Informatik Berichte 199, FernUniversität Hagen, Germany

On the Use of Types in Logic Programming*

Gregor Meyer

FernUniversität Hagen, Praktische Informatik VIII

58084 Hagen, Germany

gregor.meyer@fernuni-hagen.de

June 1996

Abstract

In procedural and functional languages it is rather clear what a type error is and the main purpose of typing is to detect programming errors automatically. Furthermore, in these languages the semantics of a program is usually independent of types. Type systems for logic programming languages differ considerably on these aspects. We will show that these differences lead to several theoretical and practical problems. The mixture of static type checking purposes on the one hand and computations with type constraints in the CLP(X) style on the other hand will be marked as the central source of problems. As a solution we propose a typing scheme which separates these issues. We distinguish three almost independent dimensions of types in logic programming: types as approximations, types as constraints and types for proving partial correctness. Based on the notion of useless expressions, we develop a method that allows static program analysis for finding type errors, which enables error detection to be more precise than hitherto possible.

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 "Type Checking and Type Inferencing for Logic Programming with Subtypes and Parametric Polymorphism" [Mey96].

*Research supported by DFG within "Schwerpunkt Deduktion"

Contents

1 Introduction

The key advantage of using a typed programming language instead of an untyped one is that a compiler is able to spot programming errors. This is done automatically by static analysis and the system can point to the location of an error more or less exactly. Further benefits of types are their support in modelling, documentation, and abstraction, especially Abstract Data Types (ADT's). For logic programming languages there is another interesting use of types. Several proposals use types as a special domain for constraint programming in the CLP(X) scheme. The purpose of this report is to analyze various approaches, identify different dimensions of using types, and to present a new approach that clearly separates these dimensions thereby avoiding several theoretical and practical problems.

Our proposal aims at the following goals. We want to add a type system to logic programming that allows a compiler to automatically find as many programming errors as possible by static analysis. The type system should fit well with the declarative and non-procedural nature of logic programs. It should be expressive enough to support rich modelling and various programming techniques. As an additional goal we are looking for a type system that is applicable to standard Prolog, including its non-logical features.

For procedural and functional languages there seems to be a common agreement on the use of types. They are used for finding programming errors, i.e., the program can be statically checked to be free of type-errors, which could lead to undefined expressions. For logic programming languages one is confronted with a wide spread of typing schemes and their utilization. As a contribution of this report we will identify three important kinds of handling types. These kinds are often mixed within one proposal without further notice, although we think that these kinds represent almost orthogonal dimensions. We claim that a separation of different dimensions helps to understand types in logic programming. We will use it to show weaknesses in various proposals and it is the basis for a new approach presented in this paper.

As a basic assumption we expect predicate declarations to be given explicitly in a program. Predicates are defined by logical formulas, usually definite clauses. The different dimensions of using type declarations are:

Types for proving partial correctness: Types can define requirements which are imposed on program expressions to be well-defined. In part they are needed to ensure the correctness of a specific (optimized) implementation. The requirements can refer to states reached by a specific inference calculus or simply refer to a model of the program. Static type checking then can prove that these requirements are guaranteed to hold for a given program.

Types as constraints: A type structure can be used as a special constraint theory within the CLP(X) scheme. Type constraints enhance the expressivity of the programming language and sometimes can be exploited in order to speed up deductions.

Types as approximations: Predicate type declarations can be seen as approximations for the set of atoms which are intended to be true. Declarations can express a necessary condition or even a sufficient condition. Static analysis can then detect useless expressions, which are either always false or always true.

Another aspect for the comparison of typed logic languages is how they define the semantics of typed predicates. Neglecting the details about a typed or untyped universe of terms, there are generally two variants. The first uses the clauses *and* the type declarations to define the semantics of a program. This is very close to what Reddy calls prescriptive typing [LR91]. With the second variant the semantics of a predicate is independent of its type declaration. This relates to what is called descriptive typing.

Although the first variant seems to be more natural, we will show that there are unfortunate results of mixing up static type checking with semantical impacts of type declarations. These shortcomings are theoretical as well as practical. A major problem is that the notion of a type error, as it is commonly used for procedural and functional languages, has no direct correspondence with this variant. Furthermore, even if these systems use a notion of static type checking, they are related to what is called dynamic type checking in conventional systems. With prescriptive typing, predicate declarations imply type constraints for clauses, i.e., they have influence on the program semantics. Concerning static type checking, predicate declarations are used as approximations. A result of our separation of dimensions is that prescriptive typing is a mixture of the 'types as constraints' dimension and the 'types as approximations' dimension.

Instead, we will favor the second variant, related to descriptive typing, which comes close to the dimension 'types as approximations'. We propose a new approach which is widely applicable, includes subtyping and parametric polymorphism, enables detailed static type checking, and allows program implementations to be very efficient. Its core component is the detection of useless expressions, which are not consistent with type declarations.

The paper is organized along the three typing dimensions. After some preliminary remarks on type languages the type checking dimension is briefly touched in section 3. In section 4 we focus on types as constraints and analyze several problems related to the semantics of type inferencing and the use of type constraints in practice. In section 5 we concentrate on types as approximations and give a general method for finding ill-typed expressions. In the conclusion we reconsider the interplay between the different typing dimensions and recommend a framework which combines the benefits of each dimension.

2 Typed programs

Without claiming to give a complete list of variants, type declarations can use a rather general constraint language [HS88], or logical expressions, e.g., [XW90, Nai92], equational specifications[Han90], regular sets [Mis84, MR85, HJ92, DZ92, AL94], or type terms over a type structure possibly with parametric polymorphism and subtyping [MO84, LR91, Smo89, HT92, Han91]. See also [Pfe92] for a collection of proposals.

For two of these variants we will give short examples, first using type terms and second using logical rules.

```
type person := male ++ female.  %     person
type male := {a,b}.             %    /      \
type female := {c,d}.           %  female   male

type family := {fam(female,male,list(person))}.
type list(T) := { [] , [T|list(T)] }.

pred '=': T x T.

pred know: person.
  know(a).  know(b).  know(c).

pred father: male x person.        pred mother: female x person.
  father(a,b).                       mother(c,a).
                                     mother(d,b).

pred parent: person x person.
  parent(P,C) :- mother(P,C).  % P:female, C:person
  parent(P,C) :- father(P,C).  % P:male, C:person.
```

The program defines a small type hierarchy with type `person` having subtypes `female` and `male`. The other types are disjoint to `person`. Function symbols and their argument types are given by enumeration. Predicate type declarations, marked by the keyword `pred`, define the types of arguments as usual.

The same information about the types can be represented using logical predicates instead of a special type language as follows (where we use ordinary Prolog rules):

```
typ(person,X) :- typ(male,X).
typ(person,X) :- typ(female,X).
typ(male,a).    typ(male,b).  typ(female,c).  typ(female,d).
```

```
typ(family,fam(X,Y,C)) :- typ(female,X), typ(male,Y),
                          typ(list(person),C).

typ(list(T),[]).
typ(list(T),[X|Xs]) :- typ(T,X), typ(list(T),Xs).
```

The predicate declarations can be given as implications

```
pred '='(X,Y) -> typ(T,X), typ(T,Y).
pred know(X) -> typ(person,X).
pred father(X,Y) -> typ(male,X), typ(person,Y).
```

If logical expressions are used to define predicate types, they tend to be rather simple. But in principle almost any sequence of goals can be used (e.g. [Nai92]).

With type definitions as logical rules and with types as regular sets it is not required that the universe of terms has predefined partitions as it is implicit in approaches using explicit type declarations for every function symbol. If function symbols have an explicitly declared type, then the universe of terms becomes structured and a model of a program is only allowed to use well-typed terms.

Sometimes a distinction is made between logical and extra-logical definitions of types. We think that this difference is not essential for our analysis because an extra-logical definition of types usually relates to some constraint theory over a special typed domain, which can be handled in a logical context. Nevertheless, there is a difference between syntactical type systems and semantical type systems. Syntactical typing is generally used for typed functional languages and also for several logic programming languages.

A difference between the syntactic and semantic typing styles will be illustrated by the example

```
pred pi(X) -> typ(list(integer),X).
pi(nil).

pred ps(X) -> typ(list(string),X).
ps(nil).

q1 :- pi(nil), ps(nil).

q2 :- X=nil, pi(X), ps(X).
```

Note that both clauses are logically equivalent. Given the standard polymorphic definition of lists with `nil` being the empty list which has type `list(t)` for any `t`, then the clause for `q1` is well-typed. But with many type systems the clause for `q2` is not considered as being well-typed because there is no type that can be assigned to the variable `X`. I.e., there is no type that is (subtype of) `list(integer)` and also (subtype of) `list(string)` at the same time.

In [Smo88] a special type $\perp$, denoting the empty set, is introduced, which is subtype of any other type. The clause for `q2` is then well-typed with `X` having the type `list(`$\perp$`)`. In several other type systems, where regular sets are used to describe types, the variable `X` has the type { `nil` } , such that the clause for `q2` is well-typed. There are no clear advantages either for semantical type systems or for syntactical type systems. The first allows more expressions to be well-typed while the latter allows to detect more type errors automatically.

For functional languages there is a similar problem which is handled by introducing a special kind of polymorphism, called let-polymorphism. Take the example

```
li: list(integer) -> integer  % type of li
ls: list(string) -> integer   % type of ls

let x = nil in pi(x) + ps(x)
```

where the variable `x` needs to be of type `list(integer)` and of type `list(string)`. A special 'let-polymorphism' typing rule allows the variable `x` to have different instantiations of the type scheme `list(T)` within the let-expression.

3 Types for proving partial correctness

In lambda-calculus there are syntactically well-formed expressions that have no well-defined meaning. E.g., the expression `2(1)` is undefined because `2` is not a function and can not be applied. There may be functions where arguments are restricted to some domain, e.g., arithmetic addition `add: int` $\times$ `int` $\rightarrow$ `int` such that the expression `add(aString1,aString2)` ist not defined. There may also be singularities in the domain of a function, e.g., division is not defined if the divisor is 0.

A typed lambda-calculus therefore defines well-typed expressions, which have the property that "well-typed programs don't go wrong." [Mil78]. If a function requires its arguments to be of some type, then in a well-typed program this function is never applied to arguments of different types. Well-typing, also called type-correctness, only partially guarantees well-definedness of expressions, e.g. division by 0 is generally not excluded by the notion of well-typing.

In contrast to lambda-calculus, in first-order predicate logic (PL1) every syntactically well-formed expression has a well-defined meaning, i.e., it is true or false. Therefore, we can expect the meaning of type correctness in logic programs to differ considerably from type correctness in other languages. Of course, there are overlaps. Logic programming languages often include functional expressions which can "go wrong" and also higher-order logic programming may have to deal with expressions that have no truth-value, e.g., if a term which is expected to be a predicate, actually is bound to an integer constant.

Looking for a type checking method for logic programming one could try view each predicate as a function $fp : some\_domain \rightarrow boolean$ and then apply type checking methods well-known from typed λ-calculus and its extensions. If logic predicates were only used to check a relation between given ground terms, then this way might do the trick, but logic programming is more versatile. Having some bottom-up calculus for Horn clauses in mind, the predicate may even be taken as a function $fp \rightarrow set\_of(some\_tuple\_type)$. Actually it is the declarative non-directional character of logic programming which makes finding an appropriate type checking scheme difficult.

Nevertheless, having a program extended with type declarations we might want to check if a given calculus will always compute results that correspond to the type declarations. I.e., we assume a program which has a specified semantics and a calculus that is correct and complete with respect to these semantics. If a program implies solutions that do not conform to the type declarations, then we can say that the program contains a type error, otherwise it is well-typed. Unfortunately there is a wide range of possible definitions for type-correctness or well-typing.

Just to give an example how these approaches differ, we take a look at the standard Prolog definition of the predicate `append`

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

and simple clauses such as

```
q1 :- append(3,X,[]).
q2 :- append([],X,3).
```

With many type systems the clause for `q1` will be marked as being ill-typed, because `append` only takes lists in the first argument. However, the clause for `q2` is sometimes taken as well-typed. Actually, the clause succeeds. In order to express the intention that `append` is only to be used with lists, many type systems allow predicate declarations, such as

```
pred append: list x list x list.
```

Using these additional declarations, the clause for `q2` can be detected as ill-typed. Curiously, there are also type systems that use similar predicate declarations but that take the clause for `q1` as being well-typed.

If we use prescriptive typing as in section 4, possibly with some automatic type inference, and also have a calculus that will handle all type constraints, then in principle every program is well-typed, because the calculus will eliminate ill-typed bindings within its constraint solving. This extreme view can be found in [HS88].

If we use a much simpler calculus, say resolution with untyped unification, then well-typing of a program can only be ensured for rather restricted type systems; e.g., parametric polymorphism with no subtyping [MO84, LR91]. Unfortunately, due to restrictions in the type system, well-typing can only be shown for a restricted class of programs, i.e., meta-programming and extra-logical features are difficult to handle or forbidden.

Naish defines a special kind of type correctness for programs in combination with untyped resolution [Nai92]. It has an application for declarative debugging, but the type correctness of a program is not decidable.

Currently, mode dependencies or directional types are getting much attention. Besides their use for type checking, reasoning about termination and optimizations can benefit from using directional types, e.g. [BLR92, AL94, AE93, BM95]. Predicates are seen as transformers taking input arguments of some type and producing output argument of some possibly more refined type. For example, the append-predicate defined as

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

can have the directional types (using notation from [AL94])

$$\begin{array}{rcl} (List(1), List(1), 1) & \rightarrow & (List(1), List(1), List(1)) \\ (1, 1, List(1)) & \rightarrow & (List(1), List(1), List(1)) \end{array}$$

expressing that `append` can be used when the first two arguments are lists of any terms, or if the third argument is a list of any terms. Directional types are closely related to SLD-resolution with a fixed selection strategy for subgoals, e.g., left-to-right as in Prolog, because the analysis of a binding status for a variable, similar to a mode, is mixed with the type analysis. This makes directional types difficult to generalize, e.g. for deduction systems. They do not fit well with general logic specifications, because they impose a functional view on logic programs. Although in practice Prolog programs are deterministic to a great extent, non-directionality is one of the features which make logic programming different and attractive. Just as a simple example we might look at the definition for the last element in a list:

```
last(L,X) :- append(_,[X],L).
```

If a type declaration for `append` requires the first argument always to be a sufficiently instantiated list, then this perfectly logical definition would not be well-typed.

There are also type systems where type correctness can be defined but there is no effective decision procedure. E.g., if the type definition includes the types `animal` and `plant` we may want to decide whether the conjunction `animal(X)` $\wedge$ `plant(X)` is satisfiable. If this decision depends on the program part, type checking becomes undecidable in general. A type system where the notion of well-typing depends on the model of the program part is used, e.g. in [KW90].

This list of varying notions of type checking is not complete. Often type checking or program analysis is used to show that a certain condition will hold at some state at run-time. This information can then be used to optimize the calculus. This type analysis should not be confused with type checking methods that find programming errors. Although type information can be exploited very well for code optimization, types are not essential with this respect. Information about the (structure of) values expressed in terms of types is only a part of the information used for code optimization. This view on the secondary role of type for optimization can also be found in the work on compiling untyped object-oriented languages ([CUL89, CU90, HCU91]). The primary role of types is that the programmer wants or needs some condition to be true in order to ensure a well-defined semantics of the program or language in general. The role of types for optimization purpose is secondary because the compiler *may* use type information. The compiler can also do own code analysis independent from types to get almost the same results. Of course, if the type information is given explicitly in a program, then static analysis is much easier or possibly becomes tractable in cases where an analysis of an untyped program would be intractable or undecidable.

One observation is that the notion of well-typing, resp. what one wants to be a type error, is almost deliberately defined. Given a specific definition for type-correctness, type checking will ensure that some kind of error (as defined) will not occur. We do not examine further details, because the use of types for checking partial correctness is not directly linked to the other dimensions of using types. Nevertheless, it is possible and in fact useful to combine the various techniques.

4 Types as constraints

In logic programming languages with a prescriptive type system, the semantics of a predicate is generally defined by two components: the type declaration and the set of clauses. Often, type declarations induce type constraints to be part of the clause.

Several authors use logical expressions as type declarations for predicates, such as

$$\textit{predtype} \quad p(X_1,\ldots,X_n) \rightarrow \exists Y_1,\ldots,Y_k\ F$$

where F is a formula over the variables X_i and Y_i. Any (definite) clause

$$p(Z_1,\ldots,Z_n) \leftarrow \textit{Body}$$

is then automatically transformed into the clause

$$p(Z_1,\ldots,Z_n) \leftarrow \textit{Body} \wedge \theta(\exists Y_1,\ldots,Y_k\ F)$$

where $p(X'_1,\ldots,X'_n)$ is a variant of $p(X_1,\ldots,X_n)$, and θ is the most general unifier of $p(X'_1,\ldots,X'_n)$ and $p(Z_1,\ldots,Z_n)$. The variables Y_i are renamed to new variables if needed. The semantics of the original program with predicate type declarations is then defined to be the semantics of the program consisting of the set of all transformed clauses. This method is used, e.g., by [Nai92, XW90].

Another method for defining the semantics of a typed program assumes that all variables appearing in the clauses are explicitly typed, i.e., they have an associated type constraint. The universe of terms is usually given by a typed structure. The semantics of the set of clauses (without predicate type declarations) is then defined by using a typed algebra. This method is used by [Han92]. Of course, the typed clauses must conform to the type declarations.

In general, in a well-typed program an atom is true if it is a consequence of the clauses and if it conforms to the type declaration. The approaches that allow type constraints within clauses, can be considered as instances of the general Constraint Logic Programming scheme where the constraints range over a typed domain [HS88].

Because explicitly defining the type constraint for every variable is a tedious task for a programmer, many authors apply a so-called type inference procedure to a program where not all variables have an associated type. Using type declarations and partially untyped clauses, this procedure constructs a fully typed program for which the semantics can be defined analogously to the previous method. Type inferencing as a preprocessing step for the definition of the semantics is used, e.g., in [Smo89, HT92]. For the following clauses

```
pred parent: person x person.
  parent(P,C) :- mother(P,C).  % P:female, C:person
  parent(P,C) :- father(P,C).  % P:male, C:person.
```

the variable typings as given in the comment can be automatically inferred by a type inferencing procedure.

In the following we want to show that there are several flaws and problems within the "types as constraints" proposals.

1. Although using a similar terminology as for typed procedural and functional languages, the notions of well-typing and type-error have other meanings.
2. The semantic consequences of type inferencing are not well understood.
3. It is debatable if type constraints over a system with parametric polymorphism are practically useful.

A weakness of constraints over parametric types and subtypes, given in section 5.3, also has a negative effect on the expressivity of type constraints. Furthermore it appears that the notion of well-typing, used by approaches that allow types as constraints, is closely related to our notion of useless expressions. Well-typing conditions can be seen as rules for detecting useless clauses combined with several additional requirements that depend on the specific properties of the system.

4.1 Prescriptive typing with a typed calculus

A rather general approach to a typed logic programming using prescriptive semantics can be found in the proposal by Höhfeld and Smolka [HS88, Smo89]. Their language allows definite relations over general constraint expressions. The type of a relation can be defined using some logical constraint expression. They provide a notion of well-typing and also define a type inference procedure. Interestingly, the type inference algorithm will transform any syntactically well-formed clause into a well-typed clause. This is because "for technical reasons" they do not require constraints resulting from type declarations to be satisfiable. Of course, for specific instances of that general scheme the definition for well-typed clauses may require the satisfiability of type constraints, and this is actually done in virtually all proposals that use prescriptive typing, including [Smo89].

Usually, these proposals are accompanied by typed resolution and the well-known slogan "Well-typed programs don't go wrong" is carried over to well-typed logic programs. In general, this only means that the typed resolution does *dynamic* type checking. For typed functional programming, this is the context where Milner coined the slogan, well-typing means that no type-clash will occur at run-time. For logic programs with a simple type system of disjoint sort and parametric polymorphism, as in [MO84, LR91], static type checking can also ensure, that no type errors occur at run-time. For typed logic programs in general, well-typing only means that every type-clash which is not already detected statically will be detected dynamically at run-time, and if a type-clash occurs at run-time the resolution will treat it in the same way as any other unification failure. We will look again at our previous example, slightly modified by leaving out one rule.

```
pred parent: person x person.
  parent(P,C) :- mother(P,C).  % P:female, C:person
```

The clause in this example, using the variable typings as given in the comment, is usually considered as being well-typed. We can pose the query `?- X:male, parent(X,Y).`, i.e., with `X` having the type `male`. The typed resolution, assumed to implement the required order-sorted unification, may try to resolve the query with the first clause of `parent`. Because the type `male` of `X` and the type `female` of the corresponding variable `P` in the head of the clause do not have a common subtype, the unification will fail. This is an example where dynamic type checks occur in typed logic calculi.

Now consider an extension to the program by adding

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

The clause for `q1` is usually considered as not well-typed for the same reason that inhibits the resolution of the previous query with the first clause of `parent`. The type constraints for the variable `X`, i.e., `X:male` and `X:female` are not satisfiable. Because this can be seen statically, instead of dynamically as in the query before, the clause is called not well-typed. This type checking method is very similar to our method for detecting useless clauses. Even if we would call the clause for `q1` well-typed and use the whole program, the semantics of the program would not change. The body of the `q1`-clause would always fail. So, in this case, saying that a clause is not well-typed, is just another way of saying that, using the typed calculus, the clause will produce no solutions.

As a first summary we can see that for typed logic languages often the same terminology is used as for typed functional languages, but the meaning may be basically different. There can be a substantial amount of type computations at run-time for statically type checked logic programming languages. Reconsidering the slogan "well-typed programs don't go wrong", for prescriptively typed logic languages with a typed resolution calculus, that goes beyond the simple case with flat disjoint types and polymorphism, we might say "well-typed logic programs *fail* instead of going wrong". I.e., type errors are replaced by type failures.

4.2 Semantic impacts of type inference

There is intensive research on type inferencing, mostly for functional programming languages, but also for procedural and logic languages and their object-oriented variants. Although complete type inferencing is undecidable in general there are many useful results and practical applications. When type inferencing techniques, e.g. developed for functional languages, are applied to logical languages some care has to be taken due to semantical consequences of automatically inferred types. Neglecting tricky issues such as overloading, run-time type inspections, etc., the semantics of a functional expression is generally independent from its type. Type inference does not affect the semantics of a functional program. But type inference for logic programs has the (commonly ignored) danger, that it may affect the semantics. This problem is explained by the following example.

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 2

Declarations

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

pred e:  T x T.
pred q:  T.
```

Clauses/Facts

```
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. (Usually, no formal semantics are defined for 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 the following example showing that even obeying these conditions does not rescue the situation.

Example 3

```
% the type order
t1 =< t9.  t2 =< t9.     %                      t9
t0 =< t3.  t1 =< t3.     %                     /  \
t0 =< t4.  t2 =< t4.     %              t3     /  \     t4
                         %               \ \  /    \  / /
% constants              %               \   t1    t2   /
c2 -> t2.                %                 \          /
c1 -> t1.                %                   \      /
c0 -> t0.                %                      t0

pred l2: list(T).
  l2([X,Y]).

pred p: t9.
  p(c1).
  p(c2).

% goal  ?- p(X), l2([c0,X]).
```

If two types in this hierarchy have a lower bound then they have a greatest lower bound. The type hierarchy can be extended with a least type such that it becomes a lower semi-lattice. Such a type structure is allowed, e.g., in [Smo89, HT92]. The goal is well-typed with the variable typing $VT1 = \{X : t1\}$. Assuming a correct order-sorted unification, the only solution for X respecting $VT1$ is $X = c1$. But the goal is also well-typed with the variable typing $VT2 = \{X : t2\}$. The only solution for X respecting $VT2$ is $X = c2$. There is no variable typing such that the goal is well-typed and also allows both the bindings $X = c1$ and $X = c2$.

The consequence is: For CLP over typed theories with parametric polymorphism and subtyping type inferencing for variables can have semantical impacts. For our example no unique variable typing that allows all well-typed solutions can be inferred.[1]

This problem might be rectified by imposing further restrictions on the allowed type hierarchies. Also we could devise a formalism for well-typed clauses that

[1] Semantical impacts of types are also apparent in functional languages, where a well-typed term may have different derivations. This is known as the coherence problem, see e.g. [BTCGS91, Jon92]. Part of this problem is commonly prevented for typed logic programming languages by requiring the so-called transparency of function symbols. A close investigation is beyond the goal of this paper.

not only takes variable typings but also constraints between types into account. Actually this was done for functional programming languages almost from the beginning [Mil84], but to our knowledge, there is no comparable proposal for logic languages with parametric polymorphism and subtypes.

We don't want to elaborate on a new formalism for well-typing using type constraints in clauses for now, because there are further practical problems that make these approaches difficult to apply.

4.3 Efficiency considerations

Constraint programming is a very active research area and there are promising results and applications. Therefore, we can expect that there are also benefits in using a special typed domain within a constraint theory. Indeed, for hierarchically ordered sorts, i.e. types, Walther showed efficiency gains that are possible in deduction systems using order-sorted unification [Wal89]. Sort declarations for logical predicates are very similar to type declarations used in other programming languages. Therefore, it is reasonable to extend sorted deduction with more versatile constraints by including parametric polymorphism. Efficiency problems appear when it comes to implement an extended type constraint solver. For certain type systems it is possible to find several optimizations ([Smo89]), which can also be realized in an Abstract Machine ([BM94]). Our experience with large systems implemented in a logic language with polymorphism and subtypes (e.g., production planning systems implemented with the PROTOS-L system [BBM94, MB94]) shows that there is seldom any use of complex type constraints, such as `X:list(pair(nat,negint))`, other than making a clause statically well-typed. I.e., they are rarely used as active constraints. Instead, structured type terms very often occur in type declarations describing the data structures expected for arguments, and they are exploited within static type checking. Unfortunately, these declarations implicitly impose type constraints on terms which are hard to optimize and they are checked in the run-time system, possibly slowing down execution considerably. Because type constraints are automatically inferred, the programmer is often not aware of the location where type constraints have to be checked at run-time.

The general symptom is that an expressive type system is very useful for static type checking to locate programming errors. But when type declarations are inherently coupled with type constraints that have to be checked in the run-time system there may be a considerable overhead.

5 Types as approximations

We will now concentrate on the "types as approximations" dimension, which fits well with declarative programming. While the general scheme we present here is new, specific instances can be found as part of many proposed type systems, although often it is used only indirectly.

For the beginning we will allow a rather general form of predicate declarations. They may be logical or extra-logical expressions. 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.

If predicate type declarations use logical formulas the result of tc usually is an instance of the declaration with some unification applied to it. If the type declaration is given with type terms, the constraints can be generated by some additional transformation. Actually this is done within a type checking component of a compiler, which also solves these constraints. The function tc lets us abstract from these details.

Intuitively, the declaration is an approximation of the set of atoms $p(\ldots)$ which are true in the *intended* model. I.e., the model intended by the programmer is described in two ways. First by the logical clauses and second by the predicate type declarations. Possible semantical effects of type declarations are inspected at more detail in section 4. Of course, in practice the predicate declarations are much simpler than the clause specification and they only roughly approximate the intended meaning of a predicate.

For static type checking purposes, the declarations can be used 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 declarations 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.

5.1 Ill-typed clauses

If predicate type declarations use logical formulas the result of tc usually is an instance of the declaration with some unification applied to it. If the type declaration is given with type terms, the constraints can be generated by some additional transformation. Actually this is done within a type checking component of a com-

piler, which also solves these constraints. The function tc lets us abstract from these details.

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 **useless**, **ill-typed**, or **not type-consistent**. These notions can be used interchangeably. 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 where the program part consists of definite clauses and `is_integer` has the usual intended meaning.

```
type p(X) -> is_integer(X).  % type declaration
p(b).                        % conflict with the type declaration
p(1) :- p(a).                % body is always false
```

Whereas the second clause is semantically useless, i.e. in the intended semantics implied by the type declarations, `p(a)` is not true, the first clause may be called 'practically useless', i.e. it should be eliminated. Usually there is no reason for a programmer to write such a clause having an unsatisfiable type constraint, in this case `is_integer(b)`. Instead it is highly probable that the clause is not designed or written correctly.

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 $tc(A)$ 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.

If a program consists of formulas in first-order logic, we could translate it into a formula where every atom A is replaced by $tc(A)$. If the transformed formula is not satisfiable, we have shown, using our basic assumption, that the original formula is not satisfiable in the intended model of the program. Hence, the original formula probably contains an error. As we will show in the following, in practice we can make a more detailed analysis of program clauses exploiting predicate declarations.

5.1.1 Horn clauses

First consider a program which is given by Horn clauses. If there is a simple fact `p(...)` and tc_p(`p(...)`) 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. At least, it contains a programming error with respect to the predicate declaration. A similar line of reasoning can be applied to clauses of the form $\{A, \neg B_1, \ldots, \neg B_n\}$, or written as a rule

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

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). The whole formula is a tautology in that case, 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 [Mis84]. 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.

5.1.2 Non Horn clauses

This simple technique is also applicable to non Horn clauses. Then we have a clause of the form $\{A_1, \ldots A_m, \neg B_1, \ldots, \neg B_n\}$, or written as a rule with disjunctions ';' in the rule's head as

$$A_1; \ldots; A_m \leftarrow B_1, \ldots, B_n.$$

Using the same arguments as for Horn clauses, we require the type constraint expression $(tc(A_1) \vee \ldots \vee tc(A_m)) \wedge tc(B_1) \wedge \ldots \wedge tc(B_n)$ to be satisfiable. In practice we place an even stronger requirement on such a clause. Consider the example

```
is-person(P) ; is-female(P) ← is-male(P).
```

where the type constraints of an atom `is-x` requires its argument to be of type `x`, referring to the types from the introductory example. The type constraint expression as above is satisfiable for this clause. But we can say that the atom `is-female(P)` in the head of the rule is redundant, because the type declarations express that `is-female(P)` and `is-male(P)` can not be true at the same time, i.e., with the same variable binding. Hence, we can sharpen our conditions on general clauses by requiring that

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

must be satisfiable for each head atom A_i. This also ensures that the type constraints for each single atom in the clause are satisfiable.

5.1.3 Negation

Handling negation, or negation as failure as it is prominently used in Prolog, is similar to the technique used for general clauses. Consider an extended Horn clause

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

If $tc(C)$ is not satisfiable then `not` C will always 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. Seen purely logically, a negated atom in the body of a rule corresponds to a positive atom in a disjunctive head. Compared with the requirements on disjunctive rules in 5.1.2 above, it is notable that we put a stronger requirement on clauses with negation as failure which is due to the special importance of the (single) head atom.

If there is more than one negated atom in an extended Horn clause, i.e. we have $A \leftarrow B_1, \ldots B_n, \texttt{not}\ C_1, \ldots, \texttt{not}\ C_k$. the treatment is similar to disjunctive rules. 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 our running example in section 2:

```
pred p: person.
  p(P) :- know(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 would require 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 [HT92], which view this clause as not 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 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.

This notion of ill-typing does not preclude standard Prolog idioms, such as the failure driven loop, e.g.

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

assuming that `p` and `side_effect` constrain arguments to be of the same type and the type constraints of the other 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`.

5.1.4 First summary

Our notion of ill-typing has several advantages. It is independent of the operational semantics in contrast to many other definitions that rely on or are tailored towards SLD-resolution. It enables type checking to proceed clause by clause. Also there is no need to compute the model of the program part which would make type checking undecidable even if the type language is simple. This is important if separate compilation and type checking of modules is considered. Furthermore, we require the static analysis to be correct but there is no need for requiring completeness for these checks.

5.2 Sufficient vs. necessary type constraints

So far we have always assumed that predicate type declarations express a *necessary* constraint on atoms. I.e., the satisfiability of the type constraint $tc(A)$ is necessary for A to have instances in the intended model. This correspondence was used to detect expressions that are always false. The treatment of negation has shown that there are also cases where expressions that can be statically detected to be true are regarded as useless. Hence, we may also use another form of type declarations which express *sufficient* conditions for atoms to be true. Actually there are cases where such declarations make sense.

Consider a binary predicate with some type declaration

```
pred p: t1 x t2.
  p(X,Y) :- ...
```

We could also define the complement of the predicate `p` as

```
  q(X,Y) :- not p(X,Y).
```

By our basic assumption about type declarations, `t1 x t2` in the declaration of `p` covers the success set for `p`, i.e., it somehow describes a superset of the set of atoms `p(X,Y)` intended to be true. We can not directly use the same description `t1 x t2` in a type declaration for the predicate `q`, because we have to describe the complement of the success set of `p`. Instead we can say: if a constraint that is necessary for an atom `p(X,Y)` is not satisfiable then `p(X,Y)` always fails and we know that the atom `q(X,Y)` will be true. We can express this with some kind of inverted or negated type declaration, e.g.

```
pred-neg q: t1 x t2.
```

Then a clause where an atom `q(X,Y)` appears in the body, will be treated within the ill-typing constraints as an atom `not q'(X,Y)`, where `q'` is assumed to have the type declaration `q': t1 x t2`. There are many cases, where complementary predicates are defined in logic programs, e.g. equality and disequality, or predicates that define disjointness and the complementary overlap for sets. See [Mey96] for an application of this technique within a specific type system.

5.3 Handling parametric polymorphism

Parametric polymorphism is known to be very useful for expressing the types of generic predicates. If it is used in type declarations for logical predicates without further adaptations, we will see that in many cases the chances to detect a clause as ill-typed will be missed, even for clauses where ill-typing is almost obvious.

First, consider an extension to the example program by adding

```
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 simple parametric type declaration for equality, i.e., `'=': T x T`, the predicate

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

although logically equivalent to `q1`, is not detected as ill-typed. 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`. Furthermore, with many other type systems this clause is perfectly well-typed, e.g., [Smo89, HT92]. This anomaly seems to be generally neglected in the literature; [Wu92] is a notable exception mentioning the weakness in type checking, which is caused by the generally used method for combining subtypes and parametric polymorphism.

In [Mey96] we present a refined method for type checking within a type system that includes parametric polymorphism. As part of our solution we propose a type system that allows more exact declarations such as

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

The expressions on the right of `|>` describe type constraints where `=<` stands for the subtype relationship.

First assume that the variable `L` has the (least) type `list(person)`. Given the least type of the first argument `[bob]` as `list(male)` and the least type of `[sue]` as `list(female)`, the type constraints `male =< person` and `female =< person` are satisfied, and an atom `append([bob], [sue], L)` is well-typed. But if the variable `L` is constrained to the type `list(male)` the same atom is ill-typed, because there is no type `T2` such that `[sue]` has (least) type `list(T2)` and also `T2 =< male`. This indicates that the atom as a goal literal will always fail, because with the intended meaning there is no list `L` of `male`-terms that also contains `female`-terms.

Using a refined type declaration for the predicate `'='` such that the generated type constraint requires the least types of the terms to be equal, then also the clause for `q2` can be detected as ill-typed.

These examples show that it is possible to give very detailed predicate type declarations even for generic predicates. In general, the more specific the predicate type declarations are, the more programming errors can be detected.

6 Conclusions

Although our investigation may look a bit destructive, there is actually a very practical way to use it. We have seen that there are several problems with systems that use a prescriptive form of typing and type constraints in the calculus. Furthermore, applying functional or directional typing schemes has advantages when it comes to prove a program to be free of certain kinds of type errors, but there are restrictions on the programming style and these schemes are difficult to generalize to a broader class of logic calculi. On the positive side we see our general technique of detecting useless formulas (s. sections 5.1 and 5.3) as very promising. The static program analysis can be more precise than possible with several other general proposals, without restricting the language or programming style.

Furthermore, we believe that type declarations should not be part of the program semantics. I.e., we prefer to distinguish between a program part for which the semantics can be completely specified and an additional part with type declarations or annotations in general. The latter is then used to check the consistency or even partial correctness of the program part. Instead of using a (correct and complete) typed calculus which *makes* every solution well-typed and simply fails on others, we want to use type declarations to *check if* a solution is well-typed. The more precise the type declarations are, in the sense of types as approximations, the more errors can be detected by a procedure that finds useless clauses. Unfortunately, with more specific declarations, proving that some calculus applied to some program will always produce results that conform to these declarations becomes more difficult or even impossible due to undecidability problems. There is a similar situation for procedural programs that contain annotations, such as pre- and postconditions and invariants. Although we might not be able to prove that these conditions will always hold at run-time, these annotations will help very much in the process of finding programming errors.

Type declarations are seen as special program annotations. The first application of these annotations is to find program expressions which are *useless*. The detection of useless expressions is independent of any inference calculus and only relies on the logical meaning of clauses and declarations. Requiring correctness but not necessarily completeness of a program checker also allows to apply it to a very wide range of programs, e.g., standard Prolog with no restrictions. If a programming error, in our case a type error, can be found by static analysis, the error can be shown more or less exactly to the programmer.

The second use of type declarations is somewhat more related to a calculus. Here we want to find preconditions under which the specific calculus will only produce results that are well-typed. Because the dimensions of using types in logic programming, i.e. types as approximations, types as constraints, and types for type checking can be taken as separate issues, the conditions used for a complete type checker even do not necessarily have to be the same as for declarations that are exploited to find useless expression.

For specific type systems, e.g. for Horn clauses with parametric polymorphism but no subtyping, the detection of ill-typed clauses can be combined with the check for the genericity condition. It is then equivalent to the type checking method that ensures type correctness with untyped unification as in [LR91].

For the type system which was briefly presented in section 5.3 we are looking for an enhancement that allows to prove type-correctness of a program in some cases, even if untyped unification as in Prolog is used. This extension uses context conditions of the following kind. The type correctness of a predicate may depend on some context condition, if the predicate is used in a context where the conditions are ensured, then the implementation of that predicate is type-correct. Context conditions are similar to directional types but do not require a specific evaluation method.

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.

[BBM94] C. Beierle, S. Böttcher, and G. Meyer. Report of the Logic Programming Language PROTOS-L (revised version). Working Paper No 4, IBM Germany, Scientific Center, Inst. for Logics and Linguistics, Heidelberg, July 1994.

[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 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.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–222, July 1991.

[CU90] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proc. of the ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 150 – 164. acm Press, June 1990. published as SIGPLAN Notices Vol.25, No.6.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. In N. Meyrowitz, editor, *OOPSLA '89 Proceedings*, pages 49–70, Portland, Oregon, September/October 1989. acm Press. Special issue of SIGPLAN Notices, Vol.21, No.11, Nov.1989.

[DZ92] Philip W. Dart and Justin Zobel. Efficient run-time type checking of typed logic programs. *The Journal of Logic Programming*, 14:31–69, 1992.

[Han90] Michael Hanus. Logic programs with equational type specifications. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming: Proc. 2nd Int. Conf., Nancy, France*, volume 463 of *LNCS*, pages 70 – 85. Springer Verlag, October 1990.

[Han91] Michael Hanus. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

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

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, pages 21–38, 1991.

[HJ92] Nevin Heintze and Joxan Jaffar. Semantic types for logic programs. In Pfenning [Pfe92], chapter 4, pages 141–155.

[HS88] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. IWBS Report 53, IBM Scientific Center, Institute for Knowledge Based Systems, Stuttgart, October 1988.

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

[Jon92] Mark Philip Jones. *Qualified Types: Theory and Practice.* PhD thesis, Oxford University, July 1992. Cambridge University Press, 1994, (distinguished dissertation series).

[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.

[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.

[MB94] G. Meyer and C. Beierle. PROTOS-L Users's Manual. Working Paper No 5, IBM Germany, Scientific Center, Inst. for Logics and Linguistics, Heidelberg, July 1994.

[Mey96] Gregor Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Informatik Berichte 200, 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.

[Mil84] Robin Milner. Coercion and type inference. In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, Salt Lake City, Utah, January 1984.

[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.

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

[MR85] P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of the Annual ACM Symposium on Principles of Programming Languages*, pages 7–21, January 1985.

[Nai92] Lee Naish. Types and intended meaning. In Pfenning [Pfe92], chapter 6, pages 189–216.

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

[Wal89] Christoph Walther. Many-sorted inferences in automated theorem proving. In K. H. Bläsius, U. Hedtstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence: Proc. Workshop, Eringerfeld, Germany*, volume 418 of *Lecture Notes in Artificial Intelligence*, pages 18 – 48. Springer Verlag, April 1989.

[Wu92] Jwo-nan James Wu. *A First-order Theory of Types and Polymorphism in Logic Programming*. PhD thesis, State University of New York at Stony Brook, 1992.

[XW90] Jiyang Xu and David S. Warren. Semantics of types in logic programming. DPS-102, ECRC, Munich, March 1990.