Using Types as Approximations for Type Checking Prolog Programs

Christoph Beierle[1] and Gregor Meyer[2]

[1] FernUniversität Hagen, FB Informatik,
58084 Hagen, Germany
beierle@fernuni-hagen.de
[2] IBM Germany, SWSD,
71003 Böblingen, Germany
grmeyer@de.ibm.com

Abstract. Subtyping tends to undermine the effects of parametric polymorphism as far as the static detection of type errors is concerned. Starting with this observation we present a new approach for type checking logic programs to overcome these difficulties. The two basic ideas are, first, to interpret a predicate type declaration as an approximation for the success set of the predicate. Second, declarations are extended with type constraints such that they can be more refined than in other conventional type systems. The type system has been implemented in a system called Typical which provides a type checker for Standard Prolog enriched with type annotations.

1 Introduction

There are quite a few approaches to typed logic programming. Several type systems support parametric polymorphism and subtypes e.g. [28,12,8], see also the collection in [24]. In [18] a classification scheme for the various uses of types in logic programming is developed. It distinguishes three almost independent dimensions of using types in logic programming: *types for proving partial correctness*, *types as constraints*, and *types as approximations* used in consistency annotations. Another aspect for the comparison of typed logic languages is how the semantics of typed predicates is defined, depending either on the clauses *and* the type declarations (called *prescriptive typing* [15]) or independent from type declarations (*descriptive typing*).

While there are many motivations for introducing types (naturalness of the representation, efficiency when using types as active constraints, etc.) the software engineering point of view seems to be the most important one: The aim is to detect as many programming errors as possible by static program analysis before running the program. In this paper, we argue that in logic programming subtyping tends to undermine the effects of parametric polymorphism as far as the (static) detection of type errors is concerned. To overcome these difficulties we use powerful type constraints in predicate type declarations which are interpreted as approximations of the intended model. Here, we present an overview of the Typical system in which these ideas have been implemented.

In Sec. 2 we motivate our approach by showing shortcomings of polymorphic type systems with subtypes. In Sec. 3 we tailor the "types as approximations" dimension of [18] towards Prolog clauses. Sec. 4 describes the Typical system and shows how predicate declarations with constraints are used as approximations of the intended model. Typical has been applied successfully to various programs, including its own source code in Standard Prolog enriched with type annotations. In Sec. 5 we argue why the descriptive approach is useful for Prolog type checking. Finally, we give some conclusions and point out further work.

2 Problems of Polymorphic Type Systems with Subtypes

An ML-like type system for logic programming was proposed and used by Mycroft and O'Keefe [21]. It includes explicit type declarations and parametric polymorphism but no subtypes. Most prominently the languages Gödel [11] and Mercury [29] 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'.

There are many different proposals for combining a logical programming language with a type system comprising parametric polymorphisms as well as subtyping. Smolka uses type rewriting [28], partial order on type symbols is used by [3,12], Hanus proposes more general equational type specifications [10] and also Horn clauses for the subtype relation [9]. Naish uses Prolog clauses to define polymorphic predicates [22]; the predicate type is specified by some general constraint expression in [13], and so on. However, these approaches have a serious shortcoming when it comes to detect obviously ill-typed expressions involving subtypes.

Example 1.

```
:- type male --> peter; paul.     %      person
:- type female --> anne; mary.    %     /      \
:- type person.                   %  female   male
:- subtype male < person.         %
:- subtype female < person.       %

:- pred father(male,person).
     father(peter,paul).
:- pred mother(female,person).
     mother(anne,peter).
     mother(mary,paul).
```

The program defines a small type hierarchy with type `person` having (disjoint) subtypes `female` and `male`. By and large we follow the syntactical style used in [21]. Function symbols and their argument types are given by enumeration. Predicate declarations define the expected types of arguments. If we add

```
:- pred q1(person).
     q1(X) :- father(X,Y), mother(X,Z).
```

the clause for `q1` can be detected as ill-typed. The type constraints for the variable `X`, i.e., `X:male` and `X:female` are not simultaneously satisfiable. However, using the common parametric type declaration for equality, i.e., `'=': T x T`, the clause for `q2` in

```
:- pred q2(person).
       q2(X) :- father(X,Y), X = Xm, mother(Xm,Z).
```

is usually not detected as ill-typed (see e.g. [28,12]) although it is logically equivalent to `q1`! If the type parameter `T` in the declaration `'=': T x T` is substituted by `person`, then `X = Xm` is not ill-typed, because the variable `X` has type `male` which is a subtype of `person`, and the same applies to the type `female` of the variable `Xm`.

Note that the problem illustrated here does not depend on the equality predicate; as we will show in the next sections similar problems occur with many often-used polymorphic predicates like `append`, `member` etc.

Subtyping tends to undermine the effects of parametric polymorphism in the conventional approaches as far as the detection of type errors is concerned. This anomaly seems to be generally neglected in the literature; [30] is an exception mentioning the weakness in type-checking, which is caused by the generally used method for combining subtypes and parametric polymorphism. We will present a new type system which enables static type checking and type inferencing to spot such errors.

Logic programming in general has no modes for input/output. One way to attack the difficulties for type systems is to restrict logic programming towards a functional or directional style with fixed modes and then apply ideas known from typed functional programing (c.f. Sec. 6). Our approach instead is to extend the type system and make it suitable for general logic programming.

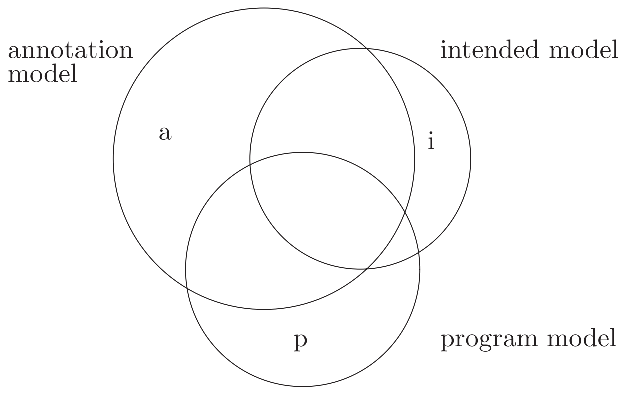
3 Types as Approximations

In this section, by tailoring the "types as approximations" dimension of [18] towards Prolog clauses, we develop a general method of static program analysis for finding programming errors like the ones illustrated in the examples above. We interpret predicate declarations as consistency annotations and take these annotations as approximations of a set of atoms intended to be true. We will discuss the applicability of consistency annotations and show how they can reasonably be used to find erroneous expressions. Specific instances of the general scheme we present here can be found as part of many proposed type systems (e.g. [22]), although mostly it is used only indirectly.

3.1 Consistency Annotations

For the beginning we will allow a rather general form of predicate declarations. For each predicate `p` we assume that there is a function tc_p which generates an appropriate constraint over some theory. The function tc_p is directly or indirectly defined by the type declaration for the predicate `p`. Given a syntactically well-formed atom $A = \mathtt{p}(\ldots)$, then $tc_p(A)$ yields a constraint which is wanted to be satisfiable, otherwise A is called ill-typed.

Intuitively, the declaration is an approximation of the set of atoms $p(\ldots)$ which should be true. I.e., a model intended by the programmer is described in two ways: first by the logical clauses and second by the predicate type declarations. Of course, in practice the predicate declarations are much simpler than the clauses and they only roughly approximate the intended meaning of a predicate. Thus, for any program we can distinguish the following three models whose possible interrelationships are illustrated as in the following diagram:



p) The program clauses alone define some model shown as region p; this model is of central importance. Based on the formal semantics of the programming language in general this model exactly represents the meaning of the program, i.e., which facts are true and which are false.

a) The predicate types also define a model that may be different from the program model. With 'types as approximations' the model of the predicate declarations, or annotations in general, should be a superset of the model of the program clauses.

i) Last but not least, the intended model, i.e., what the program should compute as required by the application domain, is just another model. In an ideal program the program model should be the same as the intended model.

We do not use the annotation model **a** in a specification of a program. This differs from the equation "Specification = Program + Types" in [22]. A consequence of the approach in [22] would be to further include modes and other implicit or explicit assumptions in the specification for a program.

Let us now discuss various cases where the model of the program clauses **p** coincides or differs from the other models:

$\mathbf{p} = \mathbf{i}$) In the optimal case the program model coincides with the intended model. I.e., the set of inferred solutions is exactly the set wanted by the programmer. Formal annotations describe some superset of the intended and inferred model.

$\mathbf{a} \setminus \mathbf{p} \neq \emptyset$) There is no problem if the model of annotations **a** is a strict superset of the program model **p**. Annotations are not required and are not even intended to describe the model as exactly as the program clauses. They can only provide an approximation as far as it can be described using the type language alone.

$\mathbf{p} \setminus \mathbf{a} \neq \emptyset$) A program may happen to have solutions which are not consistent with the type annotations, i.e., these solutions are ill-typed and they are marked as being errors. Such an error may also be due to an inappropriate type declaration. If an inconsistency between a program solution and the annotations is detected, it is not possible to decide automatically whether the program clauses or the annotations are not correct; this decision depends on the intended model.

$\mathbf{i} \setminus \mathbf{p} \neq \emptyset$) If a solution in the intended model is not computed by the program then this corresponds to an error because an intended answer is missing.

$\mathbf{p} \setminus \mathbf{i} \neq \emptyset$) The difference of the program model and the model intended by the programmer marks inferred solutions which are 'not wanted' by the programmer. E.g., '`append([],3,3)`' is true w.r.t. the usual untyped implementation of `append`.

Of course, if we do not have a formal description of the intended model **i**, we do not have a chance to automatically analyze the cases involving **i**. Since in this paper we do not want to deal with formal program specifications other than the discussed annotations we will therefore use the annotations as a specification of a superset of the intended model and assume $\mathbf{i} \subseteq \mathbf{a}$. As a consequence, the declarations can be used for static type checking purposes in the following way: each program clause is inspected statically if it contains expressions, possibly the whole clause, that do not fit with the semantics given by the predicate declarations. If the checking procedure finds that some expression is inconsistent with the declaration then probably a programming error is detected. In all cases the semantics of the clauses remain unaffected and well-defined. This is similar to the detection of redundant code, e.g., unreachable statements in a procedural program, which is done by many state-of-the-art compilers.

3.2 Inconsistent Atoms

For any atom $A = p(t_1, \cdots, t_n)$ the type-constraint is given by $tc_p(A)$. For simplicity we often write $tc(A)$. If there is no type declaration for p in the type part, by default we take the type constraint to be *true*.

Clauses that conflict with the predicate declaration will be called *type-inconsistent* or *ill-typed*. We argue that such clauses are useless in the program because they contain subexpressions which are not satisfiable in the intended model. A sound but not necessarily complete algorithm for detecting ill-typed clauses will point out clauses which conflict with the type declaration of the head atom, or which can be eliminated without affecting the semantics of the specification. These two cases can be illustrated by the following specification:

```
:- pred p(male).          % type declaration
     p(peter) :- p(1).    % body is always false
     p(2).                % conflict with the type declaration
```

In the intended semantics implied by the type declarations, the body of the first clause is not true. If the body of a clause is known to be unsatisfiable due to its inconsistency with the type declarations, then such a clause can be called useless: it is logically redundant. Also the second clause is inconsistent. Usually there is no reason for a programmer to write such a clause having an unsatisfiable type constraint, in this case `2:male`.

For every atom A that has an instance which is true in the intended model, we assume that $tc(A)$ is satisfiable. This is a very important assumption, because it gives the programmer a device to describe properties of the intended model. If for some atom A we can show that the type constraint $tc(A)$ is not satisfiable, then we have found an atom that has no instance in the intended model. This simple correspondence is the basis for an automated type checking method where unsatisfiable type constraints indicate programming errors.

3.3 Consistency Checks for Clauses

In a program we could translate each clause into a formula where every atom A is replaced by the type constraint $tc(A)$. If the transformed formula is not satisfiable, we have shown, using our basic assumption, that the original clause is not satisfiable in the intended model of the program. Hence, the original clause probably contains an error (at least, it is inconsistent with the annotation model). As we will show in the following, in practice we can make a more detailed analysis of program clauses exploiting predicate declarations.

Horn Clauses: First consider a program which is given by Horn clauses. If there is a simple fact $\mathtt{p}(\ldots)$ and $tc_p(\mathtt{p}(\ldots))$ is not satisfiable, then the fact contradicts with the intended model as specified by the predicate declaration. Hence this fact can be marked as erroneous, it contains a programming error with respect to the predicate declaration. A similar line of reasoning applies to clauses

$$A \leftarrow B_1, \ldots, B_n.$$

where A and B_i are atoms. If the conjunction of type constraints $tc(B_1) \wedge \ldots \wedge tc(B_n)$ is not satisfiable, we know that the body of the rule is not satisfiable (in the intended model). Formally, the complete clause together with the type constraints is a tautology, but practically we can say that the clause is useless. I.e., if such a clause appears in a logic program, this clause can be marked as containing a type error. A similar view is generally taken in type inferencing frameworks for Prolog, starting with [19]. Furthermore, it is reasonable to require the type constraint

$$tc(A) \wedge tc(B_1) \wedge \ldots \wedge tc(B_n)$$

to be satisfiable. Otherwise the body of the rule would imply an atom A that contradicts the consistency requirement as given by its declaration.

Clauses with Negation: In the general scheme we develop here we want to be independent of specific semantics for negation. We assume that for an atom C

such that a model does not contain any instance of it, with any reasonable semantics of negation `not` C is true in the model. If in the extended Horn clause

$$A \leftarrow B_1, \ldots, B_n, \texttt{not}\ C$$

$tc(C)$ is not satisfiable then `not` C will thus be true in the intended model, and therefore this subexpression can be seen as practically useless. Also, if $tc(B_1) \wedge \ldots \wedge tc(B_n) \wedge tc(C)$ is not satisfiable, then we either know that the conjunction $B_1, \ldots, B_n$ always fails or `not` C is always true when the conjunction $B_1, \ldots, B_n$ succeeds. In both cases we can argue that the body of the rule contains a programming error. As before, we will also require the type constraint of the head atom to be satisfiable simultaneously. I.e., if the type constraint expression

$$tc(A) \wedge tc(B_1) \wedge \ldots \wedge tc(B_n) \wedge tc(C)$$

is not satisfiable, we argue that the clause contains a type error. If there is more than one negated atom in an extended Horn clause, i.e. we have

$$A \leftarrow B_1, \ldots, B_n, \texttt{not}\ C_1, \ldots, \texttt{not}\ C_k$$

we require

$$tc(A) \wedge tc(B_1) \wedge \ldots \wedge tc(B_n) \wedge tc(C_i)$$

to be satisfiable for each atom C_i. We will take a closer look at the following rule, referring to Example 1:

```
:- pred p(person).
     p(P) :- not mother(P,X), not father(P,Y).
```

Intuitively, `p` is true for persons that are neither mother nor father of someone. If we required the variable `P` to be of type `female`, due to its occurrence in `mother(P,X)`, and also to be of type `male`, due to its occurrence in `father(P,Y)`, then these constraints would not be simultaneously satisfiable, because the types `female` and `male` are disjoint. Instead, with our condition the rule for `p` has no type error. It is interesting to note that there are other proposals such as [12], which view this clause as not being well-typed. The reason is that the variable `P` is required to have a unique type such that all atoms are well-typed, including both negated atoms. We think that this requirement is appropriate for pure Horn clauses but that in general it is too strong for predicate logic formulas or their variants with negation as failure as the given example illustrates.

4 The Typical System

The aim of Typical is to do static type checking on logic programs. The software will check Standard Prolog programs that are extended with type declarations. The type system includes subtyping and parametric polymorphism. In addition to the usual Prolog clauses, Typical expects type definitions and predicate declarations. No type declarations for variables are needed; variable types are inferred automatically by Typical.

4.1 The Type Language

Here we give an overview on the language for defining types in Typical. The basic form of monomorphic type definitions are the same as already used in Example 1. If a function symbol has arguments, then the corresponding types must be given in the definition

```
:- type machine --> fastm(int,string); slowm(int,string).
```

Parametric types are defined by using type variables and they can be ordered in the same way as other types. However, the parameters of parametric types that have a subtype relation must be the same. E.g., a complete type definition for the ubiquitous list and for binary trees modelled as a subtype of trees in general could look like

```
:- type list(T) --> [] ; [ T | list(T) ].
:- type bintree(T) --> leaf(T) ; bnode(T,bintree(T),bintree(T)).
:- type tree(T) --> node(T,list(tree(T))).
:- subtype bintree(T) < tree(T).
```

Note that infix notation for operators and types mix well.

There are various technical conditions the defined type hierarchy must fulfill, e.g. the existence of greatest lower bounds for non-disjoint types required for the existence of principal types as needed in Sec. 4.3. For technical simplicity we assume that each function symbol has a unique declared type. Subtype relationships must be given explicitly, we do not automatically detect subtyping between types if the set of terms in one type is a subset of the terms in another type. A precise description of these conditions is given in [17].

4.2 Predicate Declarations

Predicate declarations specify the types which are expected for the actual arguments when a predicated is called. E.g., if the first argument of a predicate `sumlist` must be a list of integers and the second argument must be an integer, the declaration is

```
:- pred sumlist(list(int), int)).
```

Declarations may also contain (implicitly existentially quantified) type parameters, written as Prolog variables, e.g., for `append`

```
:- pred append(list(T), list(T), list(T)).
```

More specific type declarations are possible by using *type constraints* over type variables occurring in the declaration. In order to identify this new form of declarations syntactically, we prefix type parameters within formal argument types with '`@`' [16]:

```
:- pred sublist(list(@T1), list(@T2)) |> T1 =< T2.
```

The expressions on the right of `|>` describe type constraints where `=<` stands for the subtype relationship. Syntactically similar type declarations have been used independently for type dependencies in [7] and in [23].

4.3 Approximations and Type Consistency

We first illustrate the use of the new form of type declarations for predicates by means of an example. Intuitively, in Typical a type declaration for a predicate describes a superset of the predicate solutions (cf. Section 3). E.g., the conventional declaration

$$\texttt{:- pred append}_{\texttt{old}}\texttt{(list(T), list(T), list(T)).}$$

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 anomalies as pointed out in Section 2. As part of our solution we allow for more exact declarations using type constraints:

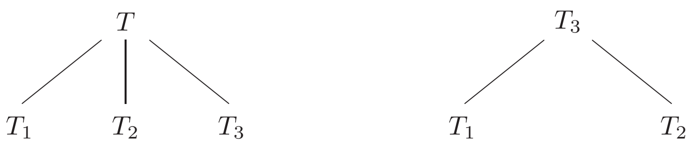
$$\begin{array}{l}\texttt{:- pred append}_{\texttt{new}}\texttt{(list(@T1), list(@T2), list(@T3))}\\ \qquad\qquad\qquad\qquad\qquad\qquad\texttt{|> T1 =< T3, T2 =< T3.}\end{array}$$

Now an atom `append(L1,L2,L3)`, where the arguments `L1`, `L2`, and `L3` have the *principal* (or *least*) *types* `list(A)`, `list(B)`, and `list(C)` respectively, is well-typed (also called type consistent) if the conjunction of type constraints `A =< C, B =< C` is satisfied.

We can easily transform the declaration for $\texttt{append}_{\texttt{old}}$ into an equivalent one using the new framework, yielding

$$\begin{array}{l}\texttt{:- pred append}_{\texttt{old}}\texttt{(list(@T1), list(@T2), list(@T3))}\\ \qquad\qquad\qquad\qquad\qquad\qquad\texttt{|> T1 =< T, T2 =< T, T3 =< T.}\end{array}$$

which is obviously weaker than the $\texttt{append}_{\texttt{new}}$ declaration. (Note that type variables occurring only in the constraint part of a predicate declaration - like `T` here - are implicitly existentially quantified.) The following figure illustrates the type constraints imposed by $\texttt{append}_{\texttt{old}}$ and $\texttt{append}_{\texttt{new}}$, respectively:



Now consider the $\texttt{append}_{\texttt{new}}$ declaration. Since predicate declarations are seen as annotations that approximate the intended semantics of the program, the atom `append([1],[-1],L)` is well-typed with the variable `L` having the (least) type `list(int)`. Given the least type of the first argument `[1]` as `list(nat)` and the least type of `[-1]` as `list(negint)`, the type constraints `nat =< int` and `negint =< int` are satisfied.

On the other hand, if the variable `L` is constrained to the type `list(nat)` the same atom is not well-typed, because there is no type `T2'` such that `[-1]` has (least) type `list(T2')` and also `T2' =< nat`. This indicates that the atom as a goal literal will always fail, because with the intended meaning there is no list of natural numbers that contains a negative number.

Similarly, the atom `append(L2,[-1],[1])` is not well-typed under the declaration for $\texttt{append}_{\texttt{new}}$ although it is considered well-typed w.r.t. the conventional typing for $\texttt{append}_{\texttt{old}}$.

4.4 Principal Types

One of the central notions within the Typical type system is the *principal type* of a term, which is the most specific type of a term. A principal type π of the term t has the property: t has type π (denoted as $t : \pi$) and if t has type τ then there is a type substitution Θ such that $\Theta(\pi) \leq \tau$, i.e.,

$$\pi \text{ is principal type of } t \quad \Leftrightarrow \quad t : \pi \text{ and } (\forall \tau)\ (t : \tau \Rightarrow (\exists \Theta)\ \Theta(\pi) \leq \tau).$$

The principal type is as minimal as possible with respect to the type order, but it is also as polymorphic as possible.

Example 2. Given the type declaration

```
   :- type pair(T, S) --> mkpair(T, S).
```

and usual declarations for `int` and `list` then the term `mkpair([1],[])` has type `pair(list(int),list(int))`, as well as `pair(list(nat),list(nat))`, `pair(list(nat),list(T))`, etc. The latter is the principal type of the term.

The notion of principal types is well-known from typed functional programming. Our definition is somewhat different because in our framework we don't have λ-abstraction and the principal type does not depend on type constraints.[1]

The syntactical appearance of type constraints in declarations is similar to declarations in functional programming with parametric types and subsumption [20,6], commonly known as $F_{\leq}$. However, their impact within Typical is rather different. As an important factor we will see that a type parameter with an `@`-prefix matches with the principal type of an argument term. Therefore, a declaration for $\texttt{append}_{\texttt{new}}$ should *not* be seen as a simple 'logical variant' of functional declarations such as

```
 func app: list(T1)×list(T2)×list(T3) → bool with T1≤T3, T2≤T3.
 func app: list(T1)×list(T2) → list(T3) with T1≤T3, T2≤T3.
```

because of the following observations: The first function declaration is similar to $\texttt{append}_{\texttt{old}}$ because the expression `app([1],[-1],L)` is well-typed even if `L` has type `list(nat)`. The type parameter `T3` can be instantiated with `int`. The second function declaration is closer to the declaration $\texttt{append}_{\texttt{new}}$ in Typical. An equation `L = app([1],[-1])` is detected as ill-typed when `L` has type `list(nat)`. However, this requires a fixed partitioning of input and output arguments which is not appropriate in logic programming.

[1] A similar definition of principal types is used in [12]. Neglecting technical problems in [12] (see [2]) our approach is rather different in the way how principal types are used for checks of type consistency.

4.5 Procedure for Type Consistency Checks

In general, a predicate declaration has the form

$$p : \text{pattern}_1 \ldots \text{pattern}_n \rhd \text{Constraint}.$$

Given some atom $p(\ldots)$, for each argument we will determine the most specific instance of each pattern$_i$ that matches the type of the corresponding argument and the constraint part is checked for satisfiability. Thus, in order 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 (where the whole procedure fails if any of the steps fails):

1. compute the principal type τ_i of every argument t_i,
2. for each τ_i determine the 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.

The first three steps implement the abstract function tc_p as used in Section 3.1. As usual, a clause is type-consistent if every atom in it is type-consistent w.r.t. the same variable typing.

Example 3. The steps to compute type consistency are illustrated by checking the atom `append([],[-1],[1])` with respect to the declaration

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

In the first step the principal types for the argument terms are computed. The empty list `[]` has the principal type $\tau_1 = \texttt{list}(\alpha)$, For `[-1]` and `[1]` the principal types are $\tau_2 = \texttt{list}(\texttt{negint})$ and $\tau_3 = \texttt{list}(\texttt{nat})$, respectively. If there was any argument term which is not type correct, e.g., `[1|2]` does not conform to the declaration of `[_|_]` because `2` is not a list, then the atom would not be type consistent.

Second, the principal types τ_i are matched against the formal types π_i in the predicate declaration. The least instances $\Theta_i(\pi_i)$ are given by $\Theta_1 = \{\texttt{T1} \leftarrow \alpha\}$, $\Theta_2 = \{\texttt{T2} \leftarrow \texttt{negint}\}$, and $\Theta_3 = \{\texttt{T3} \leftarrow \texttt{nat}\}$. If there is no least instance of a formal type π_i, e.g., if an integer occurs where a list is expected, then the atom would not be type consistent.

Third, all substitutions are combined into a single substitution $\Theta = \{\texttt{T1} \leftarrow \alpha, \texttt{T2} \leftarrow \texttt{negint}, \texttt{T3} \leftarrow \texttt{nat}\}$. In case of conflicts between the single substitutions Θ_i, e.g., if Θ_3 was $\{\texttt{T2} \leftarrow \texttt{nat}\}$ then the atom would not be type consistent.

In the last step we determine that the constraint set $\Theta(\{\texttt{T1} \leq \texttt{T3}, \texttt{T2} \leq \texttt{T3}\}) = \{\alpha \leq \texttt{nat}, \texttt{negint} \leq \texttt{nat}\}$ is not satisfiable. Hence, `append([],[-1],[1])` is not type consistent. For the atom `append([],[1],[1])`, however, we would get the set of constraints $\{\alpha \leq \texttt{nat}, \texttt{nat} \leq \texttt{nat}\}$ which is satisfiable with the substitution $\{\alpha \leftarrow \texttt{posint}\}$, i.e., the modified atom is type consistent.

Consider the Typical type declaration

```
:- pred '=': @T x @T.
```

With this declaration for '=' the type error in the clause (from Section 2)

```
    q2(X) :- father(X,Y), X = Xm, mother(Xm,Z).
```

is detected since the atom `X = Xm` with `X` of type `male` and `Xm` of type `female` is illtyped.

Example 4. Using the declarations

```
    :- pred abs(int, nat).
    :- pred member(@T1, list(@T2)) |> T1 =< T2.
```

the expression

```
    member(X,[-1,-2]), abs(Z,X)
```

is found to be not type consistent: The principal type of `X` in the first subgoal is inferred to be `negint` which is incompatible with the type `nat` required for `X` in the second subgoal.

Further examples of type declarations in Typical are:

```
:- pred reverse(list(@T), list(@T)).
% principal types of arguments must be identical
:- pred intersection(list(@T1),list(@T2),list(@T)) |> T=<T1, T=<T2.
% intersection(L1,L2,L) :- elements of L are in both L1 and L2.
:- pred overlap(list(@T), list(@U)) |> V =< T, V =<U.
% overlap(L1,L2) :- lists L1 and L2 have members in common.
```

Our notion of ill-typing does not preclude standard Prolog idioms, such as the failure driven loop. For instance, consider

```
    q :- p(X), side_effect(X), fail.
    q :- succeed.
```

where we assume that `p` and `side_effect` constrain their arguments to be of the same type and the type constraints of the nullary predicates are always true. Although in a purely declarative setting the body of the first clause is not satisfiable (assuming `fail` to be always false), i.e. the whole clause is trivially true, it is not rejected as ill-typed because we use the constant *true* as type constraint for the atom `fail`.

In [17] typing rules are given that precisely define well-typedness for a program and clauses by a set of logical inference rules. A complete type inferencing algorithm together with a description of the involved (finite domain) constraint solving over a partially ordered set of type symbols is also given in [17].

5 Why Syntactical Type Checking is Useful for Prolog

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. While a corresponding semantics for well-typed models could be defined, e.g. using results from [9], 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, int, 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 e.g. [3], 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 it does not produce the intended results. Type-correct inference calculi tend to (correctly) produce a logical failure instead of reporting a type-error. Here a difference between our approach and [22] becomes apparent. While [22] uses checks for type consistency to find clauses which would produce a type incorrect answer, we use type consistency also to find clauses which produce no answer at all.

Thus, rephrasing Millner’s slogan *Well-typed [functional] programs don’t go wrong* yields *Well-typed logic programs fail instead of going wrong*. On the other hand, for our types-as-approximations approach whose purpose is to detect useless atoms and clauses we get the slogan *Type inconsistent clauses don’t succeed [in the intended model]*.

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: We have a type system that allows natural modeling of data structures and also enables detailed static program analysis proceeding incrementally clause by clause. At the same time the runtime execution of the program can still be done by any (efficient, commercially available) Prolog system that does not have to provide any form of typed unification. 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.

6 Conclusions and Further Work

Generally used type systems for logic programming with parametric polymorphism and subtyping have an anomaly which weakens the ability to detect type errors as shown in Sec. 2. Our new type system Typical overcomes this anomaly.

It provides type checking at a very detailed level without restricting common programming practice.

The Typical typing approach is independent of any mode system for specifying an input/output behavior for predicates. In this way it differs from other proposals e.g., using so-called implication types [26,25] or type dependencies (e.g. [14]). 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. Typical does not impose a functional or directional view on logic programs as it is done by further systems with 'directional types' and variants thereof (see e.g. [5,27,1,4]). 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. In [7] and similarly in [23] mode declarations are used which are syntactically similar to our type declarations. However, in Typical the declarations are exploited for checking clauses for logical consistency with the model of the declarations instead of checking input/output correctness.

Here, we could only present an overview of the complete type checking and inferencing algorithms underlying Typical; they are spelled out in detail in [17]. Apart from providing a method for dealing with negation, Typical contains several extensions for higher-order programming and extra-logical built-ins (e.g. a built-in type `goal` which is used in declarations like `:- pred call(goal).`) such that the system could be applied successfully to its own source code of about 4000 lines of Prolog code [16,17]. A more refined treatment of such higher order features within the types-as-approximations approach is subject of our current work.

Acknowledgements

We thank the anonymous referees for helpful comments.

References

1. A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In *Static Analysis Symposium*, LNCS. Springer Verlag, 1994. 264
2. C. 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. 260
3. C. Beierle and G. Meyer. Run-time type computations in the Warren Abstract Machine. *The Journal of Logic Programming*, 18(2):123–148, Feb. 1994. 252, 263
4. J. Boye and J. Małuszyński. 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. 264

5. F. Bronsard, T. 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. 264
6. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, Dec. 1986. 260
7. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In B. L. Charlier, editor, *Static Analysis, First International Symposium, SAS'94*, volume 864 of *LNCS*, pages 281–296, Namur, Belgium, Sept. 1994. Springer Verlag. 258, 264
8. M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991. 251
9. M. Hanus. Parametric order-sorted types in logic programming. In *Proc. TAPSOFT'91*, volume 494 of *LNCS*, pages 181–200, Brighton, Apr. 1991. Springer Verlag. 252, 262
10. M. Hanus. Logic programming with type specifications. In Pfenning [24], chapter 3, pages 91–140. 252
11. P. Hill and J. Lloyd. *The Gödel Programming Language*. Logic programming series. The MIT Press, 1994. 252
12. P. M. Hill and R. W. Topor. A semantics for typed logic programs. In Pfenning [24], chapter 1, pages 1–62. 251, 252, 253, 257, 260, 260
13. M. Höhfeld and G. Smolka. Definite relations over constraint languages. IWBS Report 53, IBM Scientific Center, Stuttgart, Germany, Oct. 1988. 252
14. M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. Techn.Rep. 90/23, SUNY, New York, July 1990. 264
15. T. K. Lakshman and U. 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. 251
16. G. Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Informatik Berichte 200, FernUniversität Hagen, June 1996. available via `http://www.fernuni-hagen.de/pi8/typical/`. 258, 264
17. G. Meyer. *On Types and Type Consistency in Logic Programming*. PhD thesis, FernUniversität Hagen, Germany, 1999. (to appear). 258, 262, 262, 264, 264
18. G. Meyer and C. Beierle. Dimensions of types in logic programming. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction - A Basis for Applications*, chapter 10. Kluwer Academic Publishers, Netherlands, 1998. 251, 252, 253
19. P. 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. 256
20. J. Mitchell. Type inference and type containment. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *International Symposium Semantics of Data Types*, number 173 in LNCS, pages 257–277, Sophia-Antipolis, France, June 1984. Springer Verlag. 260
21. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984. 252, 252
22. L. Naish. Types and intended meaning. In Pfenning [24], chapter 6, pages 189–216. 252, 253, 254, 254, 263, 263
23. L. Naish. A declarative view of modes. In M. Maher, editor, *Proc. of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199, Bonn, Sept. 1996. The MIT Press. 258, 264

24. F. Pfenning, editor. *Types in Logic Programming*. Logic Programming Series. The MIT Press, 1992. 251, 265, 265, 265
25. C. Pyo and U. 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. 264
26. U. S. Reddy. Notions of polymorphism for predicate logic programs. In *Int. Conf. on Logic Programming*, pages 17–34, (addendum, distributed at conference). The MIT Press, Aug. 1988. 264
27. Y. Rouzaud and L. 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. 264
28. G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Universität Kaiserslautern, Germany, 1989. 251, 252, 253
29. Z. Somogyi. The Mercury project. http://www.cs.mu.oz.au/˜/mercury.html, 1995. 252
30. J. J. Wu. *A First-order Theory of Types and Polymorphism in Logic Programming*. PhD thesis, State University of New York at Stony Brook, 1992. 253