Electronic Notes in Theoretical Computer Science 30 No. 1 (1999)
URL: http://www.elsevier.nl/locate/entcs/volume30.html 21 pages

A pragmatic type concept for Prolog supporting polymorphism, subtyping, and meta-programming [1]

Christoph Beierle [a], Reinhold Kloos [a], Gregor Meyer [b]

[a] *FernUniversität Hagen, FB Informatik, 58084 Hagen, Germany,*
{beierle, reinhold.kloos}@fernuni-hagen.de

[b] *IBM Germany, SWSD, 71003 Böblingen, Germany*
grmeyer@de.ibm.com

Abstract

Approaches to typing logic programs often exclude various features of Standard Prolog. The system "Typical for annotated Prolog" (TaP) is a pragmatic approach to type checking programs written in Prolog without restricting the scope of the language. TaP checks Prolog programs that are extended with type declarations that support parametric polymorphism and subtyping. The purpose of this paper is to present an approach that extends Typical by meta-types for handling Prolog meta-programming techniques.

1 Introduction

There has been extensive research on the topic of types in logic programming, see e.g. the collection of papers in [16]. 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 promininent one: The aim is to detect as many programming errors as possible by static program analysis before running the program. However, approaches to typing logic programs often exclude various difficult-to-type features of Standard Prolog. The aim of the Typical system [12,13] is a pragmatic approach to type checking Prolog programs without restricting the scope of the language. The Typical system checks Standard Prolog programs that are extended with type declarations using subtypes and parametric polymorphism. It has been applied successfully to e.g. its own source code of about 4000 lines of Prolog code.

[1] Part of the research reported here was supported by the Deutsche Forschungsgemeinschaft (DFG) under grant Be 1700/3-1.

The purpose of this paper is to present an approach [9] that extends Typical by meta-types for handling Prolog meta-programming techniques. The examples we give include meta-predicates like `assert/1`, `retract/1`, `sort/3`, `map/3`, and `foldl/4`.

This paper is organized as follows: In Sec. 2 we indicate some of the problems encountered when polymorphism is combined with subtyping in logic programming. Sec. 3 provides a brief overview on the first-order part of the Typical type system. In Sec. 4, an extension of this type system towards the support of meta-programming is presented [9]. Sec. 5 discusses the current limitations of our approach, and Sec. 6 points out some conclusions and further work.

2 Subtyping and polymorphism

An ML-like type system for logic programming was proposed and implemented by Mycroft and O'Keefe [14]. It includes explicit type declarations and parametric polymorphism but no subtypes. Most prominently the languages Gödel [7] and Mercury [19] 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'.

Type systems with parametric polymorphism and subtypes were proposed by [18,8,5,6,2,21] and others. However, these approaches have a serious shortcoming when it comes to detect obviously ill-typed expressions involving subtypes.

Example 2.1

```
:- subtype nat < int.                 %       int
:- subtype negint < int.              %     /     \
                                      % negint   nat

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

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

With `-1` of type `negint` and the variable `X` having the type `nat` the goal 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.

Example 2.2

```
:- 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`. 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. [18,8]) 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; [20] 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. Furthermore, we will illustrate to what extent this type system can also deal with meta types.

3 Overview on the Typical system

In this section, we give a brief overview on the Typical type system. In order to ease the presentation, we stick to first-order case here and do not discuss any meta-type aspects; these are presented in Sec. 4

3.1 The type language

All types used in the predicate declarations of a program and the corresponding function symbols must be defined. The basic form of monomorphic type definitions are the same as already used in the examples of Sec. 2. 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. 3.3. A precise description of these conditions is given in [12].

3.2 Predicate declarations

Predicate declarations specify the types which are expected for the actual arguments when a predicate 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 type parameters, written as Prolog variables with the prefix '`@`'. For instance, the declaration for `reverse` is

```
:- pred reverse(list(@T), list(@T)).
```

More specific type declarations are possible by using *type constraints* over type variables occurring in the declaration:

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

The expressions on the right of `|>` describe type constraints where `=<` stands for the subtype relationship. Note that the prefix `@` is is not used within the constraint part.

3.3 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 [12,13]. 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:

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

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

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

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, let us consider the atom `append(L2,[-1],[1])`. Now, the third argument has the least type `list(nat)`. This atom, although directly comparable to the former one, 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.

Thus, 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 $\texttt{append}_{\texttt{old}}$.

Now 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. Using this declaration for the equality predicate also the type error in `:- abs(Z,X), X = -1.` from Example 1 is detected.

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 member of
%                                    both L1 and of L2.

:- pred overlap(list(@T), list(@U)) |> V =< T, V =<U.
% overlap(L1,L2) :- lists L1 and L2 have members in common.
```

For a predicate whose declaration contains type parameters we say that a clause is *weakly type-consistent* if it fits *at least one* instance of the type parameters; it is called (generically) type-consistent if it fits *all* instances. (This is the so-called head condition, c.f. [18,8,10].) For example, the clause `reverse([X],[X]).` is (generically) type-consistent w.r.t. the declaration for `reverse` given above, but the clause `reverse([1],[1]).` is only weakly type-consistent. Typical supports both variants of type-consistency.

In [12] 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 [12].

3.4 Anchored and non-anchored type parameters

Typical uses the syntactic symbol `@` in predicate declarations in order to prefix type parameters within formal arguments. Since these type parameters refer to the principal type of a term, we call them *anchored* type parameters.

In order to simplify the declaration of predicates, Typical provides a simple but powerful syntactic abbreviation by allowing also non-anchored type parameters. A non-anchored type parameter `T` is seen as an abbreviation for a new anchored type parameter together with a subtype constraint as follows:

> Every occurrence of a non-anchored type parameter `T` within formal arguments is implicitly replaced by a *new* anchored type parameter `@Tnew` and a constraint `Tnew =< T` is added to the constraint part.

Hence, according to this transformation the declaration

```
:- pred append_new(list(T), list(T), list(@T)).
```

is implicitly replaced by

```
:- pred append_new(list(@Tnew1), list(@Tnew2), list(@T))
                                   |> Tnew1 =< T, Tnew2 =< T.
```

Note that this declaration is obviously equivalent to the $\texttt{append}_{\texttt{new}}$ declaration given in the previous subsection since it is just a variable-renamed instance of it.

As another example, consider the comparison of the two different declarations for the predicate `sublist/2` determining if the first list is a sublist of the second one:

```
% declaration 1 with anchored type-parameters
:- pred sublist(list(@T1), list(@T2)) |> T1 =< T2.

% declaration 2 with non-anchored type-parameters
:- pred sublist(list(T1), list(T2)) |> T1 =< T2.

% declaration 2 rewritten with anchored type-parameters
:- pred sublist(list(@T1new), list(@T2new))
                   |> T1new =< T1, T2new =< T2, T1 =< T2.
```

It is obvious, that the declaration 2 of `sublist/2` with non-anchored type-parameters for T1 and T2 has weaker contraints in comparison to declaration 1 with anchored type-parameters.

Please keep in mind that anchored type-parameters allow us to describe constraints more precisely. However, with non-anchored type-parameters in a declaration, we can make some useful abbreviations which furthermore fit nicely with other common notations. In the rest of this paper we will frequently take use of this abbreviated declaration style.

4 An approach to extend Typical with meta-types

In this section we present an extension of our type-concept, allowing us to describe not only first-order types, but higher-order types (meta-types), too. In the first subsection we give some examples of predicates for the meta-programming. Later on we introduce our meta-type description for the possible meta-structures in such a predicate. Several examples for the application of the meta-type with outputs of the type-checker will end this section.

4.1 Motivation

Now, we present some predicates of meta-programming in order to show different aspects of a meta-structure. The predicates assert/1 and retractall/1 show, that already Standard-Prolog predicates need meta-types for their correct declaration.

```
?- map(inc, [1,3,5,6], X).     % result:  X = [2,4,6,7]
```

The predicate `map/3` gets a meta-structure in the first argument: the functor of a predicate with the arity two. In the given example, the functor comes from the predicate `inc/2`, which increments the number in the first argument and gives the result in the second argument. The predicate given with the functor in the first argument of `map/3` is applied to every element of the lists, given in the two other arguments of `map/3`. Thus, in our example the variable X will be unified with the list [2,4,6,7], where every element of the first list was incremented.

A further example for a predicate with a meta-structure as an argument is `assert/1`. Here the meta-structure is a complete clause or fact.

```
?- assert((append([A|B], C, [A|D]) :- append(B, C, D))).
```

The meta-structure used as an argument in the predicate `retractall/1` is the head of a clause or simply a fact to be removed from the database.

```
?- retractall(append(A, B, C)).
```

4.2 Dimensions of a meta-type

The examples demonstrate that the description of a type for a meta-structure is more complex than the description of a first-order type. Thus, there are several kinds of additional information combined with a meta-type. We call these different kinds of information the dimensions of a meta-type:

- Type-arguments and arity
- argument application (currying)
- nature of meta-type

4.2.1 First dimension: Type-arguments and arity

In Typical a predicate can be declared with arguments containing a meta-type. Like the predicate itself a meta-type has a description of its type-arguments and its arity. It is possible to use type-parameters, in order to describe constraints to other arguments in the predicate declaration. The type-arguments and arity of the meta-type must fit to the type-arguments, given in a definition of a function or in a declaration of a predicate, respectively, depending on the actual meta-structure.

4.2.2 Argument application

The meta-structure can be a function or an atom (callable predicate) with all its arguments. However a functor is possible as a meta-structure, too. Therefore an additional specification for the argument application of a meta-structure is necessary. The keyword `trmName` states that just one functor (with no arguments) is expected. A meta-type for a meta-structure with all its declared or defined arguments contains the keyword `trmFull`.

Let us suppose we have the following declarations:

```
:- type machine --> fastm(int, string); slowm(int, string).
:- pred inc(int, int).
:- pred map( meta(trmName(T1,T2)), list(T1), list(T2)).
```

The first argument's type of `map/3` is a meta-type given by `meta(trmName(T1,T2))`. It describes the possible meta-structures for the first argument of the predicate `map/3`. So this argument of `map/3` must be a functor with the arity two, which is applied to the elements of the lists. The type-parameters show the dependencies between the different arguments of the predicate.

With the given declaration of `map/3`, the results of the type checker are:

```
tap: Info
   clause
    :-map(inc,[1,3,5,6],A).
    is type-consistent with variable-typing:
    [A*list(int)]
```

However, the description of the meta-type in our declaration is not complete:

```
tap: Info
   directive
    :-map(fastm,[1,2,3],A).
    is type-consistent with variable-typing:
    [A*list(string)]
```

The given directive is type-consistent, although `fastm` is not a functor of an executable predicate but a functor of a function belonging to the type `machine`.

Let us look at a further example. The predicate `assert/1` expects not just a functor like `map/3`, but a meta-structure complete with all its arguments.

This kind of argument-application is expressed by the keyword `trmFull`. With the declaration

```
:- pred assert( meta(trmFull)).  % a meta-structure
                                 % with all arguments
```

the directives

```
:- assert(inc(1,2)).
:- assert(map(inc,[1],[2])).
```

are accepted as type-consistent. However, a term being neither a clause nor a fact is accepted, too:

```
tap: Info
   directive
    :-assert((a,b)).
    is type-consistent with variable-typing:
    []
```

Obviously we need more fine-grained restrictions for meta-structures in order to accept e.g. `:- assert((a :- b)).` as being type consistent, but rejecting `:- assert((a,b)).`, where `a`, `b` are nullary predicates.

4.2.3 Nature of a meta-type

A further dimension of a meta-type is a more detailed description of its (what we call) *nature*. In the given example of the predicate `map/3` only a functor of a *declared predicate* should be in the first argument. We express this restriction of the meta-structure with the *nature* `mpred`. The declaration

```
:- pred map( meta(trmName(T1,T2) of mpred), list(T1), list(T2)).
```

requires that the first argument of `map/3` must be a functor with the *nature* `mpred`.

Now the directive `:- map(fastm, [1,2,3], X).` is no longer type-consistent:

```
  The directive
     :- map(fastm,[1,2,3],A).
    is not type-consistent.

    The directive with the last atom GENERALIZED to
    map(A4,[1,2,3],A)
    would be type-consistent using
        VT=[A4*meta((trmName([posint,A1])of mpred)),A*list(A1)]
```

Comment: The type-checker gives some information about a solution of the problem. The meta-structure in the first argument is replaced by the variable A4. The type-checker demands that the meta-structure must be of the nature `mpred`. The constraint between the type-parameters is already resolved as far as possible.

In order to declare that only a clause or a fact can be the argument of `assert/1`, we use the nature `mclause`. Thus, an assert of a conjunction is no longer type-consistent:

```
:- pred assert(meta(trmFull of mclause)).

  The directive
     :- assert((a,b)).
    is not type-consistent.

    The directive with the last atom GENERALIZED to
    assert(A4)
    would be type-consistent using
        VT=[A4*meta((trmFull(A1) of mclause))]
```

Comment: The type-checker sets a variable for 'not given' dimensions of a meta-type. Thus, there is the variable A1 for the arguments of the meta-type.

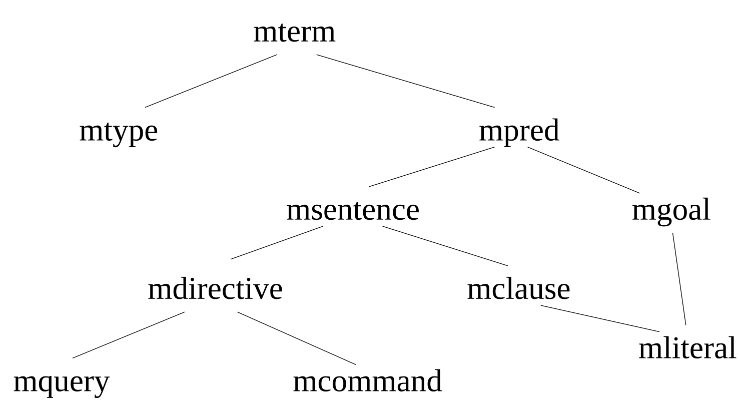

Fig. 1. The hierarchy of natures for meta-types

Our concept supports a hierarchy of natures (c.f. Figure 1). The nature called `mterm` gives no restrictions for meta-structures. The first refinement of the nature `mterm` is, whether the meta-structure is connected with a type-definition or a predicate declaration (`mtype`, `mpred`). Further natures describe, whether the meta-structure should be a clause (or fact) like in `assert/1` (`mclause`) or a Prolog-goal (`mgoal`). There is a connection between the natures `mliteral`, `mclause` and `mgoal`, because a logical atom can be used as a goal or as a fact. At the bottom of the hierarchy the meta-structure is restricted to be a single function introduced by a type-definition (`mtype`) or to be a logical atom (`mliteral`). Thus, here is the connection between the meta-type and a first-order type-definition or a first-order predicate declaration. With the natures of this hierarchy we can declare most of the built-in meta-predicates very precisely:

```
:- pred retractall(meta(trmFull of mliteral)).
:- pred ','(meta(trmFull of mliteral),           % conjunction
            meta(trmFull of mgoal))
:- pred ':-'(meta(trmFull of mliteral), meta(trmFull of mgoal))
```

The current implementation of Typical offers the natures `mpred`, `mclause` and `mtype`. If you give a meta-type without any mentioned nature, the default nature `mpred` will be taken. In the following we will illustrate our meta-type concept with these natures. It already provides a very flexible method to deal with many popular meta programming techniques. A detailed description of the extended typing rules for meta-typing and the realized implementation is presented in [9].

4.3 Further Examples of meta-typing

The following type-definitions and predicate declarations are used in the examples:

```
:- type int > posint.
:- type int > negint.
:- type int > zero.
:- type list(T) --> [ ]; [T|list(T)].
:- pred append( list(T), list(T), list(@T)).
:- pred top =.. list(top).
```

where `posint` (resp. `negint`) is the type of all positive (resp. negative) integers. `zero` is the type just containing 0, and `top` is a default type containing every Prolog term (the most common type). The type `top` is only known for the type inference at certain positions, where it is given in a used declaration. Thus, this special type has no influence on the determination of the least type at other positions.

4.3.1 `fcall/N` - Capsulation of the meta-call

In Standard-Prolog there is the meta-call predicate `call/1` which allows to call the predicate given in the argument. The predicate `call/1` can handle every callable predicate. Thus, one can only give the following predicate declaration for `call/1`:

```
:- pred call( meta(trmFull of mpred) ).
```

The great disadvantage of this declaration is that one has no control of the types or the arity of the term given in the first argument of `call/1`.

Therefore, we use the predicate `fcall/N` [15]. The predicate `fcall/N` takes in the first argument the functor of a predicate with the arity N-1. The other N-1 arguments of `fcall/N` are the arguments fitting to the functor in order to make the meta-call. The advantage of using `fcall/N` over the direct use of `unif/2` and `call/1` is the capsulation of these two predicates. Furthermore with `fcall/N` Typical can check the functor itself and every argument of the meta-call.

E.g., the declarations for `fcall/1` and `fcall/3` are:

```
:- pred fcall(meta(trmFull([ ]) of mpred)). % for calls with
                                             % no arguments
fcall(A) :- atom(A), call(A).

:- pred fcall(meta(trmName([T1,T2]) of mpred), T1, T2).
fcall(A, B, C) :- atom(A), D =.. [A, B, C], call(D).
```

`fcall/1` is for the call of a predicate (`mpred`) with no arguments (arity zero). The argument is callable (`trmFull`). `fcall/3` is for the call of a predicate (`mpred`) with two arguments. The first argument is only a functor (`trmName`). The types of the arguments of the meta-structure correspond with the types of the further arguments of `fcall/3`.

Type-Check of a defining clause for `fcall/3`:

```
tap: Info
   clause
    fcall(A,B,C):- atom(A), D =..[A,B,C], call(D).
    is type-consistent with variable-typing:
    [A:meta(trmName([#1,#2])of mpred), B:#1, C:#2,
                        D*meta(trmFull(A1) of mpred)]
```

Comment: The checked clause is generically type-consistent; that means it fulfills the head-condition. Thus, there is no specific substitution of the type-parameters T1 and T2 with a defined type symbol. The type-checker Typical shows this with the special type symbols #1 and #2, which still allow to see the constraints. The star * indicates that Typical determinded a least type.

4.3.2 `listmap/3` - An application of the predicate `fcall/N`

The predicate `listmap/3` takes in the first argument the functor of a predicate which is applied to the lists in the second and third argument. listmap is a direct application of `fcall/3`. It is the semantic described by the types, which makes the difference:

```
:- pred listmap(meta((trmName(list(T),list(T)) of mpred)),
                list(@T),list(@T)).
listmap(A, B, C) :- fcall(A, B, C).
```

```
tap: Info
   clause
    listmap(A,B,C):-fcall(A,B,C).
    is type-consistent with variable-typing:
    [A*meta(trmName([list(#1),list(#1)]) of mpred), B*list(#1),
                                                     C*list(#1)]
```

It is possible to use the higher-order predicate `listmap/3` in order to test algorithms dealing with two lists of the same type, e.g. sort algorithms. In the following example we assume that there is a predicate `delete_doubles/2` deleting all further occurrences of an element. Then the fourth query with listmap given below is not type-consistent.

```
:- pred bubblesort(list(@T), list(@T)).
:- pred mergesort(list(@T), list(@T)).
:- pred delete_doubles(list(@T), list(@T)).

?- listmap(bubblesort, [3, 1, 2], A).           % A = [1, 2, 3]
?- listmap(mergesort,  [3, 1, 2], [1, 2, 3]).
?- listmap(delete_doubles, [3, 1, 2, 1], [3, 1, 2]).
?- listmap(delete_doubles, [3, 1, 2, 1], A), A = [-1].
```

```
  The query
     :- listmap(delete_doubles,[3,1,2,1],A).
    is type-consistent using VT=[A*list(posint)]
    With additional subgoal A=[-1] the query
     :- listmap(delete_doubles,[3,1,2,1],A), A=[-1].
    is not type-consistent.
    The clause/query with the last atom GENERALIZED to A4=[-1]
    would be type-consistent using
        VT=[A*list(posint),A4*list(negint)]
```

The predicate declaration of `listmap/3` leads to a clear restriction of its usage. The predicate called by `listmap/3` is restricted by the least types of both lists. This is useful in the given examples. For other cases, it may be better to use other higher-order predicates like `map/3` or `filter/3`.

4.3.3 filter/3

The meta-predicate `filter/3` is comparable to `map/3`. It filters the elements of the input list fulfilling the predicate given by a functor in the first argument. As opposed to `map/3` this predicate must be of arity one.

```
:- pred filter(meta((trmName(@T) of mpred)), list(T),list(T)).
% meta-structure = functor of predicate p, with p: T
filter(_, [], []).
filter(A, [B|C], D) :-
        (fcall(A,B) -> D = [B|E];
                       D = E),
        filter(A, C, E).

% result of type-checker:
    filter(A,[B|C],D):-(fcall(A,B)->D=[B|E];D=E),filter(A,C,E).
    is type-consistent with variable-typing:
    [A*meta(trmName([#1])of mpred),B*#1,C*list(#1),D*list(#1),
```

```
E*list(#1)]
```

The Type-Checker controls the correct arity of the meta-structure:

```
The query
    ?- filter(inc,[1,2,3],A).
    is not type-consistent.
    The clause/query with the last atom GENERALIZED to
    filter(A4,[1,2,3],A)
    would be type-consistent using
    VT=[A4*meta((trmName([posint])of mpred)),A*list(posint)]
```

Comment: The query `filter(inc,[1,2,3],A).` is ill-typed, because the predicate `inc/2` is of the wrong arity. The proposed solution has only one argument, which is already set to type `posint`.

4.3.4 foldl/4

The predicate `foldl/4` is the left fold function known in functional programming:
foldl(op, b, [e1,e2,...,en], ((...((b op e1) op e2)...) op en))
The first three arguments of `foldl/4` contain the input. In the first argument is a functor `op` of a predicate. This predicate (op/3) has the arity three with two input and one output argument. It is applied to every element in the list given in the third argument of `foldl/4`. The first left argument for these repeated applications of the operation op/3 is the value `b` given in the second argument of `foldl/4`. Afterwards the left argument for the next operation is everytime the result of the application of op/3 before. The fourth argument of `foldl/4` contains the result of all these operations.

The meta-typing adapts very precisely the special function of the predicate `foldl/4`:

```
:- pred foldl(meta((trmName(@T1,@T,@T1) of mpred)),
              T1, list(T), @T2)  |>  T1 =< T2.
% meta-structure = functor of predicate op/3, with op: T1 x T x T1
foldl(A, B, [], B).
foldl(A, B, [C|D], E) :-
        fcall(A, B, C, F),
        foldl(A, F, D, E).
```

The type T1 of the third argument of the meta-structure, which is the target-type of the operation op of `foldl/4`, must be the same as the type of the first argument of the meta-structure or the type of argument b. The operation op must be able to work with every element of the list in the third argument of `foldl/4`. The target-type T2 of `foldl/4` is the upper bound of type T1, so we can give the constraint `T1 <= T2`.

```
tap: Info
   clause
```

```
foldl(A,B,[C|D],E):-fcall(A,B,C,F),foldl(A,F,D,E).
is type-consistent with variable-typing:
[A*meta((trmName([#1,#2,#1])of mpred)),B*#1,C*#2,D*list(#2),
                                                     E*#1,F*#1]
```

```
The query
    :- foldl(append,[],[[1,2,3],[5,6],[7,8,9]],[-1]).
    is not type-consistent.
    The clause/query with the last atom GENERALIZED to
    foldl(A4,[],[[1,2,3],[5,6],[7,8,9]],[-1])
    would be type-consistent using
    VT=[A4*meta((trmName([list(negint),list(posint),
                                  list(negint)]) of mpred))]
```

Comment: We gave the declaration of the predicate append/3 in Sec. 3.3:
`:- pred append(list(T), list(T), list(@T)).`
The type-parameter of the third argument of this predicate is an anchor point for the type-parameters of the lists in the first both arguments. That means that the type of the last type-parameter must be a super-type of the types of the both other type-parameters. As there is only a negative number in the target list of `foldl/4`, a functor fitting the type restriction given for A4 is required. `append` is not such a functor, because `negint` is not a super-type of `posint`.

4.4 *retract/1*

In the following examples we show an application of the nature `mclause`. This nature of a meta-type restricts the meta-structure to a clause or fact. The built-in predicate `retract/1` has the same meta-type as `assert/1`. The built-in predicate `retractall/1` takes as an argument the head of clause or a fact.

```
:- pred assert(meta((trmFull of mclause))). % meta-structure
                                            %    = clause
:- pred retract(meta((trmFull of mclause))). % meta-structure
                                             %    = clause
:- pred a.
:- pred b.
:- pred @T = @T

tap: Info
   query
    ?-retract(a).
    is type-consistent with variable-typing:
    []
```

The directive

```
:- retract(( :- a )).
is not type-consistent.
The clause/query with the last atom GENERALIZED to retract(A4)
would be type-consistent using
    VT=[A4*meta((trmFull(A1) of mclause))]
```

Comment: Directives, queries and goals composed of several logical atoms can not be asserted or retracted. This is expressed by the nature of the meta-type (`mclause`). Thus, the following directives

```
:- retract((a,b)).
:- assert((a -> b)).
:- retract((a;b)).
:- retract((?- b)).
```

are also detected as not being type-consistent by Typical.

5 Limitations of the meta-type concept

5.1 Lists with heterogeneous types

The predicate `flatten/2` flattens an embedded input-list to a non-nested list.

```
:- flatten([[ 1|[2] ]], L).      % L = [1,2]
:- flatten([a,b,[c,[d],e]],K).   % K = [a,b,c,d,e]
```

However, using the definition

```
:- type list(T) --> []; [T | list(T)];
```

as before, works fine as long as the list consists only of elements of one type, like `list(int)` or `list(list(int))`. But what happens if a list is a mix of differently deeply structured list-elements like in the first argument of the predicate `flatten/2`? In order to find a common type for T, there must be a subtyping between differently structured type-names like `list(int)` and `list(list(int))`, which are not allowed in Typical (c.f. Sect. 3.1). Such subtyping would mean an exponential extension of the possible solution-set.

5.1.1 Naive solution with list(top)

```
:- pred flatten(list(top),list(top)).
```

There is a special type `top` in Typical, which accepts terms of every type. Thus, the lists of `flatten/2` in well-typed terms are always of type `list(top)`, like in `flatten([a,b,[c,[d],e]],K)`. On the other hand terms like `flatten(a,K)`. or `flatten(a,[a])` are correctly recognized as ill-typed. However, with this simple typing `list(top)` type-errors, which are in the list, cannot be detected. `flatten([[1|2]], L)` is such an example.

5.1.2 Typing an embedded list with element(T)

For pragmatic reasons, Typical also allows to define types via goal abstraction. Thus, we can define

```
:- type element(T) --> {X | (atomic(X), T = atomic)}.
```

Now every atomic term can also be of type `element(atomic)`. This gives us the possibility to introduce a polymorphic type via subtyping that is a supertype of both list(T) and element(T):

```
:- type list_or_element(T).
:- type list_or_element(T) > element(T).
:- type list_or_element(T) > list(T).
:- pred t(list_or_element(T)).

:- A = [1, [3, 4], 2], t(A). % A is of type
                             % list(list_or_element(atomic))
```

Even `flatten([[1|2]], L)` with the predicate declaration

`flatten(list_or_element(T), list_or_element(T))`

is recognised as ill-typed now.

However, A = `[1, [[3, 4]]]` cannot be typed by `list_or_element(T)`, because there would be an impossible comparison between `element(atomic)` and `list(list(atomic))`. For a solution we could define a type `element(element(T))`, but this is quite inefficient.

Let us give here a remark to the peformance of Typical at all. Our expierience has shown that the type inference of Typical is rather efficient. So we see Typical as a good type-checker for bigger programs. We even checked our program for Typical with more then 600 clauses distributed over several files. However, the amount of type-parameters in the used declarations and the size of the universe of defined basic types influence the efficiency of the type-inference. The `element(atomic)` doubles the possible solutions at least. At last there seems to be no better typing for the `flatten/2` predicate than `list(top)`.

5.2 Typing of the meta-predicate `unif/2`

The limitation, described in the chapter before, can also be seen as a special case of the fact that it is rather difficult for a type-checker to control the type of every element of a heterogeneous list separately. The actual length and contents of a list can only be determined during the execution of a program. Yet a type-checker should be able control the types of a heterogeneous list, in order to find a fitting typing for the predicate `unif/2 ('=..'/2)`. The predicate `unif/2` is used in the meta-programming of Prolog for building a callable predicate from a list or conversely. As the list changes during the execution, the typing of the predicate `unif/2` must be dynamic and the type-checker must consider information formed at runtime. This is not compatible with the actual static concept of Typical .

5.3 *Typing of the meta-predicate* ***apply/3***

A more complicated example for a predicate, dealing with up to two m eta-structures, is the predicate `apply/3`. The `apply/3` predicate is used, in order to implement a meta-call of a logical atom with more than one argument. The meta-predicate `apply/3` is given as an alternative to `call/N` in [15]. The laster one is comparable to our introduced predicate `fcall/N`. The advantage of these predicates in comparison to the Standard-Prolog predicate `call/1` is that you can control the type of every argument of the meta-call.

In the following example we give a directive, which implements a meta-call for the logical atom `append([1,2], [3], [1,2,3])`:

```
?- apply(append, [1,2], A), apply(A, [3], [1,2,3]).
                                       % A = append([1,2])
```

The first argument of `apply/3` is always a meta-structure. The arity of this meta-structure is always lower than the arity of the logical atom to be called. Like `append` and `append([1,2])` in our example, this meta-structure is an intermediate stage for the building of the logical atom, motivated by the idea known as currying in the functional programming. In order to build the meta-call, the predicate apply/3 is used in two different semantic ways: If the difference between the arity of the argument and the arity of the logical atom is two, then `apply/3` will use the values of the second and third argument as the last two arguments of the meta-call; otherwise `apply/3` builds a new meta-structure in the third argument. The new meta-structure is the original meta-structure extended by one argument, given in the second argument of `apply/3`. Thus, like in our example, it is oftenly necessary to build a sequence of several `apply/3`, in order to make one meta-call.

At the moment, it is not possible to give a precise declaration for this predicate, because there are no constructors for the meta-type to describe the currying up to now. In future, we want to introduce a so-called `apply-value`, which can be seen as an extension of the dimension decribing the argument application of a meta-type. Having the `apply-value`, we could describe that the meta-structure may use only a subset of the possible arguments. Furthermore, there is another difficulty to give a correct declaration for the predicate `apply/3`. As this predicate has two alternative applications, we will have to introduce the possibility of alternative declarations in Typical too.

6 Conclusions and further work

In this paper we introduced a meta-type concept for Typical, a system that supports both subtypes and parametric polymorphism.Our typing approach is independent of any mode system for specifying an input/output behaviour 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. [4,17,1,3]).

Typical supports Standard Prolog. The information for the type-checker are given by normal Prolog directives, which are transparent at run-time. With the extension for meta-typing presented here, Typical supports detailed type declarations for higher-order predicates. This flexibility is achieved by introducing several dimensions of a meta-type like, for example, currying and parametric-polymorphism. Having the nature of a meta-type, the type can be restricted to specific meta-structures. However, as indicated in the last section, there are currently still some limitations, e.g. in typing the `unif/2` predicate.

In the future, we want to support more natures of meta-type and an improved currying with the apply-value. Details about the formal foundations of Typical can be found in [11,12]. The meta-type concept presented here is described in [9].

References

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

[2] C. 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.

[3] 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.

[4] 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.

[5] 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.

[6] M. Hanus. Logic programming with type specifications. In Pfenning [16], chapter 3, pages 91–140.

[7] P. Hill and J. Lloyd. *The Gödel Programming Language*. Logic programming series. The MIT Press, 1994.

[8] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In Pfenning [16], chapter 1, pages 1–62.

[9] R. Kloos. Extension of a type-concept for Prolog for the support of modular software development and meta-programming in AI. Diplomarbeit, FernUniversität Hagen, 1998. (in German).

[10] 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.

[11] 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/`.

[12] G. Meyer. *On Types and Type Consistency in Logic Programming.* PhD thesis, FernUniversität Hagen, Germany, 1999. (to appear).

[13] 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.

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

[15] L. Naish, R. A. O'Keefe, and F. Henderson. Nettalk: Meta-call with call/N and apply/3. *Logic Programming Newsletter*, 10(1), Feb. 1997.

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

[17] 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.

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

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

[20] 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.

[21] E. Yardeni, T. Frühwirth, and E. Shapiro. Polymorphically typed logic programs. In Pfenning [16], chapter 2, pages 63–90.