Informatik Berichte 200, FernUniversität Hagen, Germany

Type Checking and Type Inferencing for Logic Programs with Subtypes and Parametric Polymorphism*

Gregor Meyer
FernUniversität Hagen, Praktische Informatik VIII
58084 Hagen, Germany
gregor.meyer@fernuni-hagen.de

June 1996

Abstract

We analyze basic shortcomings of existing proposals for type checking and type inferencing in logic programming languages. A new type system is presented using simple and declarative type annotations. It includes parametric polymorphism and subtyping. Static type checking and inferencing within the new type system is able to detect more programming errors than in other comparable systems. Our approach is independent of a specific resolution calculus and therefore applicable to a wide range of logic languages. A type inferencing algorithm is presented for reconstructing variable typings. As a concrete instance, the results were applied to unrestricted standard Prolog with type annotations, for which a type checking and type inferencing tool is available.

This report is part of the documentation for a type checking tool available at `<URL:http://www.fernuni-hagen.de/pi8/typical/>`. It is complemented by a report "On the Use of Types in Logic Programming" [Mey96].

*Research supported by DFG within "Schwerpunkt Deduktion"

Contents

1 Introduction

The integration of logic programming with type systems has been a research issue since more than a decade. The major goal is to detect programming errors automatically at compile time. Unfortunately, there is no common agreement on the role of types in logic programs. Especially the conditions required for a program to be well-typed (or type correct) differ. The collection in [Pfe92] contains some of the approaches. In this paper we show deficiencies of generally used semantics for well-typing in type systems with subtyping and parametric polymorphism (e.g., [Smo89, Bei92, HT92, Han91, YFS92] and others). These problems concern the ability to check for type errors and also semantical problems with type inferencing. A new method is presented in order to solve these problems. Further advantages of our approach are that the type annotations retain the declarative aspect of logic programming without imposing a procedural or directional view. It can be seen as a conservative extension to other type systems in the sense that, using the same type declarations, a clause detected to be not well-typed by conventional systems can also be detected as not well-typed by our new system. With our proposed simple refinements of type declarations we are able to detect more programming errors. Based on this type system we developed a type checking tool for standard Prolog with type annotations and parametric types and subtypes. Other type checking tools [O'K84, Lak91] for Prolog are only able to work with type systems that exclude subtyping.

The rest of this introduction section informally describes the problems to be solved and our solution. Section 2 defines our notion of well-typing, called type-consistency, formally. Section 3 presents a type inferencing algorithm which infers types of variables. Section 4 discusses properties of our system and its relations to other sytems proposed in the literature. The last section gives a conclusion and an outlook on applications of this approach.

1.1 Shortcomings of other type systems

Starting with Mishra ([Mis84]), several authors have proposed type checking mechanisms that do not require explicit type declarations as part of the program. A type inference procedure is used to find so-called *useless* clauses. Useless clauses are clauses where the body will never succeed.

Example 1

```
append([],L,L).
append([X|Xs],L,[X|XL]) :- append(Xs,L,XL).

q :- append(a,L,L).
```

```
reverse([],[]).
reverse([X|Xs],R) :- reverse(Xs,R0),
                     append(R0, X ,R).
```

In this example the head atoms in the defining clauses for `append` can be detected to only have list constructors in the first argument. An atom such as `append(a,L,L)` can be detected as useless automatically, because the term `a` does not unify with any list. Because the body of the clause for `q` can not succeed, the whole clause is marked as erroneous. The example contains another error in the second clause for `reverse`; the second subgoal should read `append(R0,[X],R)`. Hence one might expect an automatic type checking tool to detect this error as well. However, the definition for `reverse` as given by the Prolog clauses in our example actually has solutions such as:

```
reverse([X],X).
reverse([[a],[]],[a]).
reverse([[a,b],[c,d]],[c,d,a,b]).
reverse([[a],[x]],[x,a]).
reverse([[a],[[x]]],[[x],a]).
...
```

Hence the erroneous clause for `reverse` is not useless. Possibly, such a programming error is found indirectly due to some restrictions in the type language. I.e., if the first argument of `reverse` is detected to have a type which is not expressible in the type language at hand, the clause would be rejected as ill-typed. Nevertheless, without programmer-supplied type declarations the ability to find type errors is rather restricted, because hints for the intended meaning of predicates are missing.

An ML-like type system for logic programming was proposed by Mycroft and O'Keefe ([MO84]. It includes explicit type declarations and parametric polymorphism but no subtypes. Most prominently the languages Gödel [HL94] and Mercury [Som95] are based on this kind of type system. But it is not possible to model often needed type hierarchies as in 'integers are numbers and numbers are expressions', not to speak of object-oriented extensions or feature logic with sorts.

Type systems with parametric polymorphism and subtypes were proposed by [Smo89, HT92, Han91, Han92] and others. However, all these approaches do have a serious shortcoming when it comes to detect obviously ill-typed expressions. A short example, showing an inherent anomaly in these systems, is

Example 2

Program with Declarations

```
nat ≤ int.  negint ≤ int.
1 : nat.      2 : nat.      -1 : negint.
[] → list(T).
[|]:  T x list(T) → list(T).

pred '=': T × T.     % standard syntactic equality
pred abs: int × nat.
   abs(Z,X) :- X is the absolute value of Z.

pred p: list(nat).
   ...
```

Goals

```
?- abs(Z,X), X = -1.

?- p(L), L = [-1].
?- p(L), eq(L, [-1]).
?- p(L), append([1], [-1], L).
```

The constant `-1` is of type `negint` and hence by subtyping `-1` is also of type `int`. Lists are defined (as built-in) with type parameters, i.e. they are generic, where the type `list(int)` denotes all lists of elements of the type `int`, such as `[1,1,-1]`. That list is not of type `list(nat)` since it contains `-1` which is not of type `nat`. Parametric types are generally defined as monotonic with respect to their argument types. E.g., if `t1` is a subtype of `t2`, then `list(t1)` is a subtype of `list(t2)`.

Relations, too, have an associated type, defined in predicate declarations marked by the keyword `pred`. While `p` is a unary predicate where the argument must be of type `list(nat)`, the binary predicate `'='` is declared with a type parameter `T`. The arguments of `=` can be of any type, but the types must be the same. Similarly, all three arguments of `append` must be lists of the same type. We will see in the following that the notion "of the same type" appears to be critical for type checking. One of the distinctive properties of our new approach is, that we take *least* types of terms into account.

With the conventional approaches to parametric polymorphism an atom $p(t_1, \ldots, t_n)$ is well-typed iff the predicate p is declared as $p : \pi_1, \ldots, \pi_n$, each argument t_i is of some type τ_i, and there is a type substitution Θ such that τ_i is a subtype of $\Theta(\pi_i)$ for all i.

With `-1` of type `negint` and the variable `X` having the type `nat` the goals is considered as well-typed within the systems cited above, because the types `negint` and `nat` have the common supertype `int`. But intuitively we can see that the equation `X = -1` with `X` of type `nat` is erroneous. With the intended meaning of `abs` and `'='` the goal can never succeed.

For the same reasons, the other goals in the example will fail. Because the variable `L` has the type `list(nat)` it can not be unified with a list that contains negative integers. However, the goals are well-typed with the conventional interpretation of parametric declarations. E.g., the last goal is well-typed, because there is an instance of the type of `append`, in this case `list(int) x list(int) x list(int)`, such that all actual arguments in the atom `append([1],[-1],L)` conform to their formal argument types. But, in any reasonable procedural or functional language with subtypes an expression such as `X = -1` where `X` has type `nat` would not be well-typed. Similarly, if there is a function `appfun:list(T) x list(T) --> list(T)` an expression `L = appfun([1],[-1])` would not be well-typed with L having the type `list(nat)`, because `L` should have the type `list(int)`.

Furthermore, if the type of integers and, e.g., the type of characters have no common supertype, an equation `X = 'a'` where `X` has type `int` would be detected as not well-typed. But if a common supertype is introduced the equation becomes well-typed. Although it might be argued that such a supertype is not needed, many Prolog programmers would like to have such a supertype. Otherwise, several serious Prolog programs could not be made well-typed. Also for other variants of logic programming languages, such as object-oriented extensions or variants including hierarchical feature types, taller type hierarchies are important. Sometimes they even contain a most general type. If there is a most general type 'term', then the declarations `p: T x T`, and `p: S x T`, and `p: term` $\times$ `term` are essentially equivalent. Any atom `p(a,b)` is well-typed w.r.t. these declarations in the sense: if `a` and `b` have types `ta` and `tb` respectively, then `p(a,b)` is well-typed regardless of `ta` and `tb`.

Taking a closer look at our example program, one can see that none of the (conventionally well-typed) clauses for `p` has a solution in the intended model. The clauses always fail due to the type constraints on variables. The general observation for conventional type systems with subtypes and parametric polymorphism is: Subtyping tends to undermine the effects of parametric polymorphism in the conventional approaches, as far as the detection of type errors is concerned. In many cases where a procedural or functional expression is not type correct, the corresponding logic clause is considered as well-typed although it never succeeds at run-time. The problem is, that there are too many expressions that are considered as being well-typed. A new type system presented in this paper enables static type checking and type inferencing to spot such errors.

When the type system includes subtypes, the semantics for the typed logic is commonly defined for clauses where the types of all variables are given explicitly, and the resolution is required to perform some kind of order-sorted unification in order to rule out incorrect solutions. In parallel to functional programming systems, a type inference procedure transforms a partially typed program, as it is given by a programmer, into a fully typed program for which a semantics is formally defined. A hitherto neglected problem is that the type inference procedure for a logic program may have semantical impacts. Consider the following example:

Example 3

Declarations

```
% types and subtypes
nat ≤ int.  negint ≤ int.  int ≤ expr.  int ≤ index.
% constants
1 :  int.    ce :  expr.    ci :  index.

pred e:  T x T.
pred q:  T.
  q(ce).  q(ci).
```

Goal

```
?- e(Y,1), q(Y).
```

If type inferencing comes up with the variable typing `Y : expr`, making the goal clause well-typed, then the goal can have the solution `Y = ce`. If type inferencing comes up with `Y : index` then the goal can have the solution `Y = ci`. It is commonly taken for granted that a (complete) type inferencing procedure produces a fully typed program, which has semantics that match the intent of the programmer, who wrote the original, partially untyped program. But as the example shows this is not always possible. There is no type assignment for the variable `Y` which allows both solutions. This short example does not obey the so-called head-condition or genericity-condition, but see [Mey96] for another example showing that even obeying these conditions does not rescue the situation.

1.2 A new approach

Our new approach to solve the problems above takes the view that the type declarations given above are not as exact as they should be. As one of the key elements we propose to use extended type declarations and, more important, a better suited notion for well-typing in the presence of type parameters. For easier reference, we will talk about "**type-consistency**" in order to distinguish our new approach from conventional "well-typing". Our notion of type-consistency can be seen as a combination of well-typing and the detection of useless clauses. While this section only contains an informal presentation of the type system illustrated by examples, in order to get an intuition of the key ideas, exact definitions are given in the next section.

The essential elements of the new typing approach are:

1. Extend the form of predicate declarations.
2. Type parameters match with least types of terms.

The new form of type declarations for predicates is first introduced by means of an example. Intuitively, a type declaration for a predicate describes a superset of the predicate solutions. E.g., the conventional declaration `append`$_{\texttt{old}}$`: list(T) x list(T) x list(T)` defines that for any (ground) atom `append(L1,L2,L3)`, which is true in some model, there is a type substitution Θ for the type parameter `T` such that each argument is of type Θ(`list(T)`). With type hierarchies, possibly being rather deep or even containing a most general type, this semantics leads to the anomaly explained above. As part of our solution we propose a type system that allows more exact declarations such as

```
pred append_new: list(@T1) x list(@T2) x list(@T3)
                                        |> T1 =< T3, T2 =< T3.
```

In order to identify the new form of declarations syntactically, we prefix type parameters within formal argument types with '`@`'. The expressions on the right of `|>` describe *type constraints* where `=<` stands for the subtype relationship. Then an atom `append(L1,L2,L3)`, where the arguments `L1`, `L2`, and `L3` have the *least types* `list(A)`, `list(B)`, and `list(C)` respectively, is type-consistent if the conjunction of type constraints `A =< C, B =< C` is satisfied.

In general, to check if an atom $p(t_1, \ldots, t_n)$ is type-consistent with respect to a declaration $p : \pi_1 \ldots \pi_n \,\rhd C$ the following steps are performed

1. compute the most specific (least) type τ_i of every argument t_i,
2. for each τ_i determine the most specific (least) instance $\Theta_i(\pi_i)$ with $\tau_i \leq \Theta_i(\pi_i)$,
3. check if there is a substitution Θ such that $\Theta(\pi_i) = \Theta_i(\pi_i)$ for each i.
4. check if $\Theta(C)$ is satisfiable.

As usual, a clause is well-typed if every atom in it is well-typed.

Hence, the atom `append(cons(1,nil), cons(-1,nil), L)` is type-consistent with the variable `L` having the (least) type `list(int)`. Given the least type of the first argument `cons(1,nil)` as `list(nat)` and the least type of `cons(-1,nil)` as `list(negint)`, the type constraints `nat =< int` and `negint =< int` are satisfied. But if the variable `L` is constrained to the type `list(nat)` the same atom is not type-consistent, because there is no type `TM` such that `cons(-1,nil)` has (least) type `list(TM)` and also `TM =< nat`. This indicates that the atom as a goal literal will always fail, because with the intended meaning there is no list `L` of natural numbers that contains negative numbers in its tail.

Further examples of new type declarations are:

```
pred '=':  @T x @T.
pred member: @T1 x list(@T2) |> T1 =< T2.
pred reverse: list(@T) x list(@T).
pred overlap: set(@S) x set(@T) |> U =< S, U =< T.
```

New type variables in the constraint part are implicitly existentially quantified.

With the new predicate declaration for '`=`', the goal '`abs(Z,X), X = -1`' in our introductory example is detected as not type-consistent. Also the other goals are automatically detected as being not type-consistent.

Note that by accident the new declaration `'=': @T x @T` is syntactically very similar to the conventional declaration. However, within the new type system the declaration is much stronger. Given the variable typing {`X:nat, Y:negint`}, the (conventionally well-typed) atom `X=Y` is not type-consistent. There is no instance of T which is the least instance wrt `nat` and wrt. `negint` at the same time. Deriving that atom as not type-consistent is reasonable, because with the standard semantics of '=' it has no solutions. Furthermore, the atom `X=Y` is not type-consistent with the variable typing {`X:nat, Y:int`} although there are type-correct instances, e.g., substituting `1` for both `X` and `Y`. Nevertheless, the atom is type-consistent using {`X:nat, Y:nat`}.

As a practical benefit, our system is orthogonal to predicate declarations used in several other type systems. Any predicate declaration of the conventional form `p: typeterm(Vars) x ...` can be transformed into an equivalent declaration in the new form `p: typeterm(@NewVars) x ...  |> NewVars =< Vars, ...` by introducing new type variables. I.e., a declaration `q: T x T` is equivalent to `q: @T1 x @T2 |> T1 =< T, T2 =< T`, where `T` is implicitly existentially quantified, requiring the least types of the arguments to have an upper bound `T`. The old declaration for `append` is thus equivalent to

```
pred append_old: list(@T1) x list(@T2) x list(@T3)
                             |> T1 =< T, T2 =< T, T3 =< T.
```

This declaration is weaker than the declaration for $\texttt{append}_{\texttt{new}}$: There are atoms which are type-consistent w.r.t. $\texttt{append}_{\texttt{old}}$ and not type-consistent w.r.t. $\texttt{append}_{\texttt{new}}$.

The constraints `T1 =< T3, T2 =< T3` imply `T1 =< T, T2 =< T, T3 =< T`, (e.g. let `T = T3`), but not the other way around.

In our type checking system `Typical` for standard Prolog a very short notation is possible for many declarations. E.g., the declaration for 'append' is `append(T,T,@T)` and 'member' is declared as `member(T,@T)`. These forms are internally translated into the corresponding declarations as above.

2 Type consistency

While the introduction gave an intuitive description of our type system, we will now define the type language and well-typing rules for atoms and clauses formally.

2.1 Partial order

Let T be a set of symbols and $\leq : T \times T$ a binary relation. $(T, \leq)$ is a partially ordered set (poset) if $\leq$ is reflexive, transitive, and antisymmetric.

reflexive: $\forall \tau \in T \cdot \quad \tau \leq \tau$

transitive: if $\quad \tau \leq \pi \quad$ and $\quad \pi \leq \omega \quad$ then $\quad \tau \leq \omega$

antisymmetric: if $\quad \tau \leq \pi \quad$ and $\quad \pi \leq \tau \quad$ then $\quad \tau = \pi$

$\omega \in T$ is an upper bound of a set $S = \{\tau_1, \ldots, \tau_n\}$ iff $\tau_i \leq \omega$ for each element in S. ω_l is the least upper bound of S iff ω_l is an upper bound of S and $\omega_l \leq \omega$ for each upper bound of S. A least upper bound is unique if it exists. Lower bounds and greatest lower bounds are defined analogously. The least upper bound of two elements is denoted by $\tau \sqcup \pi$; the greatest lower bound is denoted by $\tau \sqcap \pi$. A poset is a **quasi-lattice** iff $x \sqcup y$ exists whenever $\{x, y\}$ has an upper bound, and $x \sqcap y$ exists whenever $\{x, y\}$ has a lower bound.

2.2 The type language

The type signature Σ for our parametric order-sorted type language consists of

- a finite set $\mathbf{T}$ of type symbols κ with fixed arity$(\kappa) \geq 0$
- a relation '$\leq_s$' : $\mathbf{T} \times \mathbf{T}$
- a (countable infinite) set $\mathbf{TV}$ of type variables α

The set of **base type** symbols $\mathbf{TB}$ is defined as the set of all symbols $\mu \in \mathbf{T}$ with the arity$(\mu) = 0$.

Later we will distinguish a subset $\mathbf{TV}^0$ of $\mathbf{TV}$. To give some intuition in advance, $\mathbf{TV}^0$ contains don't-care type variables, which play a role similar to the type $\perp$ in [Smo89]. They will be exploited for computing principal types.

In the following, $(\mathbf{T}, \leq_s)$ is generally assumed to be a quasi-lattice. This condition is needed in order to be able to compute the principal type of a term. While the definitions for well-typed clauses also work with more general type structures, the restrictions are exploited in our type inference procedure.

2.2.1 Type terms

Given a type signature Σ, **type terms** τ are formed as usual according to the abstract grammar

$$\tau ::= \alpha \mid \kappa(\tau_1, \ldots, \tau_n)$$

We use $\mathbf{TT}$ to denote the set of all type terms.

For convenience, we will use α, β, and γ to denote a type variable, τ, π to denote a type term, and μ and κ to denote type constructor symbols. A ground type term, i.e. without type variables, is called monomorphic, while a type term containing type parameters is called a type scheme, parametric, or polymorphic. In terms of [CW86] parametric polymorphism corresponds to universal quantification and has to be distinguished from subtype polymorphism. In this report, polymorphism will generally mean parametric polymorphism.

2.2.2 Partial order on type terms

The partial order $(\mathbf{T}, \leq_s)$ is used to define a partial order on type terms $(\mathbf{TT}, \leq)$, where $\leq$ is called the **subtype relation** by the following rules.

$$\alpha \leq \alpha \qquad \text{if} \quad \alpha \text{ is a type variable}$$

$$\frac{\tau_1 \leq \pi_1 \wedge \cdots \wedge \tau_n \leq \pi_n}{\kappa(\tau_1, \cdots, \tau_n) \leq \kappa(\pi_1, \cdots, \pi_n)} \qquad \text{if} \quad \begin{array}{l} \kappa \text{ is a type symbol,} \\ \text{arity}(\kappa) = n \geq 0 \end{array}$$

$$\frac{}{\kappa(\tau_1, \cdots, \tau_n) \leq \mu(\tau_1, \cdots, \tau_n)} \qquad \text{if} \quad \begin{array}{l} \kappa \text{ and } \mu \text{ are type symbols,} \\ \text{arity}(\kappa) = \text{arity}(\mu) = n \geq 0 \\ \kappa \leq_s \mu \text{ in } (\mathbf{T}, \leq_s) \end{array}$$

Note that we allow subtyping between type symbols which are not base symbols. However, they must have the same arity.

Proposition 1 *If $(\mathbf{T}, \leq_s)$ is a quasi-lattice then $(\mathbf{TT}, \leq)$ is a quasi-lattice.*

The least upper bound of two compound type terms can be simply computed as $\kappa(\tau_1, \cdots, \tau_n) \sqcup \kappa(\pi_1, \cdots, \pi_n) = \kappa(\tau_1 \sqcup \pi_1, \cdots, \tau_n \sqcup \pi_n)$. The greatest lower bound is computed accordingly.

Example 3 $nat \leq int$ implies $list(nat) \leq list(int)$ as well as $list(list(nat)) \leq list(list(int))$. $bintree(\alpha) \leq tree(\alpha)$ iff $bintree \leq_s tree$ and $\text{arity}(bintree) = \text{arity}(tree) = 1$, $list(\alpha) \leq list(\beta)$ is not true with distinct parameters α and β. $list(nat) \sqcup list(negint) = list(int)$.

2.3 Substitutions

Type substitutions are defined analogously to standard substitution functions on first-order terms. A type substitution maps type variables onto type terms. The domain dom(Θ) is the set $\{\alpha \mid \alpha \neq \Theta(\alpha)\}$. A type substitution Θ for type variables is conservatively extended to a function on type terms using the definition $\Theta(\kappa(\tau_1, \ldots, \tau_m)) =_{\text{def}} \kappa(\Theta(\tau_1), \ldots, \Theta(\tau_m))$. A type substitution Θ is represented by a (finite) set $\{\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n\}$, having the domain $\text{dom}(\Theta) = \{\alpha_1, \ldots, \alpha_n\}$ and $\Theta(\alpha_i) = \tau_i$. For convenience, a type substitution may be applied to general expressions, e.g. constraints and variable typings (see below). Without further notice we always assume that the domain of type substitutions only contains type variables; i.e. value variables are not affected by type substitutions.

Proposition 2 *Given a partial order on type terms. for any type term τ and type substitutions Θ_1 and Θ_2: If $\Theta_1(\tau) \sqcup \Theta_2(\tau)$ is defined, then there exists a type substitution Θ_3 such that $\Theta_1(\tau) \sqcup \Theta_2(\tau) = \Theta_3(\tau)$.*

2.4 Type constraints

Type constraints are of the form

$$\tau \preceq \pi$$

where τ and π may be elements of **TT**. **Simple type constraints** are of the form $\alpha \preceq \pi$ where α is a type variable and π does not contain α. We sometimes use a special kind of type constraints, called **type variable constraints**, of the form $\alpha \preceq \beta$ where α and β are distinct type variables in **TV**. Type substitutions are straightforwardly extended to (sets of) type constraints by defining $\Theta(\tau \preceq \pi) =_{\text{def}} \Theta(\tau) \preceq \Theta(\pi)$.

A set of type constraints $\{\cdots, \tau_i \preceq \pi_i, \cdots\}$ is **satisfied** iff $\tau_i \leq \pi_i$ for each i. A type substitution Θ satisfies a set of type constraints C iff $\Theta(C)$ is satisfied. In this case we say that Θ is a satisfying substitution and C is satisfiable.

For notational convenience we use several abbreviations: $\vec{\tau}$ denotes a tuple $(\tau_1, \cdots, \tau_n)$ with finite length. $\vec{\tau} \leq \vec{\pi}$ is true iff $\vec{\tau}$ and $\vec{\pi}$ have the same length n and $\tau_i \leq \pi_i$ for all i. $\vec{\pi}'$ is an instance of $\vec{\pi}$ iff there is a type substitution Θ such that $\vec{\pi}' = \Theta(\vec{\pi})$. For the computation of matchers we use an additional kind of inequations denoted by $\tau \preceq^u \pi$. $\vec{\tau} \preceq^u \vec{\pi}$ is a shorthand for the set $\{\tau_1 \preceq^u \pi_1, \cdots, \tau_n \preceq^u \pi_n\}$.

2.5 Type consistency

2.5.1 Typed terms

For each **function symbol** we assume a declaration

$$f : \pi_1 \times \cdots \times \pi_n \rightarrow \pi$$

with n$\geq$0 and each π_i and π in **TT** where π is not a type variable. π is called the target type or simply the type of f. If π contains type variables they must be pairwise distinct. There may be more than one declaration for a function symbol. Declarations of function symbols are not required to be transparent[1]. In a transparent declaration every variable occurring in $(\pi_1, \ldots, \pi_n)$ must also occur in π. Besides other consequences, this condition is used for implementing type safe unification efficiently. If a constructor f is declared as $f : V \rightarrow \pi$ where the variable V does not occur in π, it is impossible to determine the type of an argument of π from the type π of the whole term. With a parametric type system and no subtyping transparency is required in order to produce type-correct solutions with an untyped unification. Within the context of this report, i.e., for type checking and type inferencing, transparency is not required and we also allow multiple declarations. Nevertheless, transparency and uniqueness of declarations are properties which can be enforced by simple checks if wanted.

Given countably infinite many (value) variables v, terms t are described by the abstract grammar

$$t ::= v \mid f(t_1, \cdots, t_n)$$

A **variable typing** VT, also called prefix ([Mil78]), is a set of type assignments for variables in the form $\{v_1 : \tau_1, \cdots, v_k : \tau_k\}$, with pairwise distinct (value) variables v_i. Type substitutions are extended to variable typings by $\Theta(\{v_1 : \tau_1, \cdots, v_k : \tau_k\}) =_{\mathrm{def}} \{v_1 : \Theta(\tau_1), \cdots, v_k : \Theta(\tau_k)\}$.

A typing statement $VT \vdash E : \tau$ may be read as "Under the assumptions VT about the types of variables, the expression E *has type* τ". Typing for terms and clauses is defined by *typing rules*, which are commonly used for typed λ-calculus ([DM82, Bar92, Mit91]). $VT \vdash t : \tau$ is true if and only if it is derivable using the rules *(Var), (Func), (Subtype)*:

[1] This condition is called 'transparency' in [HT92] and 'type preserving' in [Han89]. It is also part of the 'minimality condition' in [Smo88].

$$\textit{(Var)} \qquad \frac{}{VT \cup \{v:\tau\} \vdash v:\tau}\mathcal{T}$$

$$\textit{(Func)} \qquad \frac{VT \vdash t_1 : \Theta(\pi_1) \quad \cdots \quad VT \vdash t_k : \Theta(\pi_k)}{VT \vdash f(t_1, \cdots, t_k) : \Theta(\pi)}\mathcal{T} \quad \text{if} \quad \begin{array}{l} f : \pi_1 \times \ldots \times \pi_k \to \pi \\ \text{is a declaration variant,} \\ \Theta \text{ is a type substitution} \end{array}$$

$$\textit{(Subtype)} \qquad \frac{VT \vdash t : \tau}{VT \vdash t : \pi}\mathcal{T} \qquad \text{if} \quad \tau \leq \pi$$

A type τ is called non-empty iff there is at least one ground term t such that $\{\} \vdash t : \tau$. We assume that every variable free type term is a non-empty type. A variable typing is called valid if there is a type substitution Θ such that for each τ occurring in the variable typing $\Theta(\tau)$ is non-empty. We assume that every type is non-empty. Type variables from $\mathbf{TV}^0$ are not allowed to occur in a valid variable typing. Variable typings are generally assumed to be valid.

Note that there may be function declarations where the target type contains type variables not contained in any argument type, such as $nil \to list(\alpha)$. We will call them 'parameter introducing declarations'. In the typing rule *(Func)* the type substitution Θ may assign any type term to these variables. I.e., we can infer $nil : list(nat), nil : list(int), nil : list(\alpha)$, etc. Parameter introducing declarations cause a few details for the definition of type-consistency and in the type inference algorithm given below. We could forbid such declarations and apply a program transformation such as given in [Han91] which introduces new artificial (type) variables but we found such transformations not satisfying.

Given some variable typing VT and a term t, a **principal type** π of the term t has the property: t has type π and if t has type τ then there is a type substitution Θ such that $\Theta(\pi) \leq \tau$. The notion of principal types is well-known from typed functional programming. Our definition is somewhat different because we don't have λ-abstraction and the principal type does not depend on type constraints.

In order to compute principal types, we first define how to compute a **principal upper matcher** of two sequences of type terms $\vec{\tau}$ and $\vec{\pi}$. The set of type variables in $\vec{\tau}$ is assumed to be disjoint from the set of type variables in $\vec{\pi}$. Additionally, if $\vec{\tau}$ contains type variables from $\mathbf{TV}^0$, they must be pairwise different. If the exhaustive application of the following decomposition rules, starting with some set of $\preceq^u$-constraints, terminates with $\{\sigma_1 \preceq^u \alpha_1, \ldots, \sigma_k \preceq^u \alpha_k\}$ with pairwise distinct type variables α_i, then $\Theta' =_{\text{def}} \{\alpha_1 \leftarrow \sigma_1, \ldots, \alpha_k \leftarrow \sigma_k\}$. The composition of Θ' and all type substitutions applied during the decomposition, restricted to the type variables occurring in $\vec{\pi}$, is called the principal upper matcher, p.u.m., of $\vec{\tau} \preceq^u \vec{\pi}$. Otherwise the p.u.m. does not exist. Principal upper matchers are similar to 'least upper matchers' ([Smo89, p.88ff]) which require the existence of a least type term $\perp$.

For technical convenience we use the notation $X \wedge S$ in order to abbreviate $\{X\} \cup S$. In the precondition of a rule, we also require $X \notin S$.

$$\textit{(Monot.)} \quad \frac{\kappa(\tau_1,\ldots,\tau_n) \preceq^u \mu(\pi_1,\ldots,\pi_n) \wedge C}{\tau_1 \preceq^u \pi_1 \wedge \ldots \wedge \tau_n \preceq^u \pi_n \wedge C}\mathcal{D} \qquad \text{if} \quad \kappa \leq_s \mu \text{ in } (\mathbf{T}, \leq_s)$$

$$\textit{(DC)} \quad \frac{\alpha^0 \preceq^u \pi \wedge C}{C}\mathcal{D} \qquad \text{with} \quad \alpha^0 \in \mathbf{TV}^0$$

$$\textit{(NarrowY)} \quad \frac{\kappa(\pi_1,\ldots,\pi_n) \preceq^u y \wedge C}{\kappa \preceq^u y_0 \wedge \pi_1 \preceq^u y_1 \wedge \cdots \wedge \pi_n \preceq^u y_n \wedge \Theta(C)}\mathcal{D} \qquad \text{with} \quad \begin{array}{l} n \geq 1, y \in \mathbf{TV} \\ \Theta = \{y \leftarrow y_0(y_1,\ldots,y_n)\} \\ \text{each } y_i \text{ is a new variable} \end{array}$$

$$\textit{(LUB)} \quad \frac{\tau_1 \preceq^u \alpha \wedge \tau_2 \preceq^u \alpha \wedge C}{(\tau_1 \sqcup \tau_2) \preceq^u \alpha \wedge C}\mathcal{D} \qquad \text{if} \quad \tau_1 \sqcup \tau_2 \text{ exists}$$

Example 4 The set $\{list(nat) \preceq^u \alpha, list(int) \preceq^u \alpha\}$ has the principal upper matcher $\{\alpha \leftarrow list(int)\}$ which is also the p.u.m. for $\{list(\alpha^0) \preceq^u \alpha, list(int) \preceq^u \alpha\}$ (with $\alpha^0 \in \mathbf{TV}^0$). The set $\{list(\beta) \preceq^u \gamma, list(int) \preceq^u \gamma\}$ has no upper matcher if $\beta \notin \mathbf{TV}^0$.

A typing statement $VT \vdash t :_l \pi$ can be read as "If VT is a valid variable typing, then the term t *has principal type* π". Being somewhat sloppy, we also say that π is the least type of t (w.r.t. VT). Note that we are now interested in the principal type of a term, in contrast to any type of a term as used in typed lambda calculus and many other approaches for typed logic programming, that use rules similar to $(Var), (Func), (Subtype)$.

$VT \vdash t :_l \pi$ is true if and only if it is derivable using the rules *(VarL)* and *(FuncL)*.

$$\textit{(VarL)} \quad \frac{}{VT \cup \{v : \tau\} \vdash v :_l \tau}\mathcal{T}$$

$$\textit{(FuncL)} \quad \frac{VT \vdash t_1 :_l \tau_1 \quad \cdots \quad VT \vdash t_k :_l \tau_k}{VT \vdash f(t_1,\cdots,t_k) :_l \Delta(\Theta(\pi))}\mathcal{T} \quad \text{if} \quad \begin{array}{l} f : \pi_1 \times \ldots \times \pi_k \rightarrow \pi \\ \text{is a declaration variant} \\ \Theta \text{ is the principal upper} \\ \text{matcher of} \\ \{\tau_1 \preceq^u \pi_1, \ldots, \tau_k \preceq^u \pi_k\} \end{array}$$

The type substitution Δ replaces all type variables by new and distinct variables from $\mathbf{TV}^0$

Example 5 $list(\alpha_0)$ is principal type of nil. $list(nat)$ is principal type of cons(1,nil). If $VT = \{x : a\}$ and a is some type then $list(a)$ is principal type of $cons(x, nil)$.

Given the declarations $f: \ list(pair(\alpha,\alpha)) \rightarrow k(\alpha)$ and $g: \ \alpha \times k(\alpha) \rightarrow k(\alpha)$; then $g(1, f(nil)) :_l k(pos)$.

Proposition 3 *Given some valid variable typing; without parameter introducing declarations, a (value) term has a unique principal type if it has some type. In general, the principal type of a term is unique up to renaming of type variables from* $\mathbf{TV}^0$.

Proposition 4 *Given a valid variable typing* VT, π *is (a variant of) the principal type of a term* t *iff* $VT \vdash t :_l \pi$.

2.5.2 Typed clauses

For each **predicate symbol** we assume a (unique) declaration

$$p : \pi_1 \times \cdots \times \pi_n \ \triangleright C$$

with n$\geq$0 and each π_i in $\mathbf{TT}$. C is a set of simple type constraints $\tau \preceq \pi$, i.e. τ and π are type parameters or base type symbols. These constraints are interpreted as a conjunction. In principle one could allow any constraint expression over type terms in the constrain part, such as *not* $\tau \leq \pi, \pi = \pi_1 \sqcup \pi_2, \tau < \pi, C1$ *or* $C2$ and more. For ease of presentation and for simplicity in the type inference, we keep the restriction to simple type constraints. With the program syntax used in the previous section, all variables occurring in any π_i are assumed to be written with an '`@`'-prefix. Although it is possible to allow more than one declaration for a predicate symbol, for technical simplicity we require them to be unique.

A typed clause will be written as

$$VT \quad A_0 \leftarrow A_1, \cdots, A_n.$$

where all A_i are atoms, and the set VT is a variable typing.

Together with the rules $(VarL)$ and $(FuncL)$ the following rules complete the definition of well-typed expression in our type system. For convenience, the special type symbol o is used for expressions with a truth value. A type-consistent atom is written as $\overline{Atom}$. There is no separate rule for equality because we assume $=$ as a predefined predicate with declaration '$=$': $\alpha \times \alpha$.

Given a set of sets of $\preceq^u$-constraints $M = \{S_1, \ldots S_n\}$ where each S_i has principal upper matcher Θ_i, then the **simultaneous principal upper matcher** Θ of M is defined as $\Theta(\alpha) = \pi$ iff π is the most general common instance of $\{\tau_i \mid \Theta_i(\alpha) = \tau_i\}$. Note that we here use standard unification on type terms instead of some order-sorted variant thereof.

Example 6 The set $\{\{\text{nat} \preceq^u \alpha, \text{nat} \preceq^u \alpha\}, \{\text{nat} \preceq^u \alpha, \text{nat} \preceq^u \beta\}\}$ has the simultaneous principal upper matcher $\Theta = \{\alpha \leftarrow \text{nat}, \beta \leftarrow \text{nat}\}$. The set $\{\{\text{nat} \preceq^u \alpha\}, \{\text{int} \preceq^u \alpha\}\}$ has no simultaneous principal upper matcher, although there is a p.u.m. for each set separately.

$$\textit{(Atom)} \quad \frac{VT \vdash t_1 :_l \tau_1 \cdots VT \vdash t_k :_l \tau_k}{VT \vdash \overline{p(t_1, \cdots, t_k)}}\mathcal{T} \quad \text{if} \quad \begin{array}{l} p : \pi_1 \times \ldots \times \pi_k \rhd Cp \\ \text{is a declaration variant,} \\ \Theta \text{ is the simultaneous} \\ \text{principal upper matcher of} \\ \{\{\tau_1 \preceq^u \pi_1\}, \ldots, \{\tau_k \preceq^u \pi_k\}\}, \\ \Theta(Cp) \text{ is satisfiable} \end{array}$$

A Program is **weakly type-consistent** if every clause in it is weakly type-consistent. A typed clause $VT \quad A_0 \leftarrow A_1, \ldots, A_n$ is weakly type-consistent iff $VT \vdash \overline{A_0 \leftarrow A_1, \cdots, A_m}$ can be derived using *(VarL)*, *(FuncL)*, *(Atom)*, *(Conj)*, *(ClauseW)*.

$$\textit{(Conj)} \qquad \frac{VT \vdash \overline{A_1} \quad \cdots \quad VT \vdash \overline{A_m}}{VT \vdash \overline{A_1, \cdots, A_m}}\mathcal{T}$$

$$\textit{(ClauseW)} \quad \frac{VT \vdash \overline{p(t_1, \cdots, t_k)} \quad VT \vdash \overline{A_1, \ldots, A_n}}{VT \vdash \overline{p(t_1, \cdots, t_k) \leftarrow A_1, \ldots, A_n}}\mathcal{T} \quad \text{if} \quad \begin{array}{l} VT \text{ is a valid} \\ \text{variable typing} \end{array}$$

Example 7

```
pred member:  @S x list(@T) |> S =< T.

member(1,cons(1,cons(2,nil)).          % weakly type-consistent
member('c',cons(1,cons(2,nil)).        % not type-consistent
member(-1,cons(1,cons(2,nil))).        % not type-consistent
member(X,cons(X,L)).                   % (weakly) type-consistent*
member(X,cons(Y,L)) :- member(X,L).    % (weakly) type-consistent*

?- member(-1,cons(1,cons(2,nil))).     % not type-consistent
```

The type-consistent clauses marked with * are not only weakly type-consistent but also generically type-consistent, see below.

2.5.3 Conventional well-typing

With conventional interpretation of type parameters in many other type systems for logic programming a well-typed atom is defined by a rule such as *(Atom')* instead of *(Atom)*

$$\textit{(Atom')} \quad \frac{VT \vdash t_1 : \Theta(\pi_1) \ \cdots \ VT \vdash t_k : \Theta(\pi_k)}{VT \vdash \overline{p(t_1, \cdots, t_k)}} \mathcal{T} \quad \text{if} \quad \begin{array}{l} p : \pi_1 \times \ldots \times \pi_k \\ \text{is a declaration variant,} \\ \Theta \text{ is a type substitution} \end{array}$$

An interesting property of our type system is that it is backwards compatible with programs containing conventional predicate declarations. Consider a predicate declaration

```
pred p: t1 x ... x ti x ... x tn |> C.
```

(with `|> C` possibly being empty) such that the type term `ti` contains a type variable `T` without a @-prefix, an embedding transformation step produces

```
pred p: t1 x ... x ti' x ... x tn |> C, Tnew =< T.
```

where `ti'` is `ti` with a single occurrence of `T` being replaced by `@Tnew` and the type variable `Tnew` did not occur in the declaration before. The so-called 'embedded version' of a declaration is the result of the exhaustive application of the transformation step. An example for $\texttt{append}_{\texttt{old}}$ was given in the introduction.

Proposition 5 *Iff a clause is well-typed with the conventional type systems, the clause is type-consistent within our type system where predicate declarations are replaced by their embedded versions.*

2.5.4 Definitional genericity

There are several variants for the well-typing of clauses reported in the literature. E.g., Hanus defines a type system allowing for so-called lemma generation [Han92], this corresponds to our notion of weak type-consistency. In many cases it is useful to disallow this feature for clauses by imposing a so-called head-condition or definitional genericity requirement on clauses (cf. [Smo89, HT92, LR91]). If the declaration of a predicate contains type parameters, the defining clauses of that predicate shall be well-typed for all instances of the type parameters. This requirement can be expressed by the rules *(AtomG')* and *(ClauseG')*.

$$\textit{(AtomG')} \quad \frac{VT \vdash t_1 : \pi_1 \ \cdots \ VT \vdash t_k : \pi_k}{VT \vdash \overline{p(t_1, \cdots, t_k)}^{\#}} \mathcal{T} \quad \text{if} \quad \begin{array}{l} p : \pi_1 \times \ldots \times \pi_k \\ \text{is a declaration variant} \end{array}$$

$$\textit{(ClauseG')} \quad \frac{VT \vdash \overline{p(t_1, \cdots, t_k)}^{\#} \quad VT \vdash \overline{A_1, \ldots, A_n}}{VT \vdash \overline{p(t_1, \cdots, t_k) \leftarrow A_1, \ldots, A_n}} \mathcal{T} \quad \text{if} \quad \begin{array}{l} VT \text{ is a valid} \\ \text{variable typing} \end{array}$$

Note that in the precondition of the rule *(AtomG')* there is no type substitution Θ applied to π_i. The conventional typing rules ensure that type parameters occurring in the declaration of p do no get instantiated by any type substitution.

For predicates where the constraint part of the declarations contains type variable constraints only, we can adopt the requirement for genericity. We could ensure that no type substitution used in the rules for type consistency instantiates a type parameter occurring in the declaration of p by adding several additional technical requirements to the typing rules. Because this gets rather clumsy, we will simply replace these type parameters by new type symbols, thus achieving the same result.

A set of type variable constraints is a set C of constraints $\alpha \preceq \beta$ where α and β are type variables. We say that α is related to β in C iff the reflexive, transitive closure of the $\preceq$-relation given by C contains $\alpha \preceq \beta$ or $\beta \preceq \alpha$. Given a predicate declaration $p : \pi_1 \times \ldots \times \pi_k \rhd Cp$, a definitional version is defined as $\Theta(p : \pi_1 \times \ldots \times \pi_k)$ where Θ replaces all type variables by new constants $\#_i$ not occurring in the set of type symbols $\mathbf{T}$, such that $\Theta(\alpha) = \Theta(\beta)$ if type variables α and β are related in Cp, and $\Theta(\alpha) \neq \Theta(\beta)$ else. $\mathbf{T}^{\#}$ is defined as the set $\mathbf{T}$ extended by arbitrary many symbols $\#_i$ used in Θ. For all i, $\#_i \leq \pi$ is true only for $\pi = \#_i$, and $\tau \leq \#_i$ is true only for $\tau = \#_i$. A clause is called **generically type-consistent** iff type-consistency is derivable by the rules *(Var), (Func), (Subtype), (VarL), (FuncL), (Atom), (AtomG), (Conj), and (ClauseG)* using the extended type order on $\mathbf{T}^{\#}$.

(AtomG) $$\frac{VT \vdash t_1 : \pi_1 \;\cdots\; VT \vdash t_k : \pi_k}{VT \vdash \overline{p(t_1, \cdots, t_k)}^{\#}}\mathcal{T}$$ if $p : \pi_1 \times \ldots \times \pi_k$ is a definitional version of a declaration $p : \pi'_1 \times \ldots \times \pi'_k \rhd Cp$,

(ClauseG) $$\frac{VT \vdash \overline{p(t_1, \cdots, t_k)}^{\#} \quad VT \vdash \overline{A_1, \ldots, A_n}}{VT \vdash \overline{p(t_1, \cdots, t_k) \leftarrow A_1, \ldots, A_n}}\mathcal{T}$$ if VT is a valid variable typing

Proposition 6 *Iff a clause is generically well-typed with the conventional type systems, the clause is generically type-consistent within our type system where predicate declarations are replaced by their embedded versions.*

Example 8

```
pred member:  @S x list(@T) |> S =< T.

member(1,cons(1,cons(2,nil))).         % only weakly type-consistent
member(X,cons(X,L)).                   % generically type-consistent
member(X,cons(Y,L)) :- member(X,L).    % generically type-consistent
```

2.6 Useless atoms and Negation

While the presentation given so far only deals with clauses containing positive atoms, we will now extend the type checking to handle negation thereby taking a closer look to our notion of 'useless' atoms.

```
pred overlap: list(@T) x list(@U) |> V =< T, V =< U.
overlap(S1,S2) :- Lists S1 and S2 have members in common.
```

With the declaration above the atom `overlap(cons(-1,nil),cons(1,nil))` is not type-consistent. The type declaration of `overlap` says that with the intended meaning an atom `overlap(S1,S2)`, where `S1` has type `list(negint)` and `S2` has type `list(nat)`, will never be true. The types `negint` and `nat` do not have a common subtype as required by the declaration. In other words, the type declaration gives a *necessary condition* for an atom to be true.

However, a subgoal with negation such as '`not overlap(cons(-1,nil),cons(1,nil))`' succeeds. Therefore, one might call the whole subgoal type-consistent. We still find it reasonable to define such a subgoal as not type-consistent, because it can be seen by static type analysis, that the subgoal will succeed.

If the type information renders some atom either as always false or always true, we consider that atom as being **useless** and therefore not type-consistent. Although other authors only consider atoms which are always false as useless, we argue that the expanded notion is senseful. Useless atoms in some program clause (or goal) are redundant, because they can be statically detected as being logically equivalent to the constant 'false' or 'true'. In our view, an atom which is useless due to typing information, represents a programming error.

Complementary to `overlap` we could have a predicate `disjoint`:

```
disjoint(S1,S2) :- not overlap(S1,S2).
```

It is not easy to define a concise type declaration covering all atoms for `disjoint` intended to be true. Interestingly, we can give a type declaration giving a *sufficient condition* for an atom to be true. If there is an atom `disjoint(S1,S2)` where $\mathtt{S1} :_l \mathtt{list(T)}$, $\mathtt{S2} :_l \mathtt{list(U)}$, such that the constraints $\{\mathtt{V} \preceq \mathtt{T}, \mathtt{V} \preceq \mathtt{U}\}$ are not satisfiable, then the atom is always true in the intended meaning. Hence, the atom `disjoint([-1,-2],[1,2])` is not type-consistent. In order to express these kinds of type constraints syntactically in a program we allow for an additional form of predicate type declarations using the syntax `pred_neg` and `|` instead of `pred` and `|>`. For `disjoint` the declaration is

```
pred_neg disjoint: list(@T) x list(@U) | V =< T, V =< U.
```

with respect type-consistency an atom `p(...)` where `p` has a declaration marked with `pred_neg` is handled in the same way as a negated atom.

3 Type inference

The typing rules precisely define type-consistency for a program and clauses by a set of logical inference rules. If a typed clause is given together with a complete prefix VT, i.e. all local variables are declared, the problem of checking if the typed clause is type-consistent can be solved by working backwards through the typing rules. With some syntactical adaptations, the typing rules can easily be translated into Prolog clauses.

Usually a programmer doesn't want to declare local variables at all, but only write clauses without variable typings. Nevertheless, we expect predicate and function declarations to be given explicitly. The task then is to reconstruct[2] a variable typing for a clause. This can be formulated as an abductive reasoning problem: under which assumption, i.e. with which variable typing VT, do the typing rules imply that the clause is type-consistent? If we are able to show that no such VT exists, then the clause is detected as being not type-consistent.

Note that these rules are written in a reversed style when compared with typing rules. While the typing rules show sufficient conditions for type-consistency the decomposition rules implement the dual necessary conditions for type-consistency.

The type inference algorithm is divided into three major phases. As input we take an untyped clause. The first phase decomposes the clause into subproblems for principal upper matchers and other constraints over type terms. The second phase will normalize these subproblems into constraints over the quasi-lattice of type symbols $(\mathbf{T}, \leq)$. Last, the satisfiability of these basic constraints is checked.

1. Decompose the clause into
   - a set of type restrictions on variables,
   - inequations for principal upper matchers, and
   - type constraints.
2. These sets are further decomposed into
   - a set of non-emptiness conditions for type terms
   - a set of $\leq_s$-constraints over variables and type symbols
   - a set of conditions of the form $y = \sqcup S$, requiring y to be the least upper bound of the elements in S.
3. Try to find a substitution for all variables, such that these constraints are satisfied.

[2] A short note on terminology: Although the notion 'type reconstruction' might be more appropriate, we will stick to convention and say 'type inference' for building a fully typed clause out of a clause with no explicit variable typing.

Variables in the generated constraints are not necessarily type variables, in the formal sense of section 2.2, but simply represent unknown values to be found by the constraint solving procedure.

If all steps succeed, the accumulated substitutions used in intermediate steps determine a variable typing VT such that the input clause is type-consistent.

We will only give the inference rules for weak type-consistency in a brief fashion; generic type-consistency can be checked with almost the same rules if the type parameters in the definitional versions of predicate declarations are simply replaced by new base type symbols which are not comparable to any other type symbol.

3.1 Clause decomposition

Given an untyped clause $H \leftarrow B_1, \ldots, B_n$, the decomposition rules *(Cl)*, *(At)*, *(Te)*, and *(Va)* produce expressions of the form

$$Red \otimes (Mp \odot Mf) \otimes Co$$

where Red is the expression which remains to be decomposed (or reduced), Mp are the constraints for principal upper matchers derived from predicate declarations, Mf are the constraint for p.u.m. derived from functions declarations, and Co are constraints collected from predicate declarations.

The following rules are applied exhaustively until $Red \otimes M \otimes Co$ is reached, where $Red = v_1 :_l \tau_1 \wedge \ldots \wedge v_n :_l \tau_n$ with different value variables v_i. With Θ being the composition of all substitutions applied during the type inference $VT = \{v_1 : \Theta(\tau_1), \ldots, v_n : \Theta(\tau_n)\}$ is a variable typing such that the input clause is weakly type-consistent. If any steps fails, then the input clause can not be made type-consistent. In case of multiple declarations for function symbols, the rule *(Te)* is not deterministic. In a concrete implementation this nondeterminism can be handled by simple backtracking. (In the same way multiple predicate declarations could be handled.)

If not specified, new type variables are not in $\mathbf{TV}^0$.

$$(Cl) \quad \frac{H \leftarrow B_1, \cdots, B_n}{H :_l o \wedge B_1 :_l o \wedge \cdots \wedge B_n :_l o \otimes \{\} \odot \{\} \otimes \{\}}$$

$$(At) \quad \frac{p(t_1, \cdots, t_k) :_l o \wedge Red \otimes Mp \odot Mf \otimes Co}{\begin{array}{c} t_1 :_l \alpha_1 \wedge \cdots \wedge t_k :_l \alpha_k \wedge Red \otimes \\ Mp \cup \{\{\alpha_1 \preceq^u \pi_1\}, \cdots, \{\alpha_k \preceq^u \pi_k\}\} \odot Mf \otimes \\ Co \cup Cp \end{array}} \quad \text{if} \quad \begin{array}{l} p : \pi_1 \times \ldots \times \pi_k \rhd Cp \\ \text{is a declaration variant,} \\ \text{each } \alpha_i \text{ is new} \end{array}$$

$$(Te) \quad \frac{f(t_1, \cdots, t_n) :_l \tau \wedge Red \otimes Mp \odot Mf \otimes Co}{\begin{array}{c} t_1 :_l \alpha_1 \wedge \cdots \wedge t_n :_l \alpha_n \wedge Red \otimes \\ Mp \odot Mf \cup \{\{\alpha_1 \preceq^u \pi_1, \cdots, \alpha_n \preceq^u \pi_n\}\} \otimes \\ Co \cup \{\tau \doteq \Delta(\pi)\} \end{array}} \quad \text{if} \quad \begin{array}{l} f : \pi_1 \times \cdots \times \pi_n \rightarrow \pi \\ \text{is a declaration variant,} \\ \text{each } \alpha_i \text{ is new} \end{array}$$

The type substitution Δ replaces all type variables in π not occurring in any π_i by distinct type variables from $\mathbf{TV}^0$

$$(Va) \quad \frac{v :_l \tau \wedge v :_l \pi \wedge Red \otimes M \otimes Co}{v :_l \tau \wedge Red \otimes M \otimes Co \cup \{\tau \doteq \pi\}} \quad \text{if } v \text{ is a value variable}$$

Equations $\tau \doteq \pi$ are solved by standard unification. The constraints $Mp \odot Mf$ for principal upper matchers, with all accumulated substitutions applied, are decomposed by the rules in section 3.3. Then the upper matcher constraints $m =$

$(Mp \cdot Mf)$ and the constraints C, are decomposed simultaneously. Substitutions applied during the decomposition of constraints C are also applied to M, and vice versa. If not all compound type terms can be decomposed into type symbols and variables, the original clause is not type-consistent.

In order to ensure termination of the decomposition rules an appropriate check is performed in a preprocessing step similar to the algorithm 'Match' described in [FM90]. Such a process is also known as 'shape unification'. E.g., the decomposition rules do not terminate for $\{x \preceq y, x \preceq list(y)\}$. Instead the constraint problem is detected to be not satisfiable by the preprocessing step.

3.2 Type term constraints

Constraints containing compound type terms will be decomposed successively into constraints over simple type symbols and type variables.

For the sake of conciseness we will overuse syntax in the way that $x(\ldots)$ stands a type term where the functor may be a type symbol or a variable.

$$\textit{(C1)} \quad \frac{x_0(\tau_1, \ldots, \tau_k) \preceq y_0(\pi_1, \ldots, \pi_k) \wedge C}{x_0 \preceq y_0 \wedge \tau_1 \preceq \pi_1 \wedge \ldots \wedge \tau_k \preceq \pi_k \wedge C} \quad \text{with} \quad k \geq 1$$

$$\textit{(C2)} \quad \frac{x \preceq y_0(\pi_1, \ldots, \pi_k) \wedge C}{x_0 \preceq y_0 \wedge x_1 \preceq \pi_1 \wedge \cdots \wedge x_k \preceq \pi_k \wedge \Theta(C)} \quad \text{with} \quad \begin{array}{l} k \geq 1, x \text{ is a var.} \\ \Theta = \{x \leftarrow x_0(x_1, \ldots, x_k)\} \\ \text{each } x_i \text{ is a new var.} \end{array}$$

$$\textit{(C3)} \quad \frac{x_0(\pi_1, \ldots, \pi_k) \preceq y \wedge C}{x_0 \preceq y_0 \wedge \pi_1 \preceq y_1 \wedge \cdots \wedge \pi_k \preceq y_k \wedge \Theta(C)} \quad \text{with} \quad \begin{array}{l} k \geq 1, y \text{ is a var.} \\ \Theta = \{y \leftarrow y_0(y_1, \ldots, y_k)\} \\ \text{each } y_i \text{ is a new var.} \end{array}$$

3.3 Matcher

The decomposition of $\preceq^u$-constraints needs to handle type variables from $\mathbf{TV}^0$ specially. The rule *(M4f)* is used for $\preceq^u$-constraints Mf derived from function declarations, while rule *(M4p)* is used for $\preceq^u$-constraints Mp derived from predicated declarations,

$$\text{(M1)}\quad \frac{\{x_0(\tau_1,\cdots,\tau_k) \preceq^u y_0(\pi_1,\cdots,\pi_k) \wedge \Gamma\} \wedge M}{\{x_0 \preceq^u y_0 \wedge \tau_1 \preceq^u \pi_1 \wedge \cdots \wedge \tau_k \preceq^u \pi_k \wedge \Gamma\} \wedge M}$$

$$\text{(M2)}\quad \frac{\{x \preceq^u y_0(\pi_1,\cdots,\pi_k) \wedge \Gamma\} \wedge M}{\{x_0 \preceq^u y_0 \wedge x_1 \preceq^u \pi_1 \wedge \cdots \wedge x_k \preceq^u \pi_k \wedge \Theta(\Gamma)\} \wedge \Theta(M)} \quad \text{where} \quad \begin{array}{l} x \notin \mathbf{TV}^0, k \geq 1, x \in \mathbf{TV} \\ \Theta = \{x \leftarrow x_0(x_1,...,x_k)\} \\ x_i \text{ are new var.} \end{array}$$

$$\text{(M3)}\quad \frac{\{x_0(\tau_1,\cdots,\tau_k) \preceq^u y \wedge \Gamma\} \wedge M}{\{x_0 \preceq^u y_0 \wedge \tau_1 \preceq^u y_1 \wedge \cdots \wedge \tau_k \preceq^u y_k \wedge \Theta(\Gamma)\} \wedge \Theta(M)} \quad \text{where} \quad \begin{array}{l} k \geq 1, y \in \mathbf{TV} \\ \Theta = \{y \leftarrow y_0(y_1,...,y_k)\} \\ y_i \text{ are new var.} \end{array}$$

$$\text{(M4p)}\quad \frac{\{\alpha^0 \preceq^u \pi \wedge \Gamma\} \wedge M}{\{\Gamma\} \wedge M} \quad \text{where} \quad \alpha^0 \in \mathbf{TV}^0$$

$$\text{(M4fa)}\quad \frac{\{\alpha^0 \preceq^u y_0(\pi_1,\cdots,\pi_k) \wedge \Gamma\} \wedge M}{\{x_0^0 \preceq^u y_0 \wedge x_1^0 \preceq^u \pi_1 \wedge \cdots \wedge x_k^0 \preceq^u \pi_k \wedge \Gamma\} \wedge M} \quad \text{where} \quad \begin{array}{l} \alpha^0 \in \mathbf{TV}^0, \\ x_i^0 \text{ are new from } \mathbf{TV}^0 \end{array}$$

$$\text{(M4fb)}\quad \frac{\{\alpha^0 \preceq^u \beta \wedge \Gamma\} \wedge M}{\{\Gamma\} \wedge \Theta(M)} \quad \text{where} \quad \begin{array}{l} \alpha^0 \in \mathbf{TV}^0 \\ \beta \text{ does occur in } \Gamma \end{array}$$

$$\text{(M4fc)}\quad \frac{\{\alpha^0 \preceq^u \beta \wedge \Gamma\} \wedge M}{\{\Gamma\} \wedge \Theta(M)} \quad \text{where} \quad \begin{array}{l} \alpha^i \in \mathbf{TV}^0 \\ \beta \text{ does not occur in } \Gamma \\ \Theta = \{\beta \leftarrow \alpha^0\} \end{array}$$

$\preceq^u$-constraints in Mf require some additional treatment compared to $\preceq^u$-constraints in Mp, in order to compute principal types of terms correctly. For constraints in Mf variables on the right hand side of $\preceq^u$ are sometimes substituted by a variable from $\mathbf{TV}^0$, as done in rule *(M4fc)*.

Example 9 $\{\{\alpha_1^0 \preceq^u \beta, \alpha_2^0 \preceq^u \beta\}, \{posint \preceq^u \gamma, \beta \preceq^u \gamma\}\}$ is decomposed into $\{\{\}, \{posint \preceq^u \gamma, \alpha_1^0 \preceq^u \gamma\}\}$ and further into $\{\{\}, \{posint \preceq^u \gamma\}\}$. $\{\{posint \preceq^u \beta, \alpha^0 \preceq^u \beta\}, \{\beta \preceq^u \gamma\}\}$ is decomposed into $\{\{posint \preceq^u \beta\}, \{\beta \preceq^u \gamma\}\}$. Note that β is not substituted by α^0.

3.4 Finite domain constraint solving over quasi-lattices

The normalized $\preceq^u$-constraints in M and $\preceq$-constraints in C are transformed into a set C_s of constraints over the quasi-lattice $(\mathbf{T}, \leq_s)$ as follows:

$$\tau \leq \pi \ \in C_s \quad \text{if} \quad \tau \preceq \pi \ \in C$$

$$\tau \leq \pi \ \in C_s \quad \text{if} \quad \tau \preceq^u \pi \ \in \Gamma, \Gamma \in M$$
$$\sqcup(S) = y \ \in C_s \quad \text{if} \quad S = \{X \mid X \preceq^u y \ \in \Gamma, \Gamma \in M\} \text{ and } S \text{ is not empty.}$$

These transformations result in a set C_s of constraints over the finite set of type symbols $(\mathbf{T}, \leq)$. Standard constraint solving techniques can be applied (cf. [Tsa93]).

The computational complexity of solving constraints over partial ordered set is rather high in the general case. The need for solving least upper bound constraints $y = \sqcup(S)$ makes the problem even harder. Satisfiability of $\leq$-constraints in partial ordered sets, i.e. even without $\sqcup$-constraints, is NP-complete ([LM92] [PT95]). For lattices the problem is known to be in PTIME, but for quasi-lattices the problem remains NP-complete. Another computational complexity comes from overloaded function symbols. Backtracking leads to a time exponential in the number of overloaded symbols in a clause. In practice our constraint sets are not very large, because we proceed clause by clause and do not infer declarations of predicate symbols.

Proposition 7 *The type inference algorithm terminates. Iff a clause Cl, i.e. $A \leftarrow A_1, \ldots, A_n$, is type-consistent with some variable typing, then the type inference algorithm will infer a variable typing VT (s. sec. 3.1) such that Cl is type-consistent w.r.t. VT.*

4 Discussion

By writing type declarations for predicates, the programmer gives hints on the intended semantics for that predicate. Every atom intended to be true shall be well-typed with respect to the declaration. Nevertheless, our type system presented so far remains purely syntactical. The semantical problems with other type inferencing approaches, as exhibited by the second example in the introduction, do not occur within our framework. The goal `e(Y,1), q(Y)` in that example is well-typed without the need to give a new semantics to the goal. Our type checking and type inferencing does not affect the semantics of clauses by imposing new type restrictions on variables as it is done in other approaches. By separating static checking for type errors in programs from a possibly complicated or even undecidable typed inference calculus, our method has the further advantage that is does not necessarily require expensive type computations at run-time. This does not mean that we consider type or sort computations as impractical in general, but argue for advantages in separating different concerns.

While a corresponding semantics for well-typed models could be defined, e.g. using results from [Han91], we believe that it is reasonable to consider syntactical type checking for its own. Assume a programmer wants to define the absolute difference of numbers by the (erroneous) clauses

```
pred absdist: int x int x nat.
absdist(X,Y,D) :- X < Y, D is Y - X.
absdist(X,Y,D) :- X >= Y, D is Y - X.
```

Of course, the second clause should have '`D is X-Y`' instead of '`D is Y-X`'. Both clauses are well-typed under the variable typing { X:int, Y:int, D:nat}. Nevertheless, using normal Prolog resolution, the goal `absdist(5,4,D)` gives the result `D = -1`, which is not intended and, even more, does not correspond to the type declaration of the predicate `absdist`. With respect to the type system the usual reaction is to reject such an approach for typed logic programming and require the resolution and its unification to obey the type constraints on variables. In that case, i.e. using an inference calculus implementing the correct order-sorted unification as in [BM94], the goal `absdist(5,4,D)` fails. With respect to detecting the programming error in this example, the practical consequences of using or leaving out order-sorted unification are essentially the same: the program is well-typed but does not produce the intended results. Type-correct inference calculi tend to (correctly) produce a logical failure instead of explicitly reporting a type-error.

Is it reasonable to allow the logic inference calculus to produce results that do not conform to type declarations? On the positive side there are strong practical arguments: there is a type system that allows natural modeling of data structure and also enables detailed static program analysis. Other approaches to define an expressive type system and a new typed inference calculus in combination sometimes have to cope with severe undecidability problems, or they restrict the language or impose a run-time overhead that is not accepted by most programmers.

Closer to our type system are proposals using so-called implication types [Red88, PR89] or type dependencies ([KW90], see also [CK93]). An example for a type dependency is $append\langle list(T), list(T), list(T)/1,2 \rightarrow 3;\ 3 \rightarrow 1,2\rangle$. Its meaning is: For all τ, if the first two arguments of *append* have the type $list(\tau)$ then so does the third argument, and vice-versa. This type dependency has a similar effect for `append` as our declaration with type constraints, but there are other declarations, e.g. for `overlap` that are not expressible with type dependencies. I.e., using our declarations with type constraints, we are able to detect more type errors.

The special treatment of type variables from $\mathbf{TV}^0$ would not be necessary if there were a least type $\perp$ with $\perp \leq \tau$ for any type term τ (cf. [Smo89]). However, besides technical problems there would also be deficiencies in the ability to detect programming errors, e.g.

```
pred my_adjoin: list(int) x list(list(int)) x list(list(int)).
my_adjoin(X,Y,Z) :- append(Y,X,Z).
```

would be well-typed with {`X : list(⊥), ...`} although the subgoal should read `append(Y,[X],Z)`.

5 Conclusion and outlook

We have shown anomalies in generally used type systems, where subtyping tends to undermine the benefits of parametric predicates. The problems become the more severe the higher the type hierarchies are. Using a new kind of type declarations with subtypes and parametric polymorphism, we have presented a solution to these anomalies. The advantages are that refined declarations for predicates can be given. Automatic type checking can detect erroneous clauses even if the type hierarchy is high or contains a most general type. With the new approach, more type errors can be detected. It is an orthogonal extension to conventional type systems: Predicate declarations given in the conventional form can be translated into the new form automatically. In fact, they can be seen as abbreviations of the new more refined form, although in most cases we found it appropriate to give a more refined declaration with constraints. As the introductory analysis shows, the conventional declarations should be considered as too simple if parametric polymorphism is combined with subtyping.

Furthermore, our new typing approach is independent of any mode system for specifying an input/output behavior for predicates. Also, it does not impose a functional or directional view on logic programs as it is done with 'directional types' and variants thereof (see e.g. [BLR92, RNP92, AL94, AE93, BM95]). It doesn't matter if the inference calculus is top-down as in Prolog or bottom-up as it can be in a deductive database system.

An interesting application of our new typing approach is type checking annotated Prolog programs where the predicate declarations are given by the programmer. These declarations are assumed to describe the semantics of the predicate in the intended model. The declarations are not taken prescriptively, because plain Prolog does not contain any kind of order-sorted unification. Nevertheless, the types of variables are then automatically inferred. If for a given clause no variable typing can be inferred which makes the clause well-typed, then the clause is considered ill-typed. We also have developed a tool for automatic type checking and inferencing annotated Prolog programs using an optimized version of the type inferencing algorithm presented in section 3. A very short notation is possible for many declarations. E.g, the declaration for 'append' becomes `append(list(T),list(T),list(@T))` and 'member' is declared as `member(T,list(@T))`. These forms are internally translated into the corresponding declarations with explicit type constraints as used in this paper. While meta-programming is handled with a built-in type `goal` in the current version of our type checking tool, we are considering more detailed type declarations for higher-order predicates, e.g. a generic sort predicate could have a declaration like `pred sort: list(@Key) x pred(@K,@K) x list(@Key) |> Key =< K`. We also want to apply the type system to more general logic systems, such as feature logics and deduction systems using non-Horn clauses.

References

[AE93] Krzysztof R. Apt and Sandro Etalle. On the unification free prolog programs. CS-R9331, CWI Amsterdam, 1993. also in MFCS'93.

[AL94] Alexander Aiken and T. K. Lakshman. Directional type checking of logic programs. In *Static Analysis Symposium*, LNCS. Springer Verlag, 1994.

[Bar92] H. P. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Clarendon Press, Oxford, 1992.

[Bei92] Christoph Beierle. Logic programming with typed unification and its realization on an abstract machine. *IBM Journal of Research and Development*, 36(3):375–390, May 1992.

[Bei95] Christoph Beierle. Type inferencing for polymorphic order-sorted logic programs. In L. Sterling, editor, *Proc. of the 12th Int. Conf. on Logic Programming*, Tokyo, June 1995.

[BLR92] F. Bronsard, T.K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. R. Apt, editor, *Logic Programming: Proceedings of 1992 Joint International Conference and Symposium*, pages 321–335. The MIT Press, 1992.

[BM94] Christoph Beierle and Gregor Meyer. Run-time type computations in the Warren Abstract Machine. *The Journal of Logic Programming*, 18(2):123–148, February 1994.

[BM95] Johan Boye and Jan Małuszyńsky. Two aspects of directional types. In L. Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, pages 747–761, Tokio, 1995. The MIT Press.

[CK93] Weidong Chen and Michael Kifer. Polymorphic types in higher-order logic programming. TR 93/20, SUNY, December 1993.

[CW86] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, December 1986.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, January 1982.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.

[Han89] Michael Hanus. Polymorphic higher-order programming in Prolog. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conf. on Logic Programming*, pages 382–397, Lisbon, Portugal, 1989. The MIT Press.

[Han91] Michael Hanus. Parametric order-sorted types in logic programming. In *Proc. TAPSOFT'91*, volume 494 of *Lecture Notes in Computer Science*, pages 181–200, Brighton, April 1991. Springer Verlag.

[Han92] Michael Hanus. Logic programming with type specifications. In Pfenning [Pfe92], chapter 3, pages 91–140.

[HL94] Patricia Hill and John Lloyd. *The Gödel Programming Language.* Logic programming series. The MIT Press, 1994.

[HT92] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In Pfenning [Pfe92], chapter 1, pages 1–62.

[KW90] Michael Kifer and James Wu. A first-order theory of types and polymorphism in logic programming. Techn.Rep. 90/23, SUNY, New York, July 1990.

[Lak91] T. K. Lakshman. Typed prolog. ftp://a.cs.uiuc.edu/pub/reddy/typed-prolog/, 1991.

[LM92] Patrick Lincoln and John C. Mitchell. Algorithmic aspects of type inference with subtypes. In *Proc. ACM Symp. on Principles of Programming Languages*, 1992.

[LR91] T. K. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Saraswat and Ueda, editors, *Int. Symp. on Logic Programming, ILPS 91*, pages 202–217, San Diego, 1991.

[Mey96] Gregor Meyer. On the use of types in logic programming. Informatik Berichte 199, FernUniversität Hagen, June 1996.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mis84] Prateek Mishra. Towards a theory of types in Prolog. In *Proceedings of the 1984 Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, 1984.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[MO84] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[O'K84] Richard O'Keefe. Prolog type-checker. typecheck.pl in ftp://doc.ic.ac.uk/packages/prolog-pd-software/tools.tar.Z, 1984.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming.* Logic Programming Series. The MIT Press, 1992.

[PR89] Changwoo Pyo and Uday S. Reddy. Inference of polymorphic types for logic programs. In E. Lusk and R. Overbeck, editors, *Logic Programming, Proc. of the North American Conference*, pages 1115–1132, 1989.

[PT95] V. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. Tr 95-15(215), Institute of Informatics, Warsaw University, 1995. na.

[Red88] Uday S. Reddy. Notions of polymorphism for predicate logic programs. In *Int. Conf. on Logic Programming*, pages (addendum, distributed at conference) 17–34. The MIT Press, August 1988.

[RNP92] Yann Rouzaud and Lan Nguyen-Phuong. Integrating modes and subtypes into a prolog type-checker. In K. Apt, editor, *Logic Programming, Proc. of JICSLP*, pages 85–97. Cambridge, Mass., 1992.

[Smo88] Gert Smolka. TEL (Version 0.9) Report and User Manual. SEKI Report SR-87-11, University of Kaiserslautern, February 1988.

[Smo89] Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* PhD thesis, Universität Kaiserslautern, Germany, 1989.

[Som95] Zoltan Somogyi. The Mercury project. http://www.cs.mu.oz.au/~zs/mercury.html, 1995.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, 1993.

[YFS92] Eyal Yardeni, Thom Frühwirth, and Ehud Shapiro. Polymorphically typed logic programs. In Pfenning [Pfe92], chapter 2, pages 63–90.