In *Proc. TAPSOFT'91, Brighton, April 1991*, Springer LNCS 494, pp. 181–200

# Parametric Order-Sorted Types in Logic Programming

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

This paper proposes a type system for logic programming where types are structured in two ways. Firstly, functions and predicates may be declared with types containing type parameters which are universally quantified over all types. In this case each instance of the type declaration can be used in the logic program. Secondly, types are related by subset inclusions. In this case a function or predicate can be applied to all subtypes of its declared type. While previous proposals for such type systems have strong restrictions on the subtype relation, we assume that the subtype order is specified by Horn clauses for the subtype relation $\leq$. This allows the declaration of a lot of interesting type structures, e.g., type constructors which are monotonic as well as anti-monotonic in their arguments. For instance, parametric order-sorted type structures for logic programs with higher-order predicates can be specified in our framework.

This paper presents the declarative and operational semantics of the typed logic language. The operational semantics requires a unification procedure on well-typed terms. This unification procedure is described by a set of transformation rules which generate a set of type constraints from a given unification problem. The solvability of these type constraints is decidable for particular type structures.

## 1 Introduction

Types are important for programming languages because typed programs are easier to read and a lot of programming errors can be detected at compile time. Moreover, compilers can generate more efficient code if type information is available. Therefore various attempts have been made to integrate types into the classically untyped world of logic programming. These proposals can be divided into two groups. In the *inference-based approaches* [Mis84] [Zob87] [XW88] [BG89] (among others) types are not part of the program but are considered as logical consequences of the program. These approaches consider the type of a predicate as a superset of the success set of the predicate which is computed by abstract interpretation techniques. If the type of a predicate is empty, then this predicate cannot succeed and hence it is interpreted as a programming error. But in many cases the inference of types from a completely untyped logic program does not yield sufficient results because an untyped logic program does not contain the type information expected by the programmer [Nai87].

The *declaration-based approaches* try to overcome this problem by permitting the addition of type declarations to the logic program. A type checker compares the application of a function or predicate in a program clause with the programmer's declaration and reports an error if it is used in a wrong context. In these approaches the type declarations are a part of the program's semantics and may influence the execution of the program. For instance, the polymorphic type system of Mycroft and O'Keefe [MO84] is motivated from ML and the type information is only used for compile-time checks, i.e., the types are not visible at run time (but to ensure this property more restrictions on the programs are necessary than described in their paper). An extension of their type system [Han89a] allows the application of higher-order programming techniques [Han89b] but needs type information at run time to ensure that "well-typed programs do not go wrong". This is also true for order-sorted type systems where types may be related by an inclusion relation [SNGM89]. But it has been shown that type information at run time is not superfluous but may avoid unnecessary computations and reduce the search space [SS85] [HV87].

Smolka [Smo89] and Hill and Topor [HT90] have proposed declarative type systems for logic programming which integrate parametric and order-sorted polymorphism. Both approaches have several restrictions on the combination of parametric and order-sorted types. For instance, they require that type constructors like *list* and *pair* must be monotonic in their arguments, i.e., $list(\tau_1)$ is a subtype of $list(\tau_2)$ if $\tau_1$ is a subtype of $\tau_2$. But this restriction is a severe limitation if we want to use higher-order programming techniques: Warren [War82] has shown how to simulate higher-order programming in first-order logic, and the adaptation of this technique to a polymorphically typed logic language is shown in [Han89b]. In this case there is a type constructor $pred1$ denoting the type of unary predicates. But $pred1$ is not monotonic: $pred1(int)$ is a subtype of $pred1(nat)$ because all unary predicates defined on integers can be used if a unary predicate defined on naturals is required provided that *nat* is a subtype of *int* (cf. [CW85]).

In order to solve this problem, we present a generalized declaration-based type system. We allow the specification of a subtype order by arbitrary Horn clauses for the subtype relation $\leq$. Hence the user can declare type constructors which are monotonic or anti-monotonic in their arguments. Figure 1 shows an example of a typed logic program in our framework. It contains the basic types *zero*, *posint* and *nat*, where *zero* and *posint* are subtypes of *nat*, and the type constructors *list* and $pred1$ of arity 1 which are monotonic and anti-monotonic in their arguments, respectively. The subtype order specified by this program is the least quasi-ordering on the type expressions which satisfies all subtype axioms. Another example showing the application of higher-order programming techniques will be presented in section 7.

Similarly to order-sorted logic, the specified type structure influences the operational semantics of the program, i.e., the unification procedure must consider the types of the terms to be unified. For instance, if we have to prove the literal

```
?- plus(X,Y,Z)
```

and we know that variable `X` has type *posint*, i.e., it is only allowed to bind `X` to positive numbers (because of its usage in another literal), then the first clause for `plus` must not be applied to this literal since `0` has type *zero* which is incompatible with *posint*. Thus a typed unification may avoid unnecessary computations. The main result of this paper is a unification procedure which takes two well-typed literals as input and produces a solvable set of type constraints iff the literals are unifiable. The solvability of these type constraints is decidable for particular type structures. This unification procedure can be used for a sound and complete resolution procedure for typed logic programs.

This paper is organized as follows. The next two sections define the syntax and the declarative

```
type      zero, posint, nat, list/1, pred1/1
subtype   zero ≤ nat
          posint ≤ nat
          α ≤ β  ⇒  list(α) ≤ list(β)
          β ≤ α  ⇒  pred1(α) ≤ pred1(β)

func      0:  →  zero
func      s:  nat  →  posint
func      []      :            →  list(α)
func      [..|..]: α,  list(α)  →  list(α)

pred      plus   :  nat,  nat,  nat
pred      member:  α,  list(α)

plus(0,N,N) ←
plus(s(N1),N2,s(N3)) ← plus(N1,N2,N3)

member(E,[E|L]) ←
member(E,[F|L]) ← member(E,L)
```

Figure 1: A logic program with parametric and order-sorted types

semantics of typed logic programs. Since our approach is declaration-based, types are not only sets of terms but they are present in all interpretations of the program similarly to [Smo89] or [HT90]. Section 4 defines the typed Horn clause calculus which is a sound and complete method to prove valid atoms. Section 5 presents the unification procedure on typed terms which can be used for the resolution method for typed logic programs presented in section 6. A new application of our framework is presented in section 7. The proofs of the theorems are omitted from this paper. They can be found in [Han91].

## 2 The typed logic language

For the definition of types we assume familiarity with basic notions from algebraic specifications as to be found in [EM85]. A **type signature** is a single-sorted signature $\mathcal{H}$. Constants in $\mathcal{H}$ are called **basic types** and $n$-ary functions in $\mathcal{H}$ are called **type constructors** of arity $n$. For instance, the first line in figure 1 specifies a type signature with *zero*, *posint*, *nat* as basic types and two type constructors *list* and *pred*1 or arity 1.

By $\mathcal{X}$ we denote an infinite set of **type parameters**[1]. $T_{\mathcal{H}}(\mathcal{X})$ denotes the term algebra over $\mathcal{H}$ and $\mathcal{X}$, i.e., the set of all well-formed type expressions with type parameters from $\mathcal{X}$.

A **type substitution** $\sigma$ is an $\mathcal{H}$-homomorphism $\sigma\colon T_{\mathcal{H}}(\mathcal{X}) \to T_{\mathcal{H}}(\mathcal{X})$ where $\sigma(\alpha) \neq \alpha$ only for finitely many $\alpha \in \mathcal{X}$. A type substitution replaces type parameters by other type expressions. We use the following notation for the class of all type substitutions:

$$\boldsymbol{TS(\mathcal{H},\mathcal{X})} := \{\sigma\colon T_{\mathcal{H}}(\mathcal{X}) \to T_{\mathcal{H}}(\mathcal{X}) \mid \sigma \text{ is a type substitution}\}$$

[1]In order to avoid confusion with variables occurring in clauses of logic programs, we use the notion "type parameters" for variables which are quantified over types.

Inclusion relations between types are specified by Horn clauses for the binary predicate $\leq$. A **subtype declaration** is a formula

$$\tau_1 \leq \tau_1', \ldots, \tau_n \leq \tau_n' \;\Rightarrow\; \tau_0 \leq \tau_0'$$

where $\tau_i, \tau_i'$ are type expressions from $T_{\mathcal{H}}(\mathcal{X})$ $(i = 0, \ldots, n)$. If $\mathcal{S}$ is a set of subtype declarations, $\leq_{\mathcal{S}}$ denotes the least quasi-ordering generated by $\mathcal{S}$, i.e., $\tau \leq_{\mathcal{S}} \tau'$ is true iff $\tau \leq \tau'$ is a logical consequence of the Horn clause program

$$\mathcal{S} \cup \{(\Rightarrow \alpha \leq \alpha), (\alpha \leq \beta, \beta \leq \gamma \Rightarrow \alpha \leq \gamma)\}$$

where $\alpha$, $\beta$ and $\gamma$ are different type parameters from $\mathcal{X}$.

A **type specification** is a pair $(\mathcal{H}, \mathcal{S})$ where $\mathcal{H}$ is a type signature and $\mathcal{S}$ is a set of subtype declarations. For instance, the parts preceded by the keywords "type" and "subtype" in figure 1 are a type specification.

$\boldsymbol{\Sigma} = (\boldsymbol{\mathcal{H}}, \boldsymbol{\mathcal{S}}, \boldsymbol{Func}, \boldsymbol{Pred})$ is called a **polymorphic signature** for logic programs if

- $(\mathcal{H}, \mathcal{S})$ is a type specification with $T_{\mathcal{H}}(\emptyset) \neq \emptyset$
- $Func$ is a set of **function declarations** of the form $f{:}\tau_1, \ldots, \tau_n \to \tau$ with $\tau_i, \tau \in T_{\mathcal{H}}(\mathcal{X})$ $(n \geq 0)$
- $Pred$ is a set of **predicate declarations** of the form $p{:}\tau_1, \ldots, \tau_n$ with $\tau_i \in T_{\mathcal{H}}(\mathcal{X})$ $(n \geq 0)$

In addition, we assume that $\Sigma$ contains at most one type declaration for each function and predicate symbol, i.e., we exclude overloading similarly to [Smo89] and [HT90]. However, this restriction does not imply that a function or predicate can only be applied to arguments of a fixed type: if the declared type contains type parameters, then each instance of this type (replacement of type parameters by other type expressions) is a valid type for the function or predicate (*parametric polymorphism*), and if some argument types have subtypes, then the function or predicate can also be applied to these subtypes (*inclusion polymorphism*).

In the following we fix a set $\mathcal{X}$ of type parameters and a polymorphic signature $\Sigma = (\mathcal{H}, \mathcal{S}, Func, Pred)$. Let $\boldsymbol{Var}$ be a set of variable names different from symbols in $\Sigma$ and $\mathcal{X}$. A set $V$ with elements of the form $x{:}\tau$ where $x \in Var$ and $\tau \in T_{\mathcal{H}}(\mathcal{X})$ is called a **set of typed variables** if $\tau = \tau'$ whenever $x{:}\tau, x{:}\tau' \in V$. If $\sigma \in TS(\mathcal{H}, \mathcal{X})$ is a type substitution and $V$ a set of typed variables, then the application of $\sigma$ to $V$ yields a new set of typed variables defined by

$$\sigma(V) := \{x{:}\sigma(\tau) \mid x{:}\tau \in V\}$$

The set $Term_\tau(V)$ of **terms of type $\tau$ with variables from $V$** is the least set satisfying the following conditions:

- $x \in Term_\tau(V)$ if $x{:}\tau_x \in V$ and $\tau_x \leq_{\mathcal{S}} \tau$
- $f(t_1, \ldots, t_n) \in Term_\tau(V)$ if $f{:}\tau_1, \ldots, \tau_n \to \tau_0 \in Func$ $(n \geq 0)$, $\sigma \in TS(\mathcal{H}, \mathcal{X})$, $t_i \in Term_{\sigma(\tau_i)}(V)$ $(i = 1, \ldots, n)$ and $\sigma(\tau_0) \leq_{\mathcal{S}} \tau$

***Term*$(V)$** denotes the set of all (well-typed) terms with variables from $V$, i.e., $Term(V) := \bigcup_{\tau \in T_{\mathcal{H}}(\mathcal{X})} Term_\tau(V)$. Elements of $Term(V)$ are also called **$(\Sigma, V)$-terms**.

The definition of the other syntactic elements of typed logic programs is straightforward: A $(\Sigma, V)$-**atom** has the form $p(t_1, \ldots, t_n)$ where $p{:}\tau_1, \ldots, \tau_n \in Pred$, $\sigma \in TS(\mathcal{H}, \mathcal{X})$ and $t_i \in Term_{\sigma(\tau_i)}(V)$ $(i = 1, \ldots, n)$. A $(\Sigma, V)$-**goal** is a finite set of $(\Sigma, V)$-atoms. A $(\Sigma, V)$-**clause** is a pair $P \leftarrow G$ where the **head** $P$ is a $(\Sigma, V)$-atom and the **body** $G$ is a $(\Sigma, V)$-goal. A **$\Sigma$-term** (atom, goal, clause) is a $(\Sigma, V)$-term (atom, goal, clause) for some set of typed variables $V$. If $s$ is a term, atom, goal etc., then ***var*$(s)$** denotes the set of all typed variables occurring in $s$.

A **typed logic program $(\Sigma, \mathcal{P})$** is a polymorphic signature $\Sigma$ together with a set of $\Sigma$-clauses $\mathcal{P}$. Figure 1 contains an example of a typed logic program where the variables have the following types:

$$V = \{\texttt{N}{:}nat,\ \texttt{N1}{:}nat,\ \texttt{N2}{:}nat,\ \texttt{N3}{:}nat,\ \texttt{E}{:}\alpha,\ \texttt{F}{:}\alpha,\ \texttt{L}{:}list(\alpha)\}$$

## 3 Declarative semantics

Similarly to [Poi86] and [Han89a], we use a two-level approach for the declarative semantics of typed logic programs. The first level interprets the type specification $(\mathcal{H}, \mathcal{S})$ by a $\mathcal{H}$-algebra and a quasi-ordering satisfying $\leq_{\mathcal{S}}$. Type parameters vary over all elements of this $\mathcal{H}$-algebra. From such an interpretation and the given polymorphic signature we derive a dependent order-sorted signature which will be interpreted as usual [SNGM89]. Hence models for typed logic programs consists of two parts: a model for the specified type structure and a model for the derived order-sorted logic program. In the following we present the detailed definitions.

A $(\mathcal{H}, \mathcal{S})$-**type structure** $\mathcal{A}$ (interpretation of a type specification $(\mathcal{H}, \mathcal{S})$) consists of a set of sort symbols $S_{\mathcal{A}}$, a mapping $\mathcal{A}_k{:}\ (S_{\mathcal{A}})^n \rightarrow S_{\mathcal{A}}$ for each $n$-ary function symbol $k$ in $\mathcal{H}$ $(n \geq 0)$ and a quasi-ordering $\leq_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ satisfying all axioms from $\mathcal{S}$. If $\Sigma = (\mathcal{H}, \mathcal{S}, Func, Pred)$ is a polymorphic signature, then a $(\mathcal{H}, \mathcal{S})$-type structure $\mathcal{A}$ determines the following sets of function and predicate types:

$$\begin{array}{rcl} Func_{\mathcal{A}} & := & \{f{:}\sigma(\tau_f) \mid f{:}\tau_f \in Func, \sigma{:}\ \mathcal{X} \rightarrow S_{\mathcal{A}} \text{ is a type parameter assignment}\} \\ Pred_{\mathcal{A}} & := & \{p{:}\sigma(\tau_p) \mid p{:}\tau_p \in Pred, \sigma{:}\ \mathcal{X} \rightarrow S_{\mathcal{A}} \text{ is a type parameter assignment}\} \end{array}$$

(where $\sigma(\tau_f)$ and $\sigma(\tau_p)$ denotes the componentwise application of $\sigma$ to $\tau_f$ and $\tau_p$, respectively). A $(\mathcal{H}, \mathcal{S})$-type structure $\mathcal{A}$ can be extended to a $\Sigma$-interpretation by interpreting the order-sorted signature $(S_{\mathcal{A}}, \leq_{\mathcal{A}}, Func_{\mathcal{A}}, Pred_{\mathcal{A}})$ as usual [Smo86] [SNGM89]: A $\Sigma$-**interpretation** $\mathcal{A}$ consists of a $(\mathcal{H}, \mathcal{S})$-type structure, a family of sets $\{\mathcal{A}_\tau \mid \tau \in S_{\mathcal{A}}\}$, a mapping $\mathcal{A}_f{:}\ D_f^{\mathcal{A}} \rightarrow C_{\mathcal{A}}$ for each function symbol $f$ in $\Sigma$ and a relation $\mathcal{A}_p \subseteq D_p^{\mathcal{A}}$ for each predicate symbol $p$ in $\Sigma$, where the following conditions hold:

- $C_{\mathcal{A}} := \bigcup_{\tau \in S_{\mathcal{A}}} \mathcal{A}_\tau$ is called the **carrier** of $\mathcal{A}$
- $\mathcal{A}_\tau \subseteq \mathcal{A}_{\tau'}$ if $\tau \leq_{\mathcal{A}} \tau'$
- $D_f^{\mathcal{A}} \subseteq (C_{\mathcal{A}})^n$ if $f$ has arity $n$
- $D_p^{\mathcal{A}} \subseteq (C_{\mathcal{A}})^n$ if $p$ has arity $n$

- If $f{:}\tau_1,\ldots,\tau_n \to \tau \in Func_{\mathcal{A}}$, then $\mathcal{A}_{\tau_1}\times\cdots\times\mathcal{A}_{\tau_n} \subseteq D_f^{\mathcal{A}}$ and $\mathcal{A}_f(\mathcal{A}_{\tau_1}\times\cdots\times\mathcal{A}_{\tau_n}) \subseteq \mathcal{A}_\tau$
- If $p{:}\tau_1,\ldots,\tau_n \in Pred_{\mathcal{A}}$, then $\mathcal{A}_{\tau_1}\times\cdots\times\mathcal{A}_{\tau_n} \subseteq D_p^{\mathcal{A}}$

In order to compare different interpretations, we define homomorphisms between them. If $\mathcal{A}$ and $\mathcal{B}$ are two $\Sigma$-interpretations, a **$\Sigma$-homomorphism** $h$ from $\mathcal{A}$ into $\mathcal{B}$ is a mapping $h{:}\, S_{\mathcal{A}}\cup C_{\mathcal{A}} \to S_{\mathcal{B}}\cup C_{\mathcal{B}}$ with

- $h(S_{\mathcal{A}}) \subseteq S_{\mathcal{B}}$ and $h(C_{\mathcal{A}}) \subseteq C_{\mathcal{B}}$
- $h(\mathcal{A}_k(\tau_1,\ldots,\tau_n)) = \mathcal{B}_k(h(\tau_1),\ldots,h(\tau_n))$ for all $n$-ary type constructors $k$ and all $\tau_1,\ldots,\tau_n \in S_{\mathcal{A}}$
- $h(\mathcal{A}_\tau) \subseteq \mathcal{B}_{h(\tau)}$ for all $\tau \in S_{\mathcal{A}}$
- $h(D_f^{\mathcal{A}}) \subseteq D_f^{\mathcal{B}}$ and $h(\mathcal{A}_f(a_1,\ldots,a_n)) = \mathcal{B}_f(h(a_1),\ldots,h(a_n))$ for all $(a_1,\ldots,a_n) \in D_f^{\mathcal{A}}$
- $h(D_p^{\mathcal{A}}) \subseteq D_p^{\mathcal{B}}$ and $(h(a_1),\ldots,h(a_n)) \in \mathcal{B}_p$ for all $(a_1,\ldots,a_n) \in \mathcal{A}_p$

Note that if $\mathcal{A}$ and $\mathcal{B}$ have identical type structures and $h$ is the identity on $S_{\mathcal{A}}$, then $h$ is an order-sorted homomorphism in the sense of [Smo86] and [SNGM89]. It is easy to prove that the class of all $\Sigma$-interpretations together with the $\Sigma$-homomorphisms is a category.

A homomorphism in our typed framework consists of a mapping between type structures and a mapping between appropriate order-sorted structures. Consequently, a variable assignment in the typed framework maps type parameters into types and typed variables into objects of appropriate types: If $\mathcal{A}$ is a $\Sigma$-interpretation, then an **assignment** for $(\mathcal{X}, V)$ in $\mathcal{A}$ is a mapping $\delta{:}\, \mathcal{X}\cup V \to S_{\mathcal{A}}\cup C_{\mathcal{A}}$ where $\delta(\alpha) \in S_{\mathcal{A}}$ for all type parameters $\alpha \in \mathcal{X}$ and $\delta(x) \in \mathcal{A}_{\hat{\delta}(\tau)}$ for all $x{:}\tau \in V$ ($\hat{\delta}$ denotes the extension of $\delta$ to $T_{\mathcal{H}}(\mathcal{X})$ which uniquely exists [EM85]).

$\mathcal{T}$ is called the **free term interpretation over $\mathcal{X}$ and $V$** if the following conditions hold:

1. $S_{\mathcal{T}} = T_{\mathcal{H}}(\mathcal{X})$, $\mathcal{T}_k(\tau_1,\ldots,\tau_n) = k(\tau_1,\ldots,\tau_n)$ for all $n$-ary type constructors $k$ and all type expressions $\tau_1,\ldots,\tau_n \in T_{\mathcal{H}}(\mathcal{X})$, and $\leq_{\mathcal{T}} = \leq_{\mathcal{S}}$, i.e., the type structure of $\mathcal{T}$ is the initial term model (least Herbrand model) of the type specification $(\mathcal{H}, \mathcal{S})$
2. $\mathcal{T}_\tau := Term_\tau(V)$ for all $\tau \in T_{\mathcal{H}}(\mathcal{X})$, i.e., the carrier of $\mathcal{T}$ is the set of all well-typed terms with variables from $V$
3. $D_f^{\mathcal{T}} := \bigcup_{f:\tau_1,\ldots,\tau_n\to\tau\in Func_{\mathcal{T}}} \mathcal{T}_{\tau_1}\times\cdots\mathcal{T}_{\tau_n}$
4. $\mathcal{T}_f(t_1,\ldots,t_n) := f(t_1,\ldots,t_n)$ for all $n$-ary function symbols $f$ and $(t_1,\ldots,t_n) \in D_f^{\mathcal{T}}$
5. $D_p^{\mathcal{T}} := \bigcup_{p:\tau_1,\ldots,\tau_n\in Pred_{\mathcal{T}}} \mathcal{T}_{\tau_1}\times\cdots\mathcal{T}_{\tau_n}$
6. $\mathcal{T}_p := \emptyset$ for all $n$-ary predicate symbols $p$

It is easy to show that $\mathcal{T}$ is a $\Sigma$-interpretation. We denote this $\Sigma$-interpretation by $\boldsymbol{T_\Sigma(\mathcal{X}, V)}$.

**Lemma 3.1 (Free term interpretation)** *Let $\mathcal{A}$ be a $\Sigma$-interpretation and $\delta$ be an assignment for $(\mathcal{X}, V)$ in $\mathcal{A}$. There exists a unique $\Sigma$-homomorphism $h$ from $T_\Sigma(\mathcal{X}, V)$ into $\mathcal{A}$ with $h(\alpha) = \delta(\alpha)$ for all $\alpha \in \mathcal{X}$ and $h(x) = \delta(x)$ for all $x{:}\tau \in V$.*

The lemma shows that any variable assignment $\delta$ can be extended to a $\Sigma$-homomorphism in a unique way. In the following we denote this $\Sigma$-homomorphism again by $\delta$.

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 validity of atoms, goals and clauses relative to a given $\Sigma$-interpretation $\mathcal{A}$:

- Let $\delta$ be an assignment for $(\mathcal{X}, V)$ in $\mathcal{A}$.

  $\boldsymbol{\mathcal{A}, \delta} \models \boldsymbol{L}$ if $L = p(t_1, \ldots, t_n)$ is a $(\Sigma, V)$-atom with $(\delta(t_1), \ldots, \delta(t_n)) \in \mathcal{A}_p$

  $\boldsymbol{\mathcal{A}, \delta} \models \boldsymbol{G}$ if $G$ is a $(\Sigma, V)$-goal with $\mathcal{A}, \delta \models L$ for all $L \in G$

  $\boldsymbol{\mathcal{A}, \delta} \models \boldsymbol{L \leftarrow G}$ if $L \leftarrow G$ is a $(\Sigma, V)$-clause where $\mathcal{A}, \delta \models G$ implies $\mathcal{A}, \delta \models L$

- $\boldsymbol{\mathcal{A}, V} \models \boldsymbol{\mathcal{F}}$ if $\mathcal{F}$ is a $(\Sigma, V)$-atom, -goal or -clause with $\mathcal{A}, \delta \models \mathcal{F}$ for all assignments $\delta$ for $(\mathcal{X}, V)$ in $\mathcal{A}$

We say "$L$ is **valid in** $\mathcal{A}$" if $\mathcal{A}$ is a $\Sigma$-interpretation with $\mathcal{A}, var(L) \models L$ (analogously for goals and clauses). A $\Sigma$-interpretation $\mathcal{A}$ is called **model** for a typed logic program $(\Sigma, \mathcal{P})$ if all clauses from $\mathcal{P}$ are valid in $\mathcal{A}$. A $(\Sigma, V)$-goal $G$ is called **valid in** $(\Sigma, \mathcal{P})$ relative to $V$ if $\mathcal{A}, V \models G$ for every model $\mathcal{A}$ of $(\Sigma, \mathcal{P})$. We shall write: $(\boldsymbol{\Sigma}, \boldsymbol{\mathcal{P}}, \boldsymbol{V}) \models \boldsymbol{G}$. Validity of atoms and clauses in $(\Sigma, \mathcal{P})$ is analogously defined.

This notion of validity extends 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 parameters 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.

Furthermore, note that due to our two-level semantics $\Sigma$-interpretations may contain more types than specified in $\Sigma$. For instance, if the typed logic program $(\Sigma, \mathcal{P})$ contains only one type *int*, the predicate declaration `p`:$\alpha$ and the $(\Sigma, \{\texttt{i}{:}int\})$-clause `p(i)` $\leftarrow$, then $(\Sigma, \mathcal{P}, \{\texttt{x}{:}\alpha\}) \models$ `p(x)` does not hold. But the $(\Sigma, \{\texttt{x}{:}\alpha\})$-atom `p(x)` is valid in the initial model of $(\Sigma, \mathcal{P})$. This is similarly to untyped logic programming where $\forall x p(x)$ is true in the least Herbrand model of the program $\{p(a) \leftarrow\}$ but $\forall x p(x)$ is not a logical consequence of $\{p(a) \leftarrow\}$.

Let $V, V'$ be sets of typed variables. A **typed substitution** $\sigma$ is a $\Sigma$-homomorphism $\sigma$ from $T_\Sigma(\mathcal{X}, V)$ into $T_\Sigma(\mathcal{X}, V')$ where $\sigma(\alpha) \neq \alpha$ and $\sigma(x) \neq x$ only for finitely many $\alpha \in \mathcal{X}$ and $x{:}\tau \in V$. Therefore a typed substitution is a combination of a substitution on type expressions and a substitution which replaces typed variables by well-typed terms. A typed substitution keeps the set of type parameters $\mathcal{X}$ but may change the set of typed variables because the types of the variables influence validity. We extend typed substitutions on $\Sigma$-atoms by:

$$\sigma(p(t_1, \ldots, t_n)) = p(\sigma(t_1), \ldots, \sigma(t_n))$$

Furthermore we define:

$\boldsymbol{Sub}_\Sigma(\boldsymbol{\mathcal{X}}, \boldsymbol{V}, \boldsymbol{V'}) := \{\sigma \mid \sigma$ is a typed substitution from $T_\Sigma(\mathcal{X}, V)$ into $T_\Sigma(\mathcal{X}, V')\}$

$\boldsymbol{\sigma}_1 =_V \boldsymbol{\sigma}_2$ if $\sigma_1 \in Sub_\Sigma(\mathcal{X}, V_1, V')$, $\sigma_2 \in Sub_\Sigma(\mathcal{X}, V_2, V')$ with $V \subseteq V_1 \cap V_2$ and $\sigma_1(x) = \sigma_2(x)$ for all $x{:}\tau \in V$ and $\sigma_1(\alpha) = \sigma_2(\alpha)$ for all type parameters $\alpha$ occurring in $V$

By lemma 3.1, typed substitutions are determined by their behaviour on type parameters and typed variables. Therefore we represent a typed substitution $\sigma$ by the following set:

$$\{\alpha/\sigma(\alpha) \mid \sigma(\alpha) \neq \alpha,\ \alpha \in \mathcal{X}\} \cup \{x/\sigma(x) \mid \sigma(x) \neq x,\ x{:}\tau \in V\}$$

For instance, the application of the typed substitution

$$\sigma \ = \ \{\alpha/nat,\ \texttt{E}/0\}$$

to the $(\Sigma, \{\texttt{E}{:}\alpha, \texttt{L}{:}list(\alpha)\})$-atom `member(E,L)` yields the $(\Sigma, \{\texttt{L}{:}list(nat)\})$-atom `member(0,L)`.

## 4 The typed Horn clause calculus

This section presents an inference system for proving validity in typed logic programs. In contrast to the untyped Horn clause calculus it is necessary to collect all variables used in a derivation since validity depends on the types of variables. Let $(\Sigma, \mathcal{P})$ be a typed logic program. The **typed Horn clause calculus** consists of the inference rules in figure 2. We write $(\boldsymbol{\Sigma}, \boldsymbol{\mathcal{P}}, \boldsymbol{V}) \vdash \boldsymbol{L}$ if $(\Sigma, \mathcal{P}, V) \vdash L \leftarrow \emptyset$ can be deduced by these inference rules. The following theorem states soundness and completeness of the typed Horn clause calculus:

**Theorem 4.1** *Let $(\Sigma, \mathcal{P})$ be a typed logic program, $V$ be a set of typed variables and $L$ be a $(\Sigma, V)$-atom. Then:* $\quad (\Sigma, \mathcal{P}, V) \vdash L \quad \Longleftrightarrow \quad (\Sigma, \mathcal{P}, V) \models L$

*Axioms*: $$\frac{}{(\Sigma, \mathcal{P}, V) \vdash L \leftarrow G} \qquad \text{if } L \leftarrow G \in \mathcal{P} \text{ is a } (\Sigma, V)\text{-clause}$$

*Substitution rule*: $$\frac{(\Sigma, \mathcal{P}, V) \vdash L \leftarrow G}{(\Sigma, \mathcal{P}, V') \vdash \sigma(L) \leftarrow \sigma(G)} \qquad \text{if } \sigma \in Sub_\Sigma(\mathcal{X}, V, V')$$

*Cut rule*: $$\frac{(\Sigma, \mathcal{P}, V) \vdash L \leftarrow G \cup \{L'\},\ (\Sigma, \mathcal{P}, V) \vdash L' \leftarrow G'}{(\Sigma, \mathcal{P}, V) \vdash L \leftarrow G \cup G'}$$

Figure 2: The typed Horn clause calculus

## 5 Typed unification

The SLD-resolution procedure [AvE82] is an efficient method to prove validity of goals and therefore it is used as the operational semantics of programming languages based on Horn clause logic. The basic operation in a resolution step is the computation of a unifier for two atoms, i.e., a substitution which makes the atoms identical. Unfortunately, the classical unification procedure [Rob65] cannot be applied in our typed framework because the computed substitutions may be ill-typed.

**Example 5.1** Consider the type structure defined in figure 1 and the two atoms

$$\texttt{plus(0,N,N)} \qquad\qquad \texttt{plus(X,Y,Z)}$$

w.r.t. the typed variables $\{\texttt{N}{:}nat, \texttt{X}{:}posint, \texttt{Y}{:}posint, \texttt{Z}{:}posint\}$. The substitution computed by the classical (untyped) unification procedure would bind variable `X` to `0`. But this is not a typed substitution because a variable which is constrained to be a positive integer must not be bound to a term of type *zero*. In this example there is no typed substitution which makes the atoms identical and therefore the unification procedure should fail.

From a practical point of view it is important that the unification procedure may fail because of incompatible types since in this case the search space can be reduced. Thus the integration of types into the computation process yields a more efficient program execution because variables can be constrained to types *and* to values in a typed unification procedure [SS85] [HV87].

In this section we will present a unification procedure for our typed logic. The unification procedure takes two well-typed atoms or terms as input and computes a solvable set of type constraints (subtype relations) iff the atoms or terms are unifiable. In order to use the improved computational power of typed logic programs (reduction of the search tree), it is necessary to decide the solvability of a set of type constraints. Depending on the type structure, such decision procedures may not exist. But there exist decision procedures for restricted and interesting classes of type structures which can be used in our typed framework.

For a practical unification algorithm it is essential that the unifiability of two variables can be decided only by their types. We want to avoid situations where two terms have incompatible types but may have instances which are identical. Therefore we will require that no term has two types which are incompatible. Formally, we call $\Sigma$ a polymorphic signature **with least types** if there exists a type $\tau_0$ with $\tau_0 \leq_S \tau$, $\tau_0 \leq_S \tau'$ and $t \in Term_{\tau_0}(V)$ whenever $t \in Term_{\tau}(V) \cap Term_{\tau'}(V)$. Our typed unification algorithm is only complete for polymorphic signatures with least types. We will discuss this requirement later.

We describe the typed unification by a set of transformation rules which generate a set of type constraints from a set of equations between well-typed terms. In the following we denote by $E$ or $E'$ an **equation system** w.r.t. $V$ which is a finite multiset of elements of the form

$$t{:}\tau \doteq t'{:}\tau' \quad \text{or} \quad x{:}\tau \doteq t$$

where $x, t, t'$ are $(\Sigma, V)$-terms, $x$ is a variable and $\tau, \tau'$ are type expressions. By $C$ or $C'$ we denote a **type constraint system** w.r.t. $V$ which is a finite multiset of elements of the form

$$\tau \leq \tau' \quad \text{or} \quad t{:}\tau$$

where $\tau, \tau'$ are type expressions and $t$ is a $(\Sigma, V)$-term. We omit $V$ if it is clear from the context.

We call a typed substitution $\sigma \in Sub_{\Sigma}(\mathcal{X}, V, V')$ a **solution** of an equation system $E$ and a type constraint system $C$ w.r.t. $V$ if it is a solution of each element in $E$ and $C$, where $\sigma$ is a solution of

- $\boxed{t{:}\tau \doteq t'{:}\tau'}$ if $\sigma(t) = \sigma(t')$ and $\sigma(t) \in Term_{\sigma(\tau)}(V') \cap Term_{\sigma(\tau')}(V')$,
- $\boxed{x{:}\tau \doteq t}$ if $\sigma(x) = \sigma(t)$ and $\sigma(x) \in Term_{\sigma(\tau)}(V')$,

**Unification of types**

$C; \{x{:}\tau_x \doteq t{:}\tau\} \cup E \quad \xrightarrow{tu} \quad C \cup \{\alpha \leq \tau_x, \alpha \leq \tau\}; \{x{:}\alpha \doteq t{:}\alpha\} \cup E$
if $\alpha$ is a new type parameter and $\tau_x \neq \tau$

**Decomposition of equations**

$C; \{f(t_1, \ldots, t_n){:}\tau \doteq f(t'_1, \ldots, t'_n){:}\tau'\} \cup E \quad \xrightarrow{tu} \quad C \cup \{\tau_0 \leq \tau, \tau_0 \leq \tau'\}; \{t_i{:}\tau_i \doteq t'_i{:}\tau_i\}_{i=1,\ldots,n} \cup E$
if $f{:}\tau_1, \ldots, \tau_n \to \tau_0$ is a new variant of the type declaration for $f$ in $\Sigma$

**Isolation of variables**

$C; \{x{:}\tau \doteq t{:}\tau\} \cup E \quad \xrightarrow{tu} \quad C; \{x{:}\tau \doteq t{:}\tau\} \cup \{x/t\}(E)$
if $x$ occurs in $E$ but not in $t$

**Commutation of variable equations**

$C; \{t{:}\tau \doteq x{:}\tau'\} \cup E \quad \xrightarrow{tu} \quad C; \{x{:}\tau' \doteq t{:}\tau\} \cup E$
if $t$ is not a variable

**Deletion of equations**

$C; \{x{:}\tau \doteq x{:}\tau\} \cup E \quad \xrightarrow{tu} \quad C \cup \{x{:}\tau\}; E$

Figure 3: Transformation rules for typed unification

- $\boxed{\tau \leq \tau'}$ if $\sigma(\tau) \leq_{\mathcal{S}} \sigma(\tau')$,
- $\boxed{t{:}\tau}$ if $\sigma(t) \in Term_{\sigma(\tau)}(V')$.

We call the pair $C; E$ **solvable** if there is a solution of $C; E$.

Initially, $E$ contains only equations of the form $t{:}\tau \doteq t'{:}\tau'$ and $C$ contains the type constraints for the variables in $V$ (e.g., if we want to unify two terms $t, t' \in Term(V)$, then $C = V$ and $E = \{t{:}\alpha \doteq t'{:}\beta\}$ where $\alpha$ and $\beta$ are new type parameters). First we transform the pair $C; E$ by the rules in figure 3. In the first rule for typed unification a new type parameter is generated which represents the common subtype of $\tau_x$ and $\tau$. In order to relate solutions of the original type constraint and equation system with solutions of the transformed one, we need the notion of the "extension" of a typed substitution. Let $\sigma, \sigma' \in Sub_\Sigma(\mathcal{X}, V, V')$ be typed substitutions. If the only difference between $\sigma$ and $\sigma'$ is the behaviour on some type parameters $\alpha$ where $\sigma(\alpha) = \alpha$, then $\sigma'$ is called **extension** of $\sigma$.

**Example 5.2** Consider the type structure defined in figure 1 and the type constraint and equation system

$$\{\mathtt{N}{:}nat\} \ ; \ \{\mathtt{0}{:}zero \doteq \mathtt{N}{:}nat\} \tag{1}$$

which will be transformed into the system

$$\{\mathtt{N}{:}nat, \alpha \leq zero, \alpha \leq nat\} \ ; \ \{\mathtt{N}{:}\alpha \doteq \mathtt{0}{:}\alpha\} \tag{2}$$

by the rules in figure 3. The typed substitution $\sigma = \{\mathtt{N}/\mathtt{0}\}$ is a solution of (1) and not of (2). But $\sigma$ can be extended to the typed substitution $\sigma' = \{\alpha/zero, \mathtt{N}/\mathtt{0}\}$ which is a solution of (2).

**Deletion of type constraints for variables**

$C \cup \{x{:}\tau'\}; \{x{:}\tau_x \doteq t{:}\tau\} \cup E \quad \xrightarrow{sc} \quad C \cup \{\alpha \leq \tau_x, \alpha \leq \tau'\}; \{x{:}\alpha \doteq t{:}\tau\} \cup E$
if $\alpha$ is a new type parameter

**Deletion of type constraints in equations**

$C; \{x{:}\tau_x \doteq t{:}\tau\} \cup E \quad \xrightarrow{sc} \quad C \cup \{t{:}\tau_x\}; \{x{:}\tau_x \doteq t\} \cup E$
if $x$ does not occur in $C$

**Decomposition of term type constraints**

$C \cup \{f(t_1, \ldots, t_n){:}\tau\}; E \quad \xrightarrow{sc} \quad C \cup \{t_1{:}\tau_1, \ldots, t_n{:}\tau_n, \tau_0 \leq \tau\}; E$
if $f{:}\tau_1, \ldots, \tau_n \to \tau_0$ is a new variant of the type declaration for $f$ in $\Sigma$

**Deletion of multiple variable type constraints**

$C \cup \{x{:}\tau, x{:}\tau'\}; E \quad \xrightarrow{sc} \quad C \cup \{x{:}\alpha, \alpha \leq \tau, \alpha \leq \tau'\}; E$
if $\alpha$ is a new type parameter

Figure 4: Transformation rules for simplifying type constraints on terms

The following theorem states some important properties of the transformation rules for typed unification.

**Theorem 5.3 (Typed Unification)** *Let $\Sigma$ be a polymorphic signature with least types and $\xrightarrow{tu}^*$ be the reflexive and transitive closure of the relation defined in figure 3.*

1. *If $C; E \xrightarrow{tu}^* C'; E'$, then each solution of $C'; E'$ is a solution of $C; E$ and each solution of $C; E$ can be extended to a solution of $C'; E'$.*
2. *Each derivation w.r.t. $\xrightarrow{tu}$ terminates.*
3. *Let $C; E$ be solvable and $C; E \xrightarrow{tu}^* C'; E'$ where $C'; E'$ is irreducible, i.e., no rule is applicable to $C'; E'$. Then $E'$ has the form $\{x_1{:}\tau_1 \doteq t_1{:}\tau_1, \ldots, x_k{:}\tau_k \doteq t_k{:}\tau_k\}$ where $x_1, \ldots, x_k$ are pairwise distinct variables which do not occur in $t_1, \ldots, t_k$. We call a pair $C'; E'$ with this property in* **normal form**.

The normal form of a type constraint and equation system $C; E$ may contain complex type constraints on structured terms which can be easily simplified. Therefore we apply the transformation rules in figure 4 to systems in normal form in order to obtain a type constraint and equation system which has a very simple form. The next theorem states important properties of these simplification rules:

**Theorem 5.4 (Simplification)** *Let $\Sigma$ be a polymorphic signature with least types, $\xrightarrow{sc}^*$ be the reflexive and transitive closure of the relation defined in figure 4 and $C; E$ be in normal form.*

1. *If $C; E \xrightarrow{sc}^* C'; E'$, then each solution of $C'; E'$ is a solution of $C; E$ and each solution of $C; E$ can be extended to a solution of $C'; E'$.*
2. *Each derivation w.r.t. $\xrightarrow{sc}$ terminates.*

3. *Let $C; E$ be solvable and $C; E \overset{sc}{\longrightarrow}^* C'; E'$ where $C'; E'$ is irreducible, i.e., no rule from figure 4 is applicable to $C'; E'$. Then $E'$ has the form $\{x_1{:}\tau_1 \doteq t_1, \ldots, x_k{:}\tau_k \doteq t_k\}$ where $x_1, \ldots, x_k$ are pairwise distinct variables which do not occur in $t_1, \ldots, t_k$, and $C$ has the form*

$$\{\xi_1 \leq \xi_1', \ldots, \xi_l \leq \xi_l'\} \cup \{y_1{:}\tau_{y_1}, \ldots, y_m{:}\tau_{y_m}\}$$

*where $y_1, \ldots, y_m$ are pairwise distinct variables different from $x_1, \ldots, x_k$. We call a pair $C'; E'$ with this property in* **solved form**.

**Example 5.5** Consider the following polymorphic signature:

|  |  |
|---|---|
| **type** | $s_0$, $s_1$, $s_2$ |
| **subtype** | $s_0 \leq s_1$ |
|  | $s_0 \leq s_2$ |
| **func** | `a0`: $\to s_0$ |
| **func** | `f` : $s_1$, $s_2$ $\to s_0$ |

and the type constraint and equation system

$$\{\texttt{X}{:}\alpha, \texttt{Y}{:}\beta\} \; ; \; \{\texttt{f(X,Y)}{:}s_0 \doteq \texttt{f(Y,a0)}{:}s_0\}$$

We obtain the following system in normal form after applying the rules in figure 3 (we omit multiple occurrences of the same constraint):

$$\{\texttt{X}{:}\alpha, \texttt{Y}{:}\beta, s_0 \leq s_0\} \; ; \; \{\texttt{X}{:}s_1 \doteq \texttt{a0}{:}s_1,\ \texttt{Y}{:}s_2 \doteq \texttt{a0}{:}s_2\}$$

The application of the simplification rules in figure 4 yields the following system in solved form:

$$\{s_0 \leq s_0, \gamma \leq \alpha, \gamma \leq s_1, \delta \leq \beta, \delta \leq s_2, s_0 \leq \gamma, s_0 \leq \delta\} \; ; \; \{\texttt{X}{:}\gamma \doteq \texttt{a0},\ \texttt{Y}{:}\delta \doteq \texttt{a0}\}$$

The type constraints in this system are solvable and $\{\gamma/s_0, \alpha/s_0, \delta/s_0, \beta/s_0, \texttt{X/a0}, \texttt{Y/a0}\}$ is a solution of this system and $\{\alpha/s_0, \beta/s_0, \texttt{X/a0}, \texttt{Y/a0}\}$ is a solution of the original system.

Now we are in the following position. In order to unify two typed terms, we transform the type constraints of the variables together with an equation between the two terms into a reduced type constraint and equation system by the rules for typed unification in figure 3 and simplification in figure 4. If the reduced system is not in solved form, then the two terms are not unifiable by theorems 5.3 and 5.4. Otherwise the system has the solved form

$$\{\xi_1 \leq \xi_1', \ldots, \xi_l \leq \xi_l'\} \cup \{y_1{:}\tau_{y_1}, \ldots, y_m{:}\tau_{y_m}\} \; ; \; \{x_1{:}\tau_1 \doteq t_1, \ldots, x_k{:}\tau_k \doteq t_k\}$$

which is solvable iff the subtype constraints for the relation $\leq$ are solvable. Hence to decide the unifiability of two typed terms, we must decide the solvability of a type constraint system of the form

$$CS = \{\tau \leq \tau' \mid \tau, \tau' \in T_{\mathcal{H}}(\mathcal{X})\}$$

Generally, we allow arbitrary Horn clauses for the definition of $\leq$ and therefore this problem is undecidable. Fortunately, there are restricted but interesting type structures for which positive results are known:

1. Smolka [Smo89] allows subtype relations between arbitrary type constructors (e.g., between basic types and polymorphic types), but he requires that all type constructors must be monotonic in their arguments and he has some further requirements on the type structure (see [Smo89] for details). Under these conditions the solvability of $CS$ is decidable if $CS$ does not contain type parameters. If $CS$ contains type parameters, the solvability is an open problem in his framework.

2. Hill and Topor [HT90] also require the monotonicity of all type constructors and they allow only subtype relations between type constructors of the same arity. The solvability of $CS$ is decidable under these restrictions.

3. Fuh and Mishra [FM88] have worked on the problem of polymorphic type inference for a functional language which includes subtypes. In their approach they have also treated the problem of finding a solution of a set of subtype constraints. They have developed a solving algorithm for the case where there are only subtype relations between basic types and all type constructors (like "→" for function space and *pair* for products) are monotonic or anti-monotonic in their arguments. Their algorithm is divided into three parts:

   (a) *match* is the first part which transforms the subtype constraints into subtype constraints where the left-hand side and the right-hand side have the same shape (e.g., $\alpha \leq list(nat)$ is transformed into $list(\beta) \leq list(nat)$ by substituting $\alpha$ by $list(\beta)$).

   (b) *simplify* reduces the subtype constraints into a set of subtype constraints between basic types and type parameters by considering the (anti-) monotonicity property of the type constructors (e.g., $list(\beta) \leq list(nat)$ is reduced to $\beta \leq nat$).

   (c) *consistent* checks whether there exists a substitution for the type parameters such that all basic subtype constraints are satisfied.

   Hence we can use their algorithm to decide the unifiability of terms in our typed framework if there are only subtype relations between basic types and all type constructors are monotonic or anti-monotonic in their arguments, i.e., if all subtype declarations have the form

$$\tau \leq \tau' \qquad \text{where } \tau \text{ and } \tau' \text{ are basic types}$$

   or

$$\alpha_1 \leq \beta_1, \ldots, \alpha_n \leq \beta_n \;\Rightarrow\; h(\alpha_1, \ldots, \alpha_n) \leq h(\beta_1, \ldots, \beta_n)$$

   or

$$\alpha_1 \leq \beta_1, \ldots, \alpha_n \leq \beta_n \;\Rightarrow\; h(\beta_1, \ldots, \beta_n) \leq h(\alpha_1, \ldots, \alpha_n)$$

   (or mixtures of the last two cases). Thus we have found a unification algorithm for the important case of logic programs with higher-order programming techniques and a parametric order-sorted type system (see also section 7).

   Since we are mainly interested in type systems with these restrictions, we will discuss the restriction to "polymorphic signatures with least types" w.r.t. such type structures. Since all subtype relations between types are consequences of inclusions between basic types, we assume that the set of basic types with its subtype relation can be extended to a lattice by augmenting bottom and top elements $\bot$ and $\top$ which are considered as type errors (since there are no terms of this type). Unfortunately, this is not sufficient for least types. For example, consider a polymorphic constant like

$$\textbf{func}\ \texttt{[]}:\quad \rightarrow \quad \mathit{list}(\alpha)$$

Then the term `[]` has types *list*(*zero*) and *list*(*posint*) but there is no valid common subtype of these two types. Hence the signature of figure 1 does not have least types. Smolka [Smo89] solves this problem by introducing a bottom type $\bot$ which is a subtype of any type, i.e., *list*($\bot$) is the least type of `[]`. But this causes the problem that there are subtype relations between basic types and type constructors which we want to avoid in order to apply Fuh and Mishra's algorithm. Another solution can be found in Reynolds' polymorphic typed lambda calculus [Rey74] where a type must be specified if a polymorphic function should be applied, i.e., the first argument of a polymorphic function is always a type. Although we can not deal with types at the object level in our framework, we can simulate this idea by changing the declaration of the empty list into

$$\textbf{func}\ \texttt{[]}:\quad \alpha \quad \rightarrow \quad \mathit{list}(\alpha)$$

Now the argument of `[]` indicates the type instantiation of the polymorphic constant, i.e., the term `[](X)` has type *list*(*posint*) if variable `X` has type *posint*. Therefore the least type of the term `[](...)` can be computed from the least type of the argument.

Thus in order to satisfy the condition for least types, we transform typed logic programs in the following way. For each function originally declared by $f{:}\tau_1,\ldots,\tau_n \rightarrow \tau$ where $\{\alpha_1,\ldots,\alpha_k\}$ ($k > 0$) are the type parameters occurring in $\tau$ but not in $\tau_1,\ldots,\tau_n$, we do the following: Change the declaration of $f$ into

$$f{:}\,\alpha_1,\ldots,\alpha_k,\tau_1,\ldots,\tau_n \rightarrow \tau$$

and add $k$ new variables of appropriate types (the current instances of the $\alpha_i$) as new arguments in each occurrence of $f$ in the program clauses. Since this transformation can automatically be done, we omit it in the examples of this paper.

We will use the typed unification procedure presented in this section to unify an atom in a goal with a head of a clause. In order to apply the typed unification procedure for this case we introduce a new basic type *bool* and declare each predicate symbol of type

$$p{:}\,\tau_1,\ldots,\tau_n$$

as a function symbol of type

$$p{:}\,\tau_1,\ldots,\tau_n \rightarrow \mathit{bool}$$

Then we can unify two $(\Sigma, V)$-atoms $A_1$ and $A_2$ as follows: Transform the pair

$$V; \{A_1{:}\mathit{bool} \doteq A_2{:}\mathit{bool}\}$$

by applying the rules for typed unification and, if a normal form is obtained, the rules for simplification. If the result of this transformation is a pair $C; E$ in solved form, we write

$$V; \{A_1{:}\mathit{bool} \doteq A_2{:}\mathit{bool}\} \quad \xrightarrow{u} \quad V_0; C_0; E$$

where $V_0 = \{y_1{:}\tau_{y_1}, \ldots, y_m{:}\tau_{y_m}\}$, $C_0 = \{\xi_1 \le \xi_1', \ldots, \xi_l \le \xi_l'\}$ and $V_0 \cup C_0 = C$. The set of equations $E$ can be interpreted as an explicit representation of a typed unifier if the corresponding set of type constraints is solvable. Therefore we define the **solutions of the type constraint system** $C_0$ by

$$Sol(C_0) := \{\phi \in TS(\mathcal{H}, \mathcal{X}) \mid \phi \text{ is a solution of all constraints in } C_0\}$$

If $\phi \in Sol(C_0)$ and $E = \{x_1{:}\tau_1 \doteq t_1, \ldots, x_k{:}\tau_k \doteq t_k\}$, then

$$\sigma_E^\phi \;:=\; \{\alpha/\phi(\alpha) \mid \phi(\alpha) \neq \alpha, \alpha \in \mathcal{X}\} \cup \{x_1/t_1, \ldots, x_k/t_k\} \;\in Sub_\Sigma(\mathcal{X}, V, \phi(V_0))$$

is called the **typed substition corresponding to** $\phi$ **and** $E$. The following lemma shows that $\sigma_E^\phi$ is indeed a well-defined typed substition:

**Lemma 5.6** *Let $V; \{A_1{:}bool \doteq A_2{:}bool\} \xrightarrow{u} V_0; C_0; E$ and $\phi \in Sol(C_0)$ be a solution of $C_0$. Then $\sigma_E^\phi$ is a typed substition from $Sub_\Sigma(\mathcal{X}, V, \phi(V_0))$ with $\sigma_E^\phi(A_1) = \sigma_E^\phi(A_2)$.*

The next lemma shows that the typed unification algorithm computes a complete set of unifiers:

**Lemma 5.7** *Let $\Sigma$ be a polymorphic signature with least types, $A$ and $A'$ be $(\Sigma, V)$-atoms and $\sigma \in Sub_\Sigma(\mathcal{X}, V, V')$ be a typed substitution with $\sigma(A) = \sigma(A')$. Then there is a derivation*

$$V; \{A{:}bool \doteq A'{:}bool\} \xrightarrow{u} V_0; C_0; E$$

*and $\phi \in Sol(C_0)$ and $\theta \in Sub_\Sigma(\mathcal{X}, \phi(V_0), V')$ with $\theta \circ \sigma_E^\phi =_V \sigma$.*

## 6 Resolution

The resolution method in untyped Horn logic (see [Llo87]) is an efficient procedure to prove validity of goals w.r.t. Horn clause programs. It is the basic operational principle of logic programming languages like Prolog. Therefore we want to adopt this method for typed logic programs. Since types influence validity or, from an operational point of view, types restrict the set of applicable clauses in a resolution step, it is necessary to modify the resolution method from untyped Horn logic. In our framework we have to replace the untyped unification procedure in a resolution step by a typed one. In the last section we have presented a unification procedure for typed terms: it takes a set of constraints (initially the type declarations for variables) and a set of equations and produces a new set of type constraints and a new set of equations in solved form (if a unifier exists).

We call a $\Sigma$-clause a **variant** of another $\Sigma$-clause if it is obtained by replacing type parameters and typed variables by other type parameters and typed variables, respectively, such that different variables are replaced by new different variables. Let $(\Sigma, \mathcal{P})$ be a typed logic program, $V$ be a set of typed variables, and $G \cup \{L\}$ be a $(\Sigma, V)$-goal. Then a resolution step is defined by the ternary relation

$$V; G \cup \{L\} \quad \xrightarrow{r}_{\Sigma,\mathcal{P}} \quad \sigma_E^\phi \quad \phi(V_0); \sigma(G \cup G')$$

where $L' \leftarrow G'$ is a $(\Sigma, V)$-clause which is a variant of a clause from $\mathcal{P}$ and has no variables in common with $G \cup \{L\}$, and there exists a unification $V; \{L{:}bool \doteq L'{:}bool\} \xrightarrow{u} V_0; C_0; E$ with $\phi \in Sol(C_0)$. Note that $\sigma_E^\phi$ is a typed substitution from $Sub_\Sigma(\mathcal{X}, V, \phi(V_0))$ by lemma 5.6.

A **resolution** is a sequence of the form

$$V_0; G_0 \quad \xrightarrow{r}_{\Sigma,\mathcal{P}} \sigma_1 \;\; V_1; G_1 \quad \xrightarrow{r}_{\Sigma,\mathcal{P}} \sigma_2 \quad \cdots \quad \xrightarrow{r}_{\Sigma,\mathcal{P}} \sigma_n \;\; V_n; G_n$$

where $V_i$ is a set of typed variables and $G_i$ is a $(\Sigma, V_i)$-goal (for $i = 0, \ldots, n$). This resolution will be also denoted by

$$V_0; G_0 \quad {\xrightarrow{r}_{\Sigma,\mathcal{P}}}^{n} \; \sigma \quad V_n; G_n$$

where $\sigma := \sigma_n \circ \cdots \circ \sigma_1$. The resolution is called **successful** if $G_n = \emptyset$. In this case $n$ is called the length of the resolution, and $\sigma$ is called a **computed answer**. We replace ${\xrightarrow{r}_{\Sigma,\mathcal{P}}}^{n}$ by ${\xrightarrow{r}_{\Sigma,\mathcal{P}}}^{*}$ if the precise value of $n$ is not needed.

**Theorem 6.1 (Soundness of resolution)** *Let $(\Sigma, \mathcal{P})$ be a typed logic program, $V$ be a set of typed variables and $G$ be a $(\Sigma, V)$-goal. If there is a successful resolution $V; G \; {\xrightarrow{r}_{\Sigma,\mathcal{P}}}^{*} \; \sigma \; V'; \emptyset$, then $(\Sigma, \mathcal{P}, V') \models \sigma(G)$.*

Similarly to the untyped case, resolution is only complete in the sense that every correct answer is an instance of a computed answer:

**Theorem 6.2 (Completeness of resolution)** *Let $\Sigma$ be a polymorphic signature with least types, $(\Sigma, \mathcal{P})$ be a typed logic program, $V$ be a finite set of typed variables and $G$ be a $(\Sigma, V)$-goal. If $\sigma \in Sub_\Sigma(\mathcal{X}, V, V')$ is a typed substitution with $(\Sigma, \mathcal{P}, V') \models \sigma(G)$, then there exist a set of typed variables $V_0 \supseteq V$ and a resolution $V_0; G \; {\xrightarrow{r}_{\Sigma,\mathcal{P}}}^{*} \; \sigma_0 \;\; V_1; \emptyset$. Furthermore, there is a typed substitution $\theta \in Sub_\Sigma(\mathcal{X}, V_1, V')$ with $\theta \circ \sigma_0 =_V \sigma$.*

These two theorems justify the implementation of resolution with our typed unification procedure as a proof method for logic programs with parametric and order-sorted types. For the computation of a typed unifier in each resolution step our method presented in section 5 can be used. This unification procedure transforms the unification problem into a set of type constraints. In the description of the resolution method we have assumed that a solution of these type constraints is immediately computed in each resolution step. But it is also possible to collect all generated type constraints in the resolution process and solve this constraints after deriving the goal to the empty goal. Such a method is similar to "constraint logic programming" [JL87] and may save unnecessary backtracking steps over different solutions of the type constraints. However, it must be checked whether the type constraints are solvable in each resolution step. Otherwise we lose the advantage of reducing the search tree by integrating types into the resolution process.

## 7 Applications: Higher-order programming

Higher-order programming is an important programming technique used in functional programming languages because it leads to smaller and more readable programs. Many researchers have also tried to integrate higher-order features into logic programming languages. A semantically clean integration of such features into logic programming needs a unification procedure on lambda expressions. A logic language with such a feature has been proposed by Miller and Nadathur [MN86]. Since higher-order unification is a complex task and undecidable in general, it has been argued that it is

```
type      zero, posint, nat, list/1, pred1/1
subtype   zero ≤ nat
          posint ≤ nat
          α ≤ β  ⇒  list(α) ≤ list(β)
          β ≤ α  ⇒  pred1(α) ≤ pred1(β)

func      0:  →  zero
func      s:  nat  →  posint
func      []       :             →  list(α)
func      [..|..]: α, list(α)  →  list(α)
func      λeven   : →  pred1(nat)

pred      has_property:  list(α),  pred1(α)
pred      apply1:  pred1(α),  α
pred      even   :  nat

even(0) ←
even(s(s(N))) ← even(N)

has_property([],P) ←
has_property([E|L],P) ← apply1(P,E), has_property(L,P)

apply1(λeven,N) ←  even(N)
```

Figure 5: A typed logic program with higher-order predicates

sufficient to simulate higher-order programming techniques by a first-order specification of an `apply` predicate [War82] since there is a systematic and efficient method to translate lambda expressions into Prolog [CvER89]. Although this method has been used to implement a polymorphically typed functional-logic language with higher-order objects [BG86], it has been shown in [Han89b] that this approach is incompatible with polymorphic type systems for logic programming like [MO84] and [Smo89]. Since some restrictions of these type systems are dropped in our framework and we do not require the monotonicity of type constructors, we can use Warren's method to integrate higher-order programming techniques into a logic language with a parametric order-sorted type system.

We demonstrate Warren's idea by a simple example. For this purpose we want to define a binary predicate `has_property` which is satisfied if all elements of a list (fist argument) have a certain property (second argument). The property is described as a unary predicate (cf. [SS86], p. 281). In order to treat unary predicates as objects, we define for each unary predicate $p$ of type "$\tau$" a corresponding constant $\lambda p$ of type "$pred1(\tau)$". $pred1$ is a type constructor which denotes the type of unary predicates and is anti-monotonic in its argument because all unary predicates defined on a type $\tau$ can be used if a unary predicate defined on a subtype is required. The relation between each unary predicate $p$ and its functional abstraction $\lambda p$ is specified by Horn clauses for the predicate `apply1`. Figure 5 contains the complete typed logic program for this example. Note that the clause for `apply1` is not well-typed in the sense of [MO84] and [Smo89] because in the head of this clause `apply1` is used with an instance of its declared type which is forbidden in these type systems.

In order to show an application of our typed unification procedure we define an additional predicate which is satisfied if a fixed list of positive integers satisfies a certain property:

**pred** `listprop:` $\mathit{pred}1(\mathit{posint})$

```
listprop(P) ← has_property([s(s(0)), s(s(s(s(0)))), s(s(0))], P)
```

If we want to prove the goal

```
listprop(λeven)
```

this atom has to be unified with the head of the clause, i.e., the typed unification procedure is started with the following type constraint and equation system:

$$\{\mathtt{P}{:}\mathit{pred}1(\mathit{posint})\}\ ;\ \{\mathtt{listprop}(\lambda\mathtt{even}){:}\mathit{bool} \doteq \mathtt{listprop}(\mathtt{P}){:}\mathit{bool}\}$$

The application of the rules for typed unification and simplification in figures 3 and 4 yields the following system in solved form:

$$\{\mathit{bool} \leq \mathit{bool},\ \mathit{pred}1(\mathit{nat}) \leq \alpha,\ \alpha \leq \mathit{pred}1(\mathit{posint})\}\ ;\ \{\mathtt{P}{:}\alpha \doteq \lambda\mathtt{even}\}$$

The type substitution $\{\alpha/\mathit{pred}1(\mathit{posint})\}$ is a solution of the last type constraint system since $\mathit{pred}1(\mathit{nat}) \leq \mathit{pred}1(\mathit{posint})$ is a logical consequence of the specification for $\leq$ in figure 5. This solution can be computed by the algorithm in [FM88].

This example shows that it is possible to treat higher-order objects in our typed framework. Generally, it is possible to translate arbitrary lambda expressions into clauses for an `apply` predicate [CvER89]. More details about this method of higher-order logic programming in a polymorphically typed framework can be found in [Han89b].

## 8 Conclusions and related work

We have presented a declarative type system for logic programs which combines parametric and inclusion polymorphism. In order to drop limitations of other type systems with a similar goal, we have assumed that the inclusion order is specified by Horn clauses for the subtype relation $\leq$. This allows the declaration of type structures where the type constructors are not required to be monotonic. Therefore logic programs with a parametric order-sorted type structure including higher-order predicates can be specified in our framework.

We have defined the semantics of our type system in a model-theoretic way. Parametric types are interpreted as a universal quantification over all types, and order-sorted type structures are interpreted as order-sorted algebras [SNGM89]. On the operational side we have shown that the well-known resolution principle can be used to prove goals if the untyped unification is replaced by a unification procedure which considers the types of the terms. We have presented such a unification procedure for our typed framework. It takes a pair of terms together with the variable types as input and produces a set of subtype constraints as the result if the terms are unifiable. The satisfiability of such subtype constraints is decidable for particular classes of type structures, e.g., where only basic types are related by subtype inclusion and all type constructors are monotonic or anti-monotonic in their arguments. This includes the class of typed logic programs with higher-order objects.

Smolka [Smo89] and Hill and Topor [HT90] have also proposed typed logic languages with parametric and order-sorted types. In their framework the heads of clauses defining polymorphic predicates must be of the most general type and all type constructors must be monotonic in their arguments. This excludes an important programming technique as shown in section 7. Our framework drops the first restriction and assumes that the subtype relation is declared by Horn clauses. Therefore we only require that the subtype relation is a quasi-ordering (which can be specified by Horn clauses) and not a partial order as required in [Smo89] and [HT90]. This causes no problems in the semantics since quasi-orderings are sufficient for order-sorted logic [Smo86].

Another approach to polymorphic type systems with subsorts for logic programming has been presented in [Han90] where subsort relationships are described by equations. This has the advantage that well-known equation solving techniques can be used for the typed unification procedure but the disadvantage that the combination of polymorphism and subtyping is more restricted. Moreover, the semantics of our presented framework is a direct extension of order-sorted logic ("subsorts are subsets") in contrast to [Han90].

There are a lot of directions for further work. For instance, we have cited the decidability results of Fuh and Mishra [FM88] which are restricted to type structures where all type constructors are monotonic or anti-monotonic in their arguments and no other subtype relations between type constructors exists. But it seems possible to extend this algorithm to the case where subtype relations between type constructors of the same arity are allowed, since Hill and Topor have developed positive results for similar type structures (with monotonic type constructors). Another research direction is the improvement of the type checks in the unification procedure. For a lot of cases it seems that the type checks can be simplified (e.g., for monomorphic goals [Smo89]) or completely omitted (for type structures without subtypes and with restrictions on the use of polymorphic predicates [Han89b]). The development of such optimizations is important for an efficient implementation of our typed logic language.

## References

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

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

[BG89] R. Barbuti and R. Giacobazzi. A Bottom-Up Polymorphic Type Inference in Logic Programming. Technical Report 27/89, Dip. di Informatica, Università di Pisa, 1989.

[CvER89] M.H.M. Cheng, M.H. van Emden, and B.E. Richards. On Warren's Method for Functional Programming in Logic. Report LP-12 DCS-122-IR, Univ. of Victoria, 1989.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *acm computing surveys*, Vol. 17, No. 4, pp. 471–523, 1985.

[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.

[FM88] Y.-C. Fuh and P. Mishra. Type Inference with Subtypes. In *Proc. ESOP 88, Nancy*, pp. 94–114. Springer LNCS 300, 1988.

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

[Han90] M. Hanus. Logic Programs with Equational Type Specifications. In *Proc. of the 2nd International Conference on Algebraic and Logic Programming*, pp. 70–85. Springer LNCS 463, 1990.

[Han91] M. Hanus. Parametric Order-Sorted Types in Logic Programming. Technical Report, FB Informatik, Univ. Dortmund, 1991.

[HT90] P.M. Hill and R.W. Topor. A Semantics for Typed Logic Programs. Report TR-90-11, Computer Science Department, University of Bristol, 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.

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

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

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

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

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

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

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

[Rey74] J.C. Reynolds. Towards a Theory of Type Structure. In *Proc. Colloque sur la Programmation*, pp. 408–425. Springer LNCS 19, 1974.

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

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

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

[SNGM89] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-Sorted Equational Computation. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2, Rewriting Techniques*, chapter 10, pp. 297–367. Academic Press, New York, 1989.

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

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

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

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

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