Polymorphic Type Checking with Subtypes in Prolog

Thom W. Frühwirth*
Department of Computer Science, SUNY at Stony Brook
Long Island, NY 11794-4400, USA

Abstract
In this paper, we give an executable specification of a state-of-the-art polymorphic type checking system with subtypes in Prolog. We show that the implementation reduces to modifying simple well-known meta-interpreters into type meta-interpreters and applying the so-called generate-and-test approach to programming in Prolog. This study emphasizes that Prolog is suitable as a language for defining executable specifications and that Prolog can be augmented with a simple but powerful type system. The type language augmenting Prolog programs with type definitions and type declarations is a subset of Prolog itself. Therefore it is possible to use types explicitly in a program.

1 Introduction: Types for Prolog
It is generally agreed that type information supports debugging by detecting type inconsistencies as in [Mycroft/O'Keefe], program verification [Kanamori/Horiuchi], program documentation [Bruynooghe] and optimized compiling [Xu/Warren]. [Mycroft/O'Keefe] introduced type checking in Prolog, based on the work of [Milner] for the functional language ML. In such declaration based systems, the user adds typings for function and predicate symbols to the program. The typings are considered as presuppositions on the meaning of a predicate, they do not necessarily cover the denotation of the predicate. This is a very useful feature, as Prolog predicates tend to be 'under-specified' for the sake of simplicity and flexibility.

In the following example of our type checking system type definitions for functions are given as clauses for the distinguished binary predicate ':'. Type declarations for predicates are considered as type definitions for the distinguished type 'pred'. The complete program for append(L1,L2,L3), which succeeds when the list L3 is the result of appending the list L2 to list L1 is given as:

[] : list (A).
append(X,Y,Z) : pred:- X:list(A), Y:list(A), Z:list(A).
append([],L,L).
append([X|L1],L2,[X|L3]): - append(L1,L2,L3).

The predicate append/3 is declared as accepting arguments of type list(A). The type clauses for the polymorphic type list(A) state that the empty list [] is of type list(A) and that the term [X|L] is of type list(A) if X is of type A and L is of type list(A). This means that all elements of the list have a type A, which is left unspecified as a type parameter.

The possibility to define such parameterized types is called parametric polymorphism. Polymorphism refers to the fact that a variable, a function symbol or a predicate symbol is associated with multiple types [Wegner/Cardelli '85]. Inclusion polymorphism allows the specification of subtypes, and additive polymorphism makes it possible to define a type as union of other types. Only some recent proposals deal with all three kinds of polymorphism [Dietrich/Hagl '88]. Polymorphism can significantly reduce the loss of flexibility due to enforcing a type discipline in type-free languages. Generally speaking, parametric polymorphism is more easier handled in type checking systems, while inclusion and additive polymorphism is a natural option in type inference systems.

A program can be checked to see if it is well-typed with respect to the type declaration. [Mycroft/O'Keefe] adopted the slogan of [Milner] that "well-typed programs do not go wrong". The notion of "wrong" is independent of success, failure or looping of a query. For example, the query append([X],[Y],[Z]) is well-typed but fails, the query append([],a,a) is ill-typed but succeeds, the query append([X|L1],[L2],[L1]) is well-typed but loops. The usefulness of the notion of well-typing stems from the fact that if a program and a query are well-typed, then variables in the query can be instantiated only to terms allowed by their types. This follows from the theorem proven in [Mycroft/O'Keefe], that one step of

* Current Address: Technical University of Vienna, Comp. Science, Paniglgasse 16/E181B, A-1040 Wien, Austria
Research was partially supported by a grant from the Austrian Fulbright Commission and by a grant from the Austrian Bundeskammer der Gewerblichen Wirtschaft
SLD-resolution will take a well-typed resolvent into a new well-typed resolvent. Thus run-time type checking is unnecessary. Unfortunately, this theorem does not hold in general anymore when subtypes are added, as we will do later. For an in-depth discussion of this topic see [Dietrich/Hagl].

In the rest of the paper, we define a polymorphic type language for Prolog. Introduce three basic meta-interpreters, show how meta-predicates can be well-typed, answer type queries with type meta-interpreters, check the consistency of type answers, type-check programs by well-typing their clauses and discuss related work. Some familiarity with the practical [Sterling/Shapiro] and theoretical [Lloyd] aspects of Prolog is assumed.

2 A Polymorphic Type Language for Prolog

When adding a type system to a (logic) programming language, it is desirable not to destroy the semantics of the language, i.e. the type system should fit into the semantics. By using a type language equal or similar to the programming language to be extended, a consistent embedding can be supported. In addition, this makes it easier to deduce, reason about and utilize type information. Hence in our case, the type language is a subset of Prolog itself.

Terminology Types are defined by a binary predicate with the distinguished predicate symbol `\':\', which is written as infix operator. The type atom $t:p$ means that the term $t$ has type $p$. We also say that the type of $t$ is $p$. A type clause is a clause whose head is a type atom and whose body is a type goal. A type goal is either a type atom, a conjunction of type atoms or a disjunction of type atoms. A type clause with head $t:p$ is called clause for type $p$ typing $t$. A type $p$ is defined by the finite set of clauses for type $p$. A type term is the term denoting type $p$. A type term is either a type parameter or a type constructor of arity $n$ applied to $n$ type terms as arguments. A type parameter $A$ is a variable. A typed term is the term $t$ in a type atom $t:p$. A typed term is either a type argument or a function symbol of arity $n$ applied to $n$ typed terms as arguments. A type argument $X$ is a variable. Type parameters and type arguments are taken from disjoint sets of variables. A general type term is a type constructor of arity $n$ applied to $n$ type parameters as arguments. A typed logic program is a logic program augmented with type clauses.

Now we can define type definitions for function symbols and type declarations for predicate symbols.

Definition A type definition for a function symbol $f/n$ is a type clause typing $f/n$. It relates the type of $f/n$ to the type of its $n$ arguments:

$$f(X_1,X_2,\ldots,X_n):=X_1:P_1,X_2:P_2,\ldots,X_n:P_n.$$  

where $p$ is a general type term, each $p_i$ is a type parameter or a general type term, each $X_i$ is a type argument and $n\geq 0$. If $n=0$ then the type clause is a fact defining the type of a constant. There are no local type parameters, that is, each type parameter occurring in a body of a type clause also occurs in its head. There is no principal problem with allowing local type parameters, but they would make type-checking less precise as they hide type information. For example, if the type of the elements of a list is local, there is no way to (type-)check if two lists operate on elements of the same type.

We require that each symbol is typed by exactly one type clause. In this way, our type systems disallows overloading. If the same function symbol should be shared by different types, subtypes can be used.

Through subtypes we now introduce additive and inclusion polymorphism. A type is either defined by a finite set of type definitions or by a single subtype definition. In practice, this restriction can be relaxed, as types violating it can be automatically transformed to a subtype definition by introducing new subtypes.

Definition A subtype definitions for a type $p$ is a type clause defining $p$ as the union of subtypes $p_i$:

$$x:p:=x:p_1\cup x:p_2\cup \ldots \cup x:p_n.$$  

where `\cup` is an infix operator denoting disjunction, $p$ and each $p_i$ are general type terms, $X$ is a type argument and $n\geq 0$. If $n=0$ then the type clause is called an alias for type term $p_1$. We call each $p_i$ a direct subtype of $p$. A type $q$ is a subtype of a type $p$ if $q$ is a direct subtype of $p$ or $q$ is a direct subtype of $q_1$ and $q_1$ is a subtype of $p$. In other words, the subtype relation is the transitive closure of the direct subtype relation. In order to avoid circular definitions, i.e. that a type (or a instance of it) is defined as subtype of itself, and overlapping subtypes, i.e. that subtypes have intersecting domains, we require that $p$ and its subtypes $p_i$ are pairwise nonunifiable.

Definition A type declaration for a predicate symbol $p/n$ is a type clause for the distinguished predicate type `\pred':

$$p(X_1,X_2,\ldots,X_n):\pred:=X_1:p_1,X_2:p_2,\ldots,X_n:p_n.$$  

where each $p_i$ is an arbitrary type term and each $X_i$ is a type argument. Notice that type declarations are similar to type definitions. This makes it particularly easy to give types for meta-programs, i.e. programs which manipulate other programs as data. We will show at the end of the next section.

In the following we define two basic operations on types we will need for type-checking - namely intersection and union. For a more detailed discussion see [Frühwirth-3].

---

1 Originally [Frühwirth-2] the type predicates were defined by an extension of Prolog called HiLog [Chen/Kifer/Warren].

2 Of course this could be any other predicate symbol as well provided it is not used otherwise.
Definition: Given a set of types for the same typed term. We define the intersection of types to be the set of common subtypes which includes for each pair of types
   a) the result of their successful unification whenever possible
   b) the intersection of their direct subtypes otherwise.

The set of minimal subtypes of types is the minimal subset of the intersection of types such that each type in the intersection of types, i.e. each common subtype, is either
   a) contained in the subset or
   b) one of its supertypes is contained in the subset.

Definition: Given a set of terms and their types. Then the union of types is the set of common supertypes includes for each pair of types
   a) the result of their successful unification whenever possible
   b) the common supertypes of their direct supertypes otherwise.

The set of minimal supertypes of types is the minimal subset of the union of types such that each common supertype is either
   a) contained in the subset or
   b) one of its subtypes is contained in the subset.

In the following we assume, without loss of generality, that for each two types defined, there is at most one minimal subtype and minimal supertype. We call such a type system deterministic, because every term and atom respectively has at most one type.

3 Meta-Interpreter

In this section we define three basic meta-interpreters which will be of use later when typing terms, atoms and conjunctions. Because there is no distinction between program and data in logic programming, it is quite easy to write so-called meta-programs which manipulate other programs. Meta-predicates are predicates which are defined over goals. A meta-program which interprets programs is called a meta-interpreter.

The 'plain-vanilla' Meta-Interpreter is a beloved part of the Prolog folklore:

prove(true).
prove((A,B)) :- prove(A), prove(B).
prove(A) :- clause((A:-B)), prove(B).

where the predicate clause/1 unifies with a program clause. As usual in Prolog implementations, we assume that facts are represented as rules with the body 'true'. This special predefined predicate 'true' is handled in the first clause of the meta-interpreter, while the second clause recursively traverses conjunctions. The third clause performs an actual derivation step by choosing a clause with clause/1 and further proving its body. Note that unification still is implicitly handled by the underlying Prolog system, while resolution is made explicit.

This meta-interpreter behaves exactly like the underlying Prolog system. Unfortunately it also shares a unpleasant property of Prolog implementations: The danger of infinite computations due to endless recursion. A query like :-append([X1,L1],L2,L1) looks pretty harmless and we would expect it to fail, but instead it loops producing an infinite sequence of resolvents :-append([X1,L3],L2,L3), :-append([X1,L4],L2,L4),...). There is no general way to decide if a recursion will loop or not, because this would imply solving the Halting Problem. But we can make certain provisions. One idea is to limit the number of derivation steps. Another idea is to stop computation by not unfolding certain atoms.

A depth-bound Meta-Interpreter The first idea can be implemented by adding a counter to the plain vanilla meta-interpreter, which is decremented at every derivation step. If the counter is zero, no more unfolding is possible. This method is called iterative deepening and a compromise between efficient depth-first evaluation and avoidance of infinite computations. We will use natural numbers (0,s(0),s(s(0)),...) for counting to keep things simple and in pure logic.

A meta-interpreter which limits the maximum number of derivation steps can be defined as:

plimit(true,N).
plimit((A,B),N) :- plimit(A,N), plimit(B,N).
plimit(A,s(N)) :- clause((A:-B)), plimit(B,N).

In the first clause, which terminates the meta-interpreter successfully, we do not care about the value of the counter N. In the second clause, N is passed to the two conjuncts. There is no need to decrement N here, because the number of atoms in a conjunction is finite, so there is no danger of some infinite computation. But in the third clause, which actually does one derivation step by unfolding an atom A, we decrement the counter by one. Once the counter is zero, it cannot unify with s(N) anymore, so the third clause cannot be selected anymore and the meta-interpreter will terminate.

\[\text{Definition} \quad \text{Given a set of types for the same typed term. We define the intersection of types to be the set of common subtypes which includes for each pair of types}\]
\[\text{a) the result of their successful unification whenever possible}\]
\[\text{b) the intersection of their direct subtypes otherwise.}\]
\[\text{The set of minimal subtypes of types is the minimal subset of the intersection of types such that each type in the intersection of types, i.e. each common subtype, is either}\]
\[\text{a) contained in the subset or}\]
\[\text{b) one of its supertypes is contained in the subset.}\]
\[\text{Definition} \quad \text{Given a set of terms and their types. Then the union of types is the set of common supertypes includes for each pair of types}\]
\[\text{a) the result of their successful unification whenever possible}\]
\[\text{b) the common supertypes of their direct supertypes otherwise.}\]
\[\text{The set of minimal supertypes of types is the minimal subset of the union of types such that each common supertype is either}\]
\[\text{a) contained in the subset or}\]
\[\text{b) one of its subtypes is contained in the subset.}\]

\[\text{In the following we assume, without loss of generality, that for each two types defined, there is at most one minimal subtype and minimal supertype. We call such a type system deterministic, because every term and atom respectively has at most one type.}\]

3 As [Mycroft/O'Keefe] point out, unification with occur-check is a must.
4 A term (atom) has no type if a function or predicate symbol occurring in the term or literal is not defined by a type.
A Partial Evaluator

Now consider the second idea. Suppose that from a query \((-g)\) we have derived a non-empty resolvent \((-g_1, \ldots, g_n)\). Then the clause \((g; g_1, \ldots, g_n)\) is a conditional answer to the initial query [Cheng et al.], [van Emden]. Such an incomplete computation of an answer is called partial evaluation [Venken], [Takeuchi/Furukawa]. The set of all conditional answers for a given query and program is called a residue of the initial query and the program. The meta-interpreter is accordingly extended in such a way that in each derivation step, either the selected goal is unfolded or returned as part of the residue depending. A partial evaluator can be derived from the plain meta-interpreter. An additional clause to deal with residual goals and an additional argument returning the residual goals is added:

\[
\begin{align*}
\text{peval}(true, true). \\
\text{peval}(A, B, C, D) & :- \text{peval}(A, C), \text{peval}(B, D). \\
\text{peval}(A, A) & :- \text{residual}(A). \\
\end{align*}
\]

where the predicates \text{residual/1} and \text{not\_residual/1} respectively determine if a selected goal is to be returned or unfolded.

Type Declarations for Meta-Predicates

In the following we illustrate the suitability of our type language to type meta-predicates. This issue is not addressed in most other works on types and Prolog. We consider the control-predicates for conjunction, disjunction etc. as meta-predicates and declare:

\[
\begin{align*}
\text{true} : \text{pred}. \\
(X, Y) : \text{pred} & :- X: \text{pred}, Y: \text{pred}. \\
(X; Y) : \text{pred} & :- X: \text{pred}, Y: \text{pred}. \\
\end{align*}
\]

The meta-call predicate \text{call/1} of Prolog can be easily typed as:

\[
\text{call}(X) : \text{pred} & :- X: \text{pred}.
\]

We can concisely type built-in meta-predicates like \text{bagof/3}

\[
\text{bagof}(X, Y, Z) : \text{pred} & :- X: A, Y: \text{pred}, Z: \text{list}(A).
\]

and our predicates for meta-interpreters:

\[
\begin{align*}
\text{prove}(X) & : \text{pred} & :- X: \text{pred}. \\
\text{plimit}(X, N) & : \text{pred} & :- X: \text{pred}, N: \text{nat}. \\
\text{peval}(X, Y) & : \text{pred} & :- X: \text{pred}, Y: \text{pred}. \\
\end{align*}
\]

This examples should suffice to show the ease of typing meta-predicates in our versatile type system.

4 Answering Type Queries with a Type Meta-Interpreter

In the following we give a specification-like implementation for our typing algorithm. Our main concern is not efficiency gained by tricky coding, but a readable and essentially pure Prolog program.

Answering Type Queries... Consider a type query \((-t:p)\). The query will either succeed possibly instantiating \(p\) and \(t\) or fail. The successful answer means that \(t\) has type \(p\) provided the type parameters and type arguments are instantiated as given by the answer. To illustrate this, assume an additional type 'const' with two facts \(a:\text{const}\) and \(b:\text{const}\) in addition to the type for lists. Consider the following examples:

\[
\begin{align*}
& :- \text{[]} : A. \\
& A = \text{list}(B). \\
& \% \text{[]} has type '\text{list}(B)' \\
& :- \text{[a|b]} : \text{list}(A). \\
& \text{no.} \\
& \% \text{[a|b]} is not a valid list \\
& :- \text{[a,b]} : \text{list}(\text{const}). \\
& \text{yes.} \\
& \% \text{[a,b]} is of type \text{list(const)} \\
& :- \text{X} : \text{list}(\text{const}). \\
& \text{X}[\text{[a;]}] \\
& \X = \text{[a, a];} \\
& \% \text{multiple answers generated by backtracking} \\
& \text{...} \\
& \X = \text{[a, ... a];} \\
& \% \text{...}
\end{align*}
\]

Clearly, such an infinite enumeration of answers is not very instructive. In the above example, we can get no idea that the type \text{list(const)} also includes lists with elements 'b'. Such lists are produced on backtracking after the infinitely many lists of a's, so we have no chance getting to a list with b's. Remembering the meta-interpreter for iterative deepening, we can overcome this trouble and submit the query as follows:

\[
\begin{align*}
& :- \text{plimit}(X : \text{list(const)}, s(s(s(0))))). \\
& \text{X} = \text{[;]} \\
& \text{X} = \text{[a;]} \\
& \text{X} = \text{[a, a;]} \\
& \text{X} = \text{[a, b;]} \\
& \text{X} = \text{[b;]} \\
& \text{X} = \text{[b, a;]} \\
& \text{X} = \text{[b, b].}
\end{align*}
\]
...with a Type Meta-Interpreter Using plimit/2 is nice, but note that each successful answer will always bind \( t \) to a ground term. We can prove this proposition by induction in showing that (1) type clauses with empty body (facts) have a constant (which is a ground term) in the head term by definition and (2) type clauses with non-empty body (rules) produce a ground head term as each variable in the head term is typed by a type goal in the body by definition and each type goal produces a ground term by the induction hypothesis.

We are rather interested in the type of the term as it is, with all its variables. Instead of a partial enumeration with plimit/2, we would like an answer stating, for example, that the term \([X]\) is of type list(A) if the variable \( X \) is of type A. This is a conditional answer. Hence we can try to use our partial evaluator peval/2. In order to avoid the instantiation of variables, we return the current type goal whenever the term to be typed is a type argument. To check if a term is an unbound variable, we use the built-in predicate vat/1, to check that a term is non-variable term, we use not_vat/1. We modify the recta-interpreter according to our needs into ptype/2:

\[
\begin{align*}
\text{ptype} & (\text{true}, \text{true}) . \\
\text{ptype} & ((A, B), (C, D)) :- \text{ptype}(A, C), \text{ptype}(B, D) . \\
\text{ptype} & (X: A, X: A) :- \text{var}(X) . \\
\text{ptype} & (X: A, C) :- \text{not_vat}(X), \text{clause}((X: A:-B)), \text{ptype}(B, C) .
\end{align*}
\]

Some examples show the behavior of the type meta-interpreter ptypet2:

\[
\begin{align*}
\text{C} & = X: \text{A} . \\
\text{A} & = \text{list(const)}, \text{C} = X: \text{const} .
\end{align*}
\]

As ptypet2 has a clause for conjunction, we can also answer conjunctions of type queries:

\[
\begin{align*}
\text{C} & = (X: \text{A}, Y: \text{const}) . \\
\text{A} & = \text{list(const)}, \text{C} = X: \text{list(const)} .
\end{align*}
\]

Since type declarations have the same syntax as type definitions, we can also type atoms:

\[
\begin{align*}
\text{C} & = (X: \text{A}, Y: \text{list(A)}). \\
\text{A} & = \text{list(const)}, \text{C} = X: \text{list(const)} .
\end{align*}
\]

Type Answers with Subtypes To illustrate the behavior of ptypet2 with subtypes, consider the following definitions:

\[
\begin{align*}
sue & : \text{woman} . \\
joe & : \text{man} . \\
\text{X} & : \text{human} :- \text{X} : \text{woman} . \\
\text{X} & : \text{human} :- \text{X} : \text{man} .
\end{align*}
\]

With subtypes, a term may have more than one type:

\[
\begin{align*}
\text{C} & = \text{true} ; \text{C} = \text{true} .
\end{align*}
\]

If we assume that in the Prolog program, the type clauses defining the subtypes come before the type clauses defining the supertypes, the first answer will always be the most specific type. A type query to find the common type of 'sue' and 'joe' will produce:

\[
\begin{align*}
\text{A} & = \text{human}, \text{C} = \text{true} . \\
\end{align*}
\]

The first answer is the minimal supertype. According to the definition of the type language, there is at most one minimal supertype. In general backtracking may produce additional common supertypes. Concluding, if we add subtypes, it is important to keep in mind that only the first answer of ptypet2 is the most general one.

Consistent Type Answers As the result of a type query, multiple types for the same variable may be returned as answer:

\[
\begin{align*}
\text{C} & = \text{true} . \\
\end{align*}
\]

Clearly, we would expect a type error in both cases. In the first example, because the intersection of the types 'woman' and 'man' is empty. In the second example, the variable \( L \) cannot be of type A and a list with elements of type A at the same time. In other words, multiple types for the same variable have to be consistent and they are whenever their intersection is non-empty. We are also interested in the result of the intersection, i.e. the minimal subtype of a variable with multiple types. According to the definition of the type language, there is at most one minimal subtype.

---

5 In the following we assume that each subtype clause of the form \( (H:-B_1; \ldots; B_n) \) is unfolded into \( n \) clauses \( (H:-B_i) \).

6 In the examples, redundant parentheses and 'true' atoms are removed from the actual answers to enhance readability.
We implement the consistency check and intersection of types by a generate-and-test approach:

```prolog
check_types(A,B) :-
  subtypes(A,B), % generate subtypes
  factor(B), % test them by factoring
  !. % cut avoids backtracking
```

The predicate `factor/1` is a test which factors type goals, i.e., intersects types for the same variable by unifying them. It cannot deal with subtypes. Instead the predicate `subtypes/2` generates smaller and smaller guesses for the result of the intersections by replacing types in the residue of `ptype/2` with its subtypes. The cut `!` is necessary to avoid further backtracking once a consistent set of subtypes is found. Otherwise additional solutions with smaller types may be returned.

The predicate `subtypes/2` takes a type goal, and indeterministically replaces some of its type atoms with a subtype:

```prolog
subtypes(true,true).  
subtypes((A,B),(C,D)):- subtypes(A,C), subtypes(B,D).  
subtypes(X:A,X:A).  
subtypes(X:A,X:C):- not_var(A), clause((Y:A:-B)), var(Y), subtypes(B,X:C).
```

Note that `subtypes/2` is very similar to `ptype/2` and is just another partial evaluator\(^7\). The check for a non-variable type, `not_var(A)`, avoids unnecessary binding of type parameters. A subtype clause is recognized by the fact that it has exactly one subgoal and that the type argument of the head is a variable. The recursive call to `subtype/2` ensures that all (not only direct) subtypes of a given type are produced on backtracking.

The predicate `factor/1` takes a type goal, looks at each possible pair of its type atoms and intersects them, if necessary. The predicate uses a divide-and-conquer strategy. First it factors the conjuncts of a conjunction independently and then factors types of common type arguments from the conjuncts by applying `factor/2`. The predicate `factor/2` pairwise unifies the type goals of its arguments, taking one type goal from the first and the other from the second argument:

```prolog
factor(true).  
factor(X:A).  
factor((A,B)):- factor(A), factor(B), factor(A,B).  
factor(true,B).  
factor(X:A,Y:B) :- not_same_var(X,Y).  
factor(X:A,Y:B) :- same_var(X,Y), unify(A,B).  
factor((A,B),C) :- factor(X:A,B), factor(X:A,C).  
where same_var(X,Y) succeeds if X and Y are identical unbound variables and `unify/2` unifies its arguments. For example, assume the following type definitions:

```prolog
mother(X,Y) :pred:- X:woman, Y:human.  
father(X,Y) :pred:- X:man, Y:human.  
```

and the type queries

```prolog
:- ptype((father(X,Y),father(Y,Z)) :pred, C), check_types(C,D).  
:- ptype((X:woman,father(X,Y) :pred),C), check_types(C,D).  
nom.
```

## 5 Type-Checking

In this context the aim of (static) type checking is to establish that a program is indeed well-typed. Type-checking a program means to determine if each clause in the program is well-typed. Well-typing a clause consists of two steps. First we type the clause by typing its body and deriving the type of the arguments of the head. Then this derived type is compared to the given, declared type of the corresponding predicate.

### Typing Clauses

To type a clause (H:-B) we cannot use simply a conjunction `:- ptype(H:pred,TH)`, `ptype(B:pred,TB)`), because that would not produce a derived type for the arguments of the head. The reason is that the call `:-ptype(H:pred,TH)` would already type the clause head H according to its type declaration. The subsequent comparison with the declared type would be useless and not reveal any type error. For the derived type, we are interested in the type of the arguments of the head without reference to the declared type. The type of an argument is determined by the function symbols and variables occurring in that argument. Therefore we assume that each argument is initially typed by a type parameter (instead of being typed by the declared type as given in the type declaration). The predicate `make_type_param/2` replaces each type in the type declaration HD by a type parameter and that is how we can compute the derived type TH.

---

\(^7\) The unification of types needed for intersection of types is referred to as factoring in [van Emden] in the context of type inference for lambda expressions.

\(^8\) Indeed, in an earlier version of this paper, instead of a new predicate `subtypes/2` a single clause was added to `ptype/2`, but this was both inefficient and tricky coding.
type_clause((H:-B), (TH, TB), HD, HI) :-
  clause((H:pred:-HD)),
  make_type_param(HD, H1),
  ptype(H1, TH),
  ptype(B:pred, TB),
  !.
make_type_param(true, true).
make_type_param(X:A, X:B).
make_type_param((A, B), (C, D)) :-
  make_type_param(A, C),
  make_type_param(B, D).

The predicate type_clause/4 takes a clause and returns the types of its variables, the declared and derived type as its answer. The subgoal clause/1 lets us find the declared type HD of the clause. The subgoal make_type_param/2 generates a conjunction of type parameters H1 from the declared type by replacing each type argument with a new distinct type parameter. This conjunction and the body of the clause is then typed by ptype/2. Again we are only interested in the first solution, as solutions found on backtracking would contain non-minimal types. Hence a cut is present at the end of the clause.

Checking theDeclared Type against the Derived Type From the well-typing rules for the type system with parametric polymorphism alone [Mycrof/O'Keefe] we know that a clause is well-typed if the declared type is an instance of the derived type9. To find out how to handle subtypes, first consider the following example:

parent(X, Y) :- mother(X,Y).
parent(X, Y) :- father(X,Y).

The predicate parent/2 has either type man or woman for the first argument. The declared type of the list argument, human, covers the union of the derived types man and woman. This is equivalent to the condition that the declared type covers the derived type of each clause. To verify this requirement, the predicate check_decl/2 uses the same generic-and-test approach as check_type/2. The declared type covers the derived type if we can replace the types in the derived type by subtypes such that the resulting type is an instance of the derived type. We use subtypes/2 for this purpose again. Finally, the predicate instance(A, B) checks if A is an instance of B. Note that the presence of the cut, which prevents further backtracking, is important. Otherwise we could generate smaller and smaller guesses for HD1 to fulfill the instance/2 check. The effect would be that the type checker would only check if the derived type and the declared type intersect.

check_decl(HD, HI) :-
  subtypes(HD, HD1), % generate subtypes of declared type
  instance(HD1, HI), % compare declared to derived type
  !. % cut avoids backtracking

Well-Typing Clauses Now we can state that a clause10 is well-typed if:

well_typed((H:-B)) :-
  type_clause((H:-B), THB, HD, HI), % type the clause
  check_type(THB, THBI), % make types consistent
  check_decl(HD, HI). % check derived against declared type

The clause is typed by type_clause/4, the resulting residue THB made consistent with check_type/2 and the derived type H1 is checked against the declared type HD with check_decl/2. Some examples for type_clause/4 illustrate the behavior of this type-checker. First we well-type append/3:

```prolog
:- type_clause((append([], L, L) :-true), THB, HD, HI).
THB = ([L:B, L:C]),
HD = ([L:list(D), L:list(D), L:list(D)],
HI = ([L:list(A), L:B, L:C]).
```

The subgoal check_type/2 will unify B and C. Then check_decl/2 will succeed as HD is an instance of H1.

```prolog
:- type_clause((append([X|L1], L2, [X|L3]) :- append(L1, L2, L3)), HB, HD, H1).
THB = ([X:A, L1:list(A), L2:E, X:C, L3:list(C), L1:list(A), L2:list(A), L3:list(A)],
HD = ([X|L1]:list(D), L2:list(D), [X|L3]:list(D)],
H1 = ([X|L1]:list(A), L2:E, [X|L3]:list(C)).
```

The subgoal check_type/2 will unify A and C as well as E and list(A). Then check_decl/2 will succeed as HD is an instance of H1.

---

9 This is equivalent to the original rule where the head atom is first also typed by the declaration and the result must be a variant of the declaration. We do not type the head atom with the declared type to be able to deal with subtypes.

10 Facts are considered to be clauses with the body 'true', queries are considered to be clauses with the head 'true'.

An example for a predicate with subtypes is presented next:

```prolog
:- type_clause((parent(X,Y) :- mother(X,Y)), THB, HD, HI).
THB = (X:A, Y:B, X:woman, Y:human),
HD = (X:human, Y:human),
HI = (X:A, Y:B).
```

First A is bound to woman and B to human. Then check_decl/2 succeeds as there is a subtype replacement for HD, (X:woman,Y:human), such that HD is an instance of H1. Analogously the second clause of parent/2 is well-typed.

### 6 Related Work

Our type system is based on the [Mycroft/O'Keefe] approach to type checking. The main difference is that we add inclusion and additive polymorphism, i.e. subtypes, to our type-checking system. Instead of using an extra-logical functional type language, we use a type language based on Prolog itself. Thus type definitions have a clear semantics as binary predicates. They can also be used freely in the program itself.

Well-typing is defined as a fixpoint-operator for a type system with monomorphic types in [Yardeni/Shapiro]. Their approach is based on the sets of ground terms which can occur as arguments of predicates. They use a BNF-style notation to define their types. The absence of polymorphism enforces rather general types, as in the example of append/3:

```prolog
Any_list => [ ] ; [Any|Any_list].
type append(Any_list,Any_list,Any_list).

Consider the following example taken from their paper to define typings for a merge-sort program:

```
Natural => 0 ; s(Natural).
Nat_list => [ ] ; [Natural|Nat_list].
Special_list => [ ] ; [Natural|Special_list].
List_of_Nat_list => [ ] ; [Nat_list|List_of_Nat_list].
type mergesort(Nat_list,Nat_list).
type convert(Nat_list,Special_list).
type mergesort(Xs,Ys) :- convert(Xs,Xs1), msort(Xs1,[Ys]).
```

With monomorphic types one needs more type definitions than in a polymorphic type checking system like ours:

```prolog
[X]:list(A). [X|L]:list(A):- X:list(A), L:list(A).
mergesort(X,Y):pred:- X:list(nat), Y:list(nat).
convert(X,Y):pred:- X:list(nat), Y:list(list(nat)).
msort(X,Y):pred:- X:list(list(nat)), Y:list(list(nat)).
```

Note that the type of the second argument of convert/2 can be given more precise in [Yardeni/Shapiro]. Indeed, any more general type declaration, i.e. List_of_Nat_list, would result in an ill-typed program, as in that case the set of ground terms inferred relative to the type declaration is always smaller than the type declaration itself. This stems from the fact that the semantics of the type system is based on the notion of sets of ground terms, while our type checking system checks equivalency of types on the type-level. In other words, the former is a value-based and the latter is a name-based type system [Wegner/Cardelli].

[Xu/Warren] integrate type checking and type inference into a uniform type system and give a clear semantics for type declarations. This is not possible in our type system, as it is not defined on the ground terms of the Herbrand universe, but rather treats free variables different from terms. We believe that this allows more useful typings. Their system is concentrates on type inference, while our concern is solely type checking.

[Hanus] proposes a more general type language and definition of well-typing than that of [Mycroft/O'Keefe], which allows a more natural treatment of high-order predicates, but may require runtime type checking. We do not deal with typings for higher-order predicates in this paper, this is the topic of current research.

[Dietrich/Hagl] add inclusion polymorphism to the approach of [Mycroft/O'Keefe]. In contrast to our work, their elaborate well-typing algorithm deals explicitly with the problem of runtime type checking, but requires mode information to be present.

In [Cardelli-2] a powerful type language for applicative languages is proposed. There is also a notion of sets of types, based on the concept of so-called power types. The paper gives a semantic account of sets of types as types of types, while we consider type sets as a purely syntactical means.

[Jacobs] presents a type language for algebraic database programming languages. Our type language is less powerful but has a simple and decidable type checking procedure, while [Jacobs] gives type checking rules but does not deal with their efficient implementation.
An approach in the same spirit as our meta-interpreter implementation technique was taken in [van Emden] to derive conditional answers for polymorphic type inference of lambda expressions. He also uses factoring to intersect types.

[Smolka] modified order-sorted logic, which models inclusion polymorphism naturally, to parametrically order-sorted logic (POS) in order to handle parametric polymorphism. Though this elaborate approach is rather different to ours and relies on notions of generalized constraint languages, partial algebras and rewriting systems, the actual implementation, as far as it is described, employs an algorithm similar to ours. But contrary to our work, [Smolka] notes that he can 'only give an incomplete algorithm' whose success depends on the order of subgoals.

7 Conclusions
We have implemented a prototype of a state-of-the-art polymorphic type checking system with subtypes for Prolog in Prolog with Prolog. Therefore the type system is naturally embedded into typeless Prolog. In other words, Prolog is the typed language is the type language is the type system implementation language. We gave a meta-interpreter based specification of a type checking system. In this way, the problem of implementing was reduced mainly to modifying a set of standard meta-interpreters. By considering typings for predicates as special case of type definitions, it is as easy in the type system to type meta-predicates as it is to use them in Prolog. Therefore meta-programs like meta-interpreter can be typed concisely in our type system, an issue not addressed fully in previous work.

Further work will tackle the open problems associated with typing higher-order constructs such as apply/1. Other goals are to annotate the type checker with error-messages in a methodic way and to derive an efficient Prolog implementation from its specification in Prolog.

Acknowledgements
We are grateful to You-Chin Fuh, Michael Kifer, Patreek Mishra and David Warren (in alphabetical order) for enlightening discussions and anonymous referees for comments.

References
[Bruynooghe] Bruynooghe M, Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs, 1st Int Conference on Logic Programming, Marseille France, 1982
[Cardelli-1] Cardelli L, Basic Polymorphic Type Checking, Draft, AT&T Bell Laboratories, Murray Hill New Jersey, ≥ 1984
[Dietrich/Hagl] Dietrich R and Hagl F, A Polymorphic Type System with Subtypes for Prolog, 2nd European Symposium on Programming, Nancy France, 1988
[Friihwirth-2] Frühwirth Thom W, A Polymorphic Type Checking System for Prolog in HiLog, 6th Israel Conference on Artificial Intelligence and Computer Vision, Ramat Gan Israel, December 1989
[Jacobs] Jacobs D, A Type System for Algebraic Database Programming Languages, 2nd International Workshop on Database Programming Languages, Glieneden Beach Oregon, June 1989


[van Emden] van Emden M H, Conditional Answers for Polymorphic Type Inference, in 5th Int Logic programming Conference and 5th Int Symposium on Logic programming, Kowalski R A and Bowen K A (eds), MIT Press, Cambridge Massachusetts, 1988


