clauses from C are valid in I. A (Σ, X, V)-goal G is called **valid in** (Σ, C) relative to V if $I, V \models G$ for every model I of (Σ, C). We shall write: $(\boldsymbol{\Sigma}, \boldsymbol{C}, \boldsymbol{V}) \models \boldsymbol{G}$.

This notion of validity is the extension of validity in untyped Horn clause logic to the typed 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 typed case an atom, goal or clause is said to be true iff it is true for all assignments of type variables and typed variables. The reason for the definition of validity relative to a set of variables is that carrier sets in our interpretations may be empty in contrast to untyped Horn logic. This is also the case in many-sorted logic [GM84]. Validity relative to variables is different from validity in the sense of untyped logic. An example for such a difference can be found in [Han89a], p. 231. Validity in our sense is equivalent to validity in the sense of untyped logic if the types of the variables denote non-empty sets in all interpretations. But a requirement for non-empty carrier sets is not reasonable in the context of polymorphic types. Similarly to untyped Horn clause logic it can be shown that *there exists an initial model for any typed logic program.* The carrier set of this initial model contains all canonical terms without type variables and typed variables.

Example 3.2 The following interpretation is a model for the program of example 1.2. The type specification is interpreted by the $\mathcal{T}$-type algebra $A = (Ts_A, Top_A)$ where $Ts_A = \{nat, zero, posint\}$ and Top_A contains the functions $zero_A$ with $zero_A() = zero$, $posint_A$ with $posint_A() = posint$, and nat_A with $nat_A(\tau) = \tau$ for all $\tau \in Ts_A$. The carrier sets of the interpretation are:

$S_{zero} = \{0\}$
$S_{posint} = \{ n \in Nat \mid n > 0 \}$
$S_{nat} = S_{zero} \cup S_{posint}$

The constant `0` and the function `s` are interpreted as follows:

$\delta_{0:\rightarrow\ zero} = 0$
$\delta_{s:zero\ \rightarrow\ posint}(0) = 1$
$\delta_{s:posint\ \rightarrow\ posint}(n) = n+1$ for all $n \in S_{posint}$
$\delta_{s:nat\ \rightarrow\ posint}(n) = n+1$ for all $n \in S_{nat}$
$\delta_{plus:nat,\ nat,\ nat} = \{(n_1, n_2, n_3) \in Nat^3 \mid n_1 + n_2 = n_3\}$
...

The remaining interpretations of `plus` are the restriction of $\delta_{plus:nat,nat,nat}$ to appropriate subsets. It is easy to show that this interpretation is a model.

4 Unification

In order to define the semantics of typed logic programs we have used canonical terms which are annotated with equivalence classes of types. Since these equivalence classes are sets which may contain an infinite number of elements, this representation is unsuitable for a proof procedure like resolution. The resolution procedure should work on well-typed terms which can be easily handled. Therefore we have to define substitutions on well-typed terms and introduce a relation on well-typed terms that establishes the link to canonical terms.

4.1 Typed substitutions

Let $\mu: X \to T_{\mathcal{T}}(X)$ be a type variable assignment and $val: V \to Term_{\Sigma}(X, V')$ be a mapping from typed variables into well-typed terms over X and V' with $val(x{:}\tau) = t{:}\mu(\tau)$ for all $x{:}\tau \in V$, i.e., typed variables of sort τ are mapped into well-typed terms of type $\mu(\tau)$. We extend the mappings μ and val to mappings on types and well-typed terms, respectively, in the following way:

- $\mu(h(\tau_1,\ldots,\tau_n)) = h(\mu(\tau_1),\ldots,\mu(\tau_n))$ for all n-ary operation symbols h in $\mathcal{T}$ ($n \geq 0$) and all appropriate types $\tau_1,\ldots,\tau_n \in T_{\mathcal{T}}(X)$.
- $val(x{:}\tau') = t{:}\mu(\tau')$ for all $x{:}\tau \in V$ with $val(x{:}\tau) = t{:}\mu(\tau)$ and $\tau' =_{\mathcal{T}} \tau$.
- $val(f(t_1{:}\tau_1,\ldots,t_n{:}\tau_n){:}\tau) = f(val(t_1{:}\tau_1),\ldots,val(t_n{:}\tau_n)){:}\mu(\tau)$ for all (non-variable) well-typed terms $f(t_1{:}\tau_1,\ldots,t_n{:}\tau_n){:}\tau \in Term_\Sigma(X,V)$, $n \geq 0$.

It is easy to show that val maps well-typed terms into well-typed terms by this definition. The mappings are similarly extended on atoms, goals and clauses. We call (μ, val) a **typed substitution**. $\boldsymbol{Sub_\Sigma(X,V,V')}$ denotes the class of all typed substitutions from $(T_{\mathcal{T}}(X), Term_\Sigma(X,V))$ into $(T_{\mathcal{T}}(X), Term_\Sigma(X,V'))$. 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 section 3). Sometimes we represent typed substitutions by sets. For instance, the set

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

represents a typed substitution that replaces the type variable α by the type nat and the typed variable $x{:}\alpha$ by the 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 states the relationship between typed substitutions and Σ-homomorphisms on canonical term interpretations:

Lemma 4.1 *Let $(\mu, val) \in Sub_\Sigma(X,V,V')$ be a typed substitution. Then there exists a unique Σ-homomorphism σ from $T_\Sigma(X,V)$ into $T_\Sigma(X,V')$ with*

- $\sigma([\alpha]) = [\mu(\alpha)]$ *for all* $\alpha \in X$
- $\sigma(x{:}[\tau]) = \mathcal{C}(val(x{:}\tau))$ *for all* $x{:}\tau \in V$

We want to relate terms that are "equal w.r.t. the given type structure". For that purpose we define an important relation on well-typed terms: Two Σ-terms t and t' are called **$\mathcal{T}$-equal**, denoted $t =_{\mathcal{T}} t'$, if $\mathcal{C}(t) = \mathcal{C}(t')$. $\mathcal{T}$-equality on atoms is analogously defined. For instance, if the type specification of example 1.2 is given, then `0`:$nat(zero) =_{\mathcal{T}}$ `0`:$zero$ and `N`:$posint =_{\mathcal{T}}$ `N`:$nat(posint)$. The next lemma shows that $\mathcal{T}$-equal atoms have the same meaning in all interpretations:

Lemma 4.2 *Let Σ be a polymorphic signature, V be a set of typed variables, and L_1 and L_2 be two $\mathcal{T}$-equal (Σ,X,V)-atoms. If I is a Σ-interpretation, then:* $I,V \models L_1 \iff I,V \models L_2$

4.2 A unification procedure for typed terms

The basic operation in a resolution step is the computation of a unifier for two atoms (see next section). As in order-sorted logic, the unification problem is not unitary in our general framework and therefore complete sets of unifiers must be considered. This section defines the unification w.r.t. a type specification $\mathcal{T}$ and presents a non-deterministic algorithm for computing complete sets of unifiers.

Example 4.3 Consider example 1.2. The first clause for `plus`

`plus(0:`$nat(zero)$`,N:`$nat(\alpha)$`,N:`$nat(\alpha)$`)` $\leftarrow$

cannot be applied to prove the goal

`plus(N1:`$nat(posint)$`,N2:`$nat(\beta)$`,N3:`$nat(\gamma)$`)`

since this would cause the binding of variable `N1` to `0` which yields the ill-typed term `0`:$nat(posint)$. In order to avoid such bindings, the unification procedure has to take into account that `N1` and `0` have the non-unifiable types $nat(posint)$ and $nat(zero)$. On the other hand, if the clause

`p(N:`$nat(zero)$`)` $\leftarrow \cdots$

is applied to prove the goal

`p(N1:`$nat(\alpha)$`)`

then the variable `N1` is constrained to type $nat(zero)$ which may avoid some unnecessary search and backtracking steps in the subsequent proof. Therefore the unification procedure has to consider the types of the terms. An untyped unification cannot be applied in our framework.

$$\text{(T)} \quad (\sigma,\ \langle t_1{:}\tau_1 \doteq t_2{:}\tau_2, E_r\rangle) \quad \overset{unif}{\Longrightarrow} \quad (\phi \circ \sigma,\ \langle \phi(t_1{:}\tau_1) \doteq \phi(t_2{:}\tau_2), \phi(E_r)\rangle)$$
$$\text{if } \phi \in CSU_{\mathcal{T}}(\tau_1, \tau_2)$$

$$\text{(E1)} \quad (\sigma,\ \langle x{:}\tau \doteq t{:}\tau', E_r\rangle) \quad \overset{unif}{\Longrightarrow} \quad (\sigma' \circ \sigma,\ \sigma'(E_r))$$
$$\text{if } \tau =_{\mathcal{T}} \tau',\ x \in Var,\ x \text{ does not occur in } t{:}\tau' \text{ and } \sigma' = \{x{:}\tau / t{:}\tau\}$$

$$\text{(E2)} \quad (\sigma,\ \langle t{:}\tau' \doteq x{:}\tau, E_r\rangle) \quad \overset{unif}{\Longrightarrow} \quad (\sigma' \circ \sigma,\ \sigma'(E_r))$$
$$\text{if } \tau =_{\mathcal{T}} \tau',\ x \in Var,\ x \text{ does not occur in } t{:}\tau' \text{ and } \sigma' = \{x{:}\tau / t{:}\tau\}$$

$$\text{(D)} \quad (\sigma,\ \langle f(t_1, \ldots, t_n){:}\tau \doteq f(t'_1, \ldots, t'_n){:}\tau', E_r\rangle) \quad \overset{unif}{\Longrightarrow} \quad (\sigma,\ \langle t_1 \doteq t'_1, \ldots, t_n \doteq t'_n, E_r\rangle)$$
$$\text{if } \tau =_{\mathcal{T}} \tau' \ (n \geq 0)$$

Figure 2: Rules for $\mathcal{T}$-unification of well-typed terms. In the first rule (T) the type substitution ϕ is extended to a typed substitution by $\phi(x{:}\tau) := x{:}\phi(\tau)$ for all $x{:}\tau \in V'$ if $\sigma \in Sub_\Sigma(X, V, V')$.

We have mentioned that the resolution procedure should manipulate well-typed terms rather than canonical terms. Therefore we have introduced *typed substitutions* which are mappings on type expressions and well-typed terms and directly related to Σ-homomorphisms between canonical term interpretations. Hence we want to define a unifier w.r.t. a type specification $\mathcal{T}$ as a distinct typed substitution. Since the composition of two typed substitutions is again a typed substitution, we can define the following notions (we assume that V, V_1, V_2 are sets of typed variables):

- Let $\sigma, \sigma' \in Sub_\Sigma(X, V, V_1)$ be typed substitutions. We write $\sigma =_{\mathcal{T}} \sigma'$ iff $\sigma(\alpha) =_{\mathcal{T}} \sigma'(\alpha)$ for all $\alpha \in X$ and $\sigma(x{:}\tau) =_{\mathcal{T}} \sigma'(x{:}\tau)$ for all $x{:}\tau \in V$.
- Let $\sigma \in Sub_\Sigma(X, V, V_1)$ and $\sigma' \in Sub_\Sigma(X, V, V_2)$ be typed substitutions. σ is **more general** than σ' w.r.t. $\mathcal{T}$, denoted $\sigma \leq_{\mathcal{T}} \sigma'$, iff there exists $\phi \in Sub_\Sigma(X, V_1, V_2)$ with $\phi \circ \sigma =_{\mathcal{T}} \sigma'$.
- The (Σ, X, V)-terms t and t' are **$\mathcal{T}$-unifiable** if there exists a typed substitution $\sigma \in Sub_\Sigma(X, V, V')$ with $\sigma(t) =_{\mathcal{T}} \sigma(t')$ for a set of typed variables V'. In this case σ is called a **$\mathcal{T}$-unifier** for t and t'. By $\boldsymbol{SU_{\mathcal{T}}(t, t')}$ we denote the set of all $\mathcal{T}$-unifiers for t and t'.
- Let t and t' be (Σ, X, V)-terms. We call a set of typed substitutions $\boldsymbol{CSU_{\mathcal{T}}(t, t')}$ a **complete set of $\mathcal{T}$-unifiers** for t and t' if the following conditions hold:
  - $CSU_{\mathcal{T}}(t, t') \subseteq SU_{\mathcal{T}}(t, t')$
  - For all $\sigma' \in SU_{\mathcal{T}}(t, t')$ there exists a typed substitution $\sigma \in CSU_{\mathcal{T}}(t, t')$ with $\sigma \leq_{\mathcal{T}} \sigma'$.

$\mathcal{T}$-unifiers and complete sets of $\mathcal{T}$-unifiers for type expressions are analogously defined as particular (sets of) type substitutions.

Obviously, the set of all $\mathcal{T}$-unifiers is also a complete set of $\mathcal{T}$-unifiers, but usually we are interested in algorithms which enumerate a complete set of $\mathcal{T}$-unifiers with some minimality condition. We do not discuss this in detail here. We assume a given algorithm that enumerates a complete set of $\mathcal{T}$-unifiers for two arbitrary type expressions and construct an algorithm which enumerates a complete set of $\mathcal{T}$-unifiers for two arbitrary well-typed terms. We formulate the algorithm as a non-deterministic procedure for computing a $\mathcal{T}$-unifier for a given list of pairs of well-typed terms.

For this purpose we define a binary relation $\overset{unif}{\Longrightarrow}$ on pairs of the form (σ, E) where σ is a typed substitution and E is a list of appropriate equations, i.e., if $\sigma \in Sub_\Sigma(X, V, V')$ then E is a list of pairs of (Σ, X, V')-terms. We write $\langle t \doteq t', E_r\rangle$ for an equation list where the pair (t, t') is the first equation and E_r is the list of the remaining equations. The relation $\overset{unif}{\Longrightarrow}$ is defined by the rules in figure 2. In the first rule (T) the result types of the left-hand side and the right-hand side of the first equation are unified by a $\mathcal{T}$-unifier, i.e., the result types are $\mathcal{T}$-equal after an application of this rule. $\mathcal{T}$-equality of these result types is a precondition for the applicability of the other rules. The rules (E1) and (E2) eliminate an equation containing a variable in one side. The typed substitution σ' in these elimination rules is well-defined since $t{:}\tau$ is well-typed by $\tau =_{\mathcal{T}} \tau'$ and corollary 2.1. The rule (D) decomposes an equation if the left-hand side and the right-hand side are compound terms with the same main functor and arity.

Let $\overset{unif^+}{\Longrightarrow}$ be the transitive closure of $\overset{unif}{\Longrightarrow}$. The result of unifying the (Σ, X, V)-terms t and t' is the set

$$Unif(t,t') \;:=\; \{\ \sigma \mid (id, \langle t \doteq t' \rangle) \overset{unif^+}{\Longrightarrow} (\sigma, \langle\rangle)\ \}$$

where id is the identity in $Sub_\Sigma(X, V, V)$ and $\langle\rangle$ denotes the empty list of equations.

Note that $\overset{unif^+}{\Longrightarrow}$ is an extension of Robinson's unification algorithm [Rob65]: If one term is a variable which does not occur in the other term, then this variable is bound to the other term. If two composite terms have to be unified, then all corresponding components of the terms are unified. The only (but essential) difference is that the types of two terms are $\mathcal{T}$-unified before the terms will be unified.

It is easy two show that any $\overset{unif}{\Longrightarrow}$-sequence terminates. Moreover, $Unif(t,t')$ is a complete set of $\mathcal{T}$-unifiers for t and t':

Theorem 4.4 ($\mathcal{T}$-unification) *Let t and t' be (Σ, X, V)-terms. Then $Unif(t,t')$ is a complete set of $\mathcal{T}$-unifiers.*

Example 4.5 Consider the polymorphic signature of example 1.2. The terms $\texttt{0}{:}zero$ and $\texttt{N}{:}nat(\alpha)$ should be unified by our unification procedure. First, the types of terms $zero$ and $nat(\alpha)$ are $\mathcal{T}$-unified and the result is the $\mathcal{T}$-unifier $\{\alpha/zero\}$. Then `N` is bound to `0` and the result is the $\mathcal{T}$-unifier $\{\alpha/zero,\ \texttt{N}{:}nat(\alpha)/\texttt{0}{:}nat(zero)\}$. For the unification of the terms $\texttt{s(N1}{:}nat(posint)\texttt{)}{:}posint$ and $\texttt{s(N2}{:}nat(\alpha)\texttt{)}{:}nat(posint)$ the following steps are performed:

- The types $posint$ and $nat(posint)$ are $\mathcal{T}$-unified. The result is the identity on type expressions since these types are $\mathcal{T}$-equal.
- The terms $\texttt{N1}{:}nat(posint)$ and $\texttt{N2}{:}nat(\alpha)$ are unified in the next unification step.
- The types $nat(posint)$ and $nat(\alpha)$ are $\mathcal{T}$-unified. The result is the type substitution $\{\alpha/posint\}$.
- `N2` is bound to `N1` (or vice versa). Thus the complete result of the unification is the typed substitution

$$\{\alpha/posint,\ \texttt{N2}{:}nat(\alpha)/\texttt{N1}{:}nat(posint)\}$$

Example 4.6 Consider the following type specification $\mathcal{T}$:

TYPEOPS	s_0:		$\rightarrow$	$type$
	s_1:	$type$	$\rightarrow$	$type$
	s_2:	$type$	$\rightarrow$	$type$
TYPEAXIOMS	$s_1(s_0)$	$=$	s_0	
	$s_2(s_0)$	$=$	s_0	

Thus s_0 is a common subtype of s_1 and s_2. The unification of the typed terms $\texttt{X}{:}s_1(\alpha)$ and $\texttt{Y}{:}s_2(\beta)$ requires a $\mathcal{T}$-unifier for the type expressions $s_1(\alpha)$ and $s_2(\beta)$ which can be computed by the narrowing procedure (see remarks at the end of section 5). Hence the type substitution $\{\alpha/s_0, \beta/s_0\}$ is a $\mathcal{T}$-unifier for the type expressions $s_1(\alpha)$ and $s_2(\beta)$ and the typed substitution

$$\{\alpha/s_0,\ \beta/s_0,\ \texttt{X}{:}s_1(\alpha)/\texttt{Y}{:}s_1(s_0)\}$$

is a $\mathcal{T}$-unifier for the terms $\texttt{X}{:}s_1(\alpha)$ and $\texttt{Y}{:}s_2(\beta)$. Therefore the variables `X` and `Y` are constrained to the common subsort s_0 by the unification procedure (note the analogy to order-sorted unification [SNGM87]).

In the next section we will see that resolution is a sound and complete proof procedure for typed logic programs if the unification procedure used in the resolution steps computes a complete set of $\mathcal{T}$-unifiers. Therefore the unification procedure presented in this section gives us some information about the rôle of different type systems for logic programming. We have seen that the classical unification algorithm of Robinson can be adapted to the typed framework if the types of terms are unified before unifying the terms. Hence different type structures influences the complexity of the unification procedure. For the general case a complex procedure for the unification of type terms w.r.t. the equational type specification is necessary. But for simpler type structures a less complex unification procedure may be sufficient:

- If the type structure is *many-sorted* without overloading, i.e., there are only basic types and no equations in the type structure and there is exactly one type declaration for each function and predicate symbol, then all types can be omitted while unifying two terms or atoms since two composite terms or atoms with the same functor or predicate, respectively, have always the same type.
- If the type structure is *polymorphic* without equations between types, then the $\mathcal{T}$-unifier for two types is the unifier of the type expressions in the free type term algebra. Hence there exists a most general unifier for two unifiable type terms which can be computed by Robinson's unification algorithm. This implies the existence of a most general unifier for two $\mathcal{T}$-unifiable typed terms and Robinson's unification algorithm can be used as a $\mathcal{T}$-unification procedure on typed terms if type expressions are represented as first-order terms (cf. [Han89a]). Moreover, if the polymorphic signature and the typed program satisfy some additional restrictions, it can be shown that such programs are executable without any type information at run time [Han89c]. The type system of Mycroft and O'Keefe [MO84] is a special case of a polymorphic type structure.
- If the type structure is *order-sorted*, i.e., the type specification contains equations between types, then there does not exist a most general $\mathcal{T}$-unifier for any two type expressions. Hence the $\mathcal{T}$-unification procedure on typed terms must compute complete sets of $\mathcal{T}$-unifiers. Nevertheless, for practical applications it is desirable that the complete sets of $\mathcal{T}$-unifiers are finite which depends on the type specification. Criteria for finitary or unitary order-sorted unification can be found in [Wal89].
- *Polymorphically order-sorted* type structures require a full unification procedure for the equational type theory. Nevertheless, Smolka [Smo89] has shown that there are also restricted classes of polymorphically order-sorted typed logic programs where more efficient unification procedures exist.

From a conceptual point of view our unification procedure shows up the influence of types in logic programming. But for an efficient operational semantics it is necessary to omit type information at run time whenever it is possible. In [Han89a] and [Han89c] it is shown how this could be done in the polymorphic case. Similar results for the general case are a topic for further research.

5 Resolution

The resolution principle in untyped Horn logic (see [Rob65]) can be used as a proof procedure for typed Horn clause programs if the untyped unification is replaced by the $\mathcal{T}$-unification as defined in the last section. "$(\Sigma, C, V) \vdash_{\mathrm{R}} \sigma\ G$" denotes a successful resolution of the start goal G with the typed substitution σ as the computed answer, where (Σ, C) is the typed logic program, V is the set of typed variables used in the resolution, and a $\mathcal{T}$-unifier from a complete set of $\mathcal{T}$-unifiers for an atom in the goal and a clause head is computed in each resolution step. The following theorem states soundness and completeness of resolution with $\mathcal{T}$-unifiers:

Theorem 5.1 *Let (Σ, C) be a typed logic program, V be a finite set of typed variables and G be a (Σ, X, V)-goal.*

1. *If there is a successful resolution $(\Sigma, C, V) \vdash_{\mathrm{R}} \sigma\ G$ with computed answer $\sigma \in Sub_{\Sigma}(X, V, V')$, then $(\Sigma, C, V') \models \sigma(G)$.*
2. *If $\sigma \in Sub_{\Sigma}(X, V, V')$ is a typed substitution with $(\Sigma, C, V') \models \sigma(G)$, then there exist a set of typed variables V_0 and a typed substitution $\sigma_0 \in Sub_{\Sigma}(X, V_0, V_1)$ with $(\Sigma, C, V_0) \vdash_{\mathrm{R}} \sigma_0\ G$, and there is a typed substitution $\phi \in Sub_{\Sigma}(X, V_1, V')$ with $\phi(\sigma_0(G)) =_{\mathcal{T}} \sigma(G)$.*

This theorem justifies the implementation of resolution with $\mathcal{T}$-unifiers as a proof method for typed logic programs. A complete resolution method must enumerate all possible derivations. If we use a backtracking method like Prolog, the resolution method becomes incomplete because of infinite derivations (in our typed framework the search tree may have an infinite depth as well as an infinite breadth because $CSU_{\mathcal{T}}(L, L')$ may be an infinite set). If we accept this drawback, we can implement the resolution like Prolog with the difference that the unification is extended to typed terms. In section 4.2 we have shown that the classical unification algorithm can be used if the types of the terms are unified before unifying the terms. For the unification of type expressions w.r.t. the type specification a unification procedure for equational theories is needed. It is known that the narrowing procedure [Fay79] (a combination of unification and term rewriting) can be used for this purpose if the set of equations is a canonical (i.e., confluent and terminating) term rewriting system. A set of equations can be transformed into a canonical term rewriting system by the Knuth-Bendix procedure [KB70] which is successful for our applications. For instance, let $\mathcal{T}$ be a type structure for integer numbers with appropriate subtype relationships, i.e., *zero* and *posint* are subtypes of the natural numbers, and the negative integers and the natural numbers are subtypes of the integer numbers. Therefore $\mathcal{T}$ is the following equational specification:

`TYPEOPS`	$zero$:		$\rightarrow$	$type$
	$posint$:		$\rightarrow$	$type$
	nat:	$type$	$\rightarrow$	$type$
	$negint$:		$\rightarrow$	$type$
	int:	$type$	$\rightarrow$	$type$
`TYPEAXIOMS`	$nat(zero)$		$=$	$zero$
	$nat(posint)$		$=$	$posint$
	$int(negint)$		$=$	$negint$
	$int(nat(\alpha))$		$=$	$nat(\alpha)$

The Knuth-Bendix procedure transforms this specification into the following set of rewrite rules:

$$\begin{array}{lcl} nat(zero) & \Rightarrow & zero \\ nat(posint) & \Rightarrow & posint \\ int(negint) & \Rightarrow & negint \\ int(nat(\alpha)) & \Rightarrow & nat(\alpha) \\ int(zero) & \Rightarrow & zero \\ int(posint) & \Rightarrow & posint \end{array}$$

All equations are oriented from left to right and two additional rewrite rules are created ("*zero* and *posint* are subtypes of the integer numbers") which corresponds to the computation of the transitive closure of the subtype relation specified in $\mathcal{T}$. This set of rewrite rules is a canoncial term rewriting system and therefore the narrowing procedure w.r.t. these rules can be used to compute $\mathcal{T}$-unifiers for two type expressions. Thus the resolution procedure can be implemented by the following two steps:

1. Transform the given type specification into a canonical term rewriting system. For this purpose the Knuth-Bendix completion procedure can be applied. It computes the transitive closure of the subtype relation.

2. The $\mathcal{T}$-unification procedure for typed terms can be implemented like the classical unification procedure with the difference that types are $\mathcal{T}$-unified by the narrowing procedure w.r.t. the rewrite rules computed in step 1 before unifiying corresponding terms.

Note that the $\mathcal{T}$-unification procedure can be simplified if the type specification does not contain subtype relations (see remarks at the end of section 4.2). If the type specification contains subtype relations, then these subtype relations influence success or failure of unification. Therefore type information at run time is not superfluous in the context of logic programming but may avoid unnecessary computations since variables can be constraint to values *and* to types by the $\mathcal{T}$-unification prodedure. Therefore typed logic programs can be executed more efficiently than their untyped equivalents [HV87]. One reason for this efficiency is the existence of a procedure which decides whether a system of type constraints has a solution. As shown above, we solve type constraints by a narrowing procedure which is based on the type equations. This is sufficient to solve type constraints in order-sorted type structures, but in a more general setting narrowing cannot decide the solvability of constraints but enumerates only a complete set of solutions. Smolka [Smo89] has shown that type constraints can be solved in his framework. Therefore the development of efficient type constraint solvers for (restricted classes of) our framework is a topic for further research.

6 Applications

We have mentioned in the introduction that a new application of our proposed framework for typed logic programming is the possibility of higher-order logic programming with polymorphic and order-sorted type structures. It is clear that our framework combines polymorphic and order-sorted type structures (take the union of the type specifications of examples 1.1 and 1.2, or example 1.3). A semantically clean amalgamation of higher-order objects with logic programming needs a higher-order logic. Miller and Nadathur [MN86] have proposed a higher-order logic programming language based on the typed lambda calculus. The operational semantics is based on resolution with a unification procedure for typed lambda expressions which is a complex and semi-decidable problem. Moreover, the proof procedure is only complete for goals which contain no type variables.

Warren [War82] has argued that no extension to Horn clause logic is necessary because the usual higher-order programming techniques can be simulated in first-order Horn clause logic. The general idea is an explicit definition of a predicate `apply` which is used for the application of an (at compile time) unknown predicate to some arguments. It is shown in [Han89c] that Warren's approach is incompatible with polymorphic type systems for logic programming like [MO84] and [Smo89]. Since we have dropped some restrictions of these type systems, we can use Warren's approach to integrate higher-order programming techniques in our framework.

Example 6.1 We give an example for the definition of a predicate `map` which applies a binary predicate to corresponding elements of two lists. To define the type of `map` we must express the type of binary predicates which are arguments to other predicates. Therefore we introduce a type constructor $pred2$ that denotes the type of binary predicates, i.e., the type specification of our example program is:

TYPEOPS	int:		$\rightarrow$	$type$
	$bool$:		$\rightarrow$	$type$
	$list$:	$type$	$\rightarrow$	$type$
	$pred2$:	$type, type$	$\rightarrow$	$type$

For each binary predicate p of type "τ_1, τ_2" we introduce a corresponding constant λp of type "$pred2(\tau_1, \tau_2)$". The relation between each predicate p and the constant λp is defined by clauses for the predicate `apply2`. Hence we get the following example program for the predicate `map` (we omit the definitions of the predicates `inc` and `bool` and the type annotations in program clauses):

func `[]`: $\rightarrow$ $list(\alpha)$
func `[..|..]`: α, $list(\alpha)$, $\rightarrow$ $list(\alpha)$
func λ`not`: $\rightarrow$ $pred2(bool, bool)$
func λ`inc`: $\rightarrow$ $pred2(int, int)$
...
pred `not`: $bool$, $bool$
pred `inc`: int, int
pred `map`: $pred2(\alpha,\beta)$, $list(\alpha)$, $list(\beta)$
pred `apply2`: $pred2(\alpha,\beta)$, α, β

vars `P:`$pred2(\alpha,\beta)$, `E1:`α, `E2:`β, `L1:`$list(\alpha)$, `L2:`$list(\beta)$, `B1,B2:`$bool$, `I1,I2:`int

```
map(P,[],[]) ←
map(P,[E1|L1],[E2|L2]) ← apply2(P,E1,E2), map(P,L1,L2)
apply2(λnot,B1,B2) ← not(B1,B2)
apply2(λinc,I1,I2) ← inc(I1,I2)
...
```

The first two clauses constitute the standard definition of the predicate `map` (cf. [SS86], p. 281), and the clauses for `apply2` relate predicate names to the corresponding binary predicates. Since the semantics of typed logic programs is based on a typed first-order logic, the predicate symbol `map` is semantically not interpreted as a higher-order predicate. The constants λ`not` and λ`inc` are also interpreted as values and not as relations. But the clauses for `apply2` ensures that in every model of the program the constants λ`not` and λ`inc` are related to the binary predicates `not` and `inc`, respectively.

This example shows the possibility to deal with higher-order objects in our typed framework. Higher-order objects are related to predicates by particular clauses for an `apply` predicate. It is also possible to permit lambda expressions which can be translated into new identifiers and `apply` clauses for these identifiers (see [War82] for more discussion). The translation was explicitly done in our examples, but this is a simple task and can be automatically done. If the underlying system implements indexing on the clauses, e.g., indexing on the first arguments of predicates (as done in most compilers for Prolog, cf. [War83]), then there is no essential loss of efficiency in our translation scheme for higher-order objects in comparison to a specific implementation of higher-order objects [War82]. More details about this method of higher-order logic programming in a polymorphically typed framework can be found in [Han89c].

7 Conclusions

We have presented a general framework for typed logic programming. It consists of a specification of a type structure and a set of well-typed Horn clauses together with type declarations for the syntactic objects occurring in the clauses. For the definition of the type structure we have used equational specifications. This allows the specification of both polymorphic and order-sorted type structures and has the advantage that there exist well-known unification procedures for a lot of equational theories. We have defined a procedure to enumerate complete sets of unifiers for typed terms with respect to a type specification which is based on a unification procedure for the equational type specification. Furthermore, we have outlined a resolution method where this unification procedure is used to unify an atom with a clause head. This framework permits polymorphic and order-sorted type structures and the possibility of higher-order programming.

The presented framework yields a new view on the rôle of types in logic programming. A type specification can be compiled into a suitable unification algorithm which is used in the resolution procedure. Therefore *different type structures imply different unification algorithms*. A many-sorted type structure does not require any type information at run time, in a polymorphic type structure a most general unifier exists for two unifiable terms and can be computed by Robinson's unification algorithm, and in order-sorted type structures there may exist several unifiers which are not comparable, but a complete set of unifiers can be computed by a procedure which is based on a unification procedure for the type theory.

Further work remains to be done. We have mentioned that the presence of types at run time is not superfluous but may reduce the search space of the resolution method. Nevertheless, there are a lot of cases where type annotations can be omitted at run time and the unification remains to be correct. For polymorphic type structures these cases are analyzed in [Han89a] and [Han89c]. New criteria for omitting type annotations must be developed in our general typed framework. Another important point is the automatic inference of types. For practical applications it is tedious to write typed program clauses since each syntactic element must be given an appropriate type. Therefore it is necessary to deduce the right types for a clause without type annotations by a type inference algorithm. This is a difficult problem in our general framework but their are successful approaches to the type inference problem for restricted classes of type structures. For instance, in the case of polymorphic type structures the type inference algorithm of ML can be used to infer the types of the variables in a clause if the types of all functions and predicates are explicitly declared [Han89a]. For a restricted class of polymorphically order-sorted type structures Smolka has found an algorithm which infers the types of variables in most cases [Smo89]. Similar solutions must be developed for particular instances of our approach.

References

[CF88] T. Conrad and U. Furbach. Sorts are Nothing but Functions. An Equational Approach to Sorts for Logic Programming. Report FKI-89-88, Techn. Univ. München, 1988.

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

[Fay79] M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.

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

[Han89a] M. Hanus. Horn Clause Programs with Polymorphic Types: Semantics and Resolution. In *Proc. of the TAPSOFT '89*, pp. 225–240. Springer LNCS 352, 1989. Extended version to appear in *Theoretical Computer Science*.

[Han89b] M. Hanus. Logic Programming with Type Specifications. Technical Report 321, FB Informatik, Univ. Dortmund, 1989.

[Han89c] M. Hanus. Polymorphic Higher-Order Programming in Prolog. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 382–397. MIT Press, 1989.

[Han90] M. Hanus. A Functional and Logic Language with Polymorphic Types. In *Proc. Int. Symposium on Design and Implementation of Symbolic Computation Systems*, pp. 215–224. Springer LNCS 429, 1990.

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

[KB70] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon Press, 1970.

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

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

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

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

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

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

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

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. Dissertation, FB Informatik, Univ. Kaiserslautern, 1989.

[SNGM87] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-Sorted Equational Computation. SEKI Report SR-87-14, FB Informatik, Univ. Kaiserslautern, 1987.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Wal89] U. Waldmann. Unification in Order-Sorted Signatures. Technical Report 298, FB Informatik, Univ. Dortmund, 1989.

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

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

[XW88] J. Xu and D.S. Warren. A Type Inference System For Prolog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 604–619, 1988.