A Polymorphic Type System for Prolog*

Alan Mycroft**
Department of Computer Science, Edinburgh University, Kings Buildings, Mayfield Road, Edinburgh EH9 3JZ, U.K.

Richard A. O'Keefe
Department of Artificial Intelligence, Edinburgh University, Hope Park Square, Meadow Lane, Edinburgh EH8 9NW, U.K.

Recommended by Daniel Bobrow

ABSTRACT

We describe a polymorphic type scheme for Prolog which makes static type checking possible. Polymorphism gives a good degree of flexibility to the type system, and makes it intrude very little on a user's programming style. The only additions to the language are type declarations, which an interpreter can ignore if it so desires, with the guarantee that a well-typed program will behave identically with or without type checking. Our implementation is discussed and we observe that the type resolution problem for a Prolog program is another Prolog (meta)program.

1. Introduction

Prolog currently lacks any form of type checking, being designed as a language with a single type (the term). While this is useful for learning it initially and for fast construction of sketch programs, it has several deficiencies for its use as a serious tool for building large systems. These centre around the facts that type errors can only be detected at run-time and that modules cannot have secure representations.

We have observed that a theorem prover which reasons about Prolog programs can be more powerful if it has type information available. One

*This work was supported by the British Science and Engineering Research Council.

**Present address: Institutionen för Informationsbehandling, Chalmers Tekniska Högskola, S-412 96 Göteborg, Sweden.

Artificial Intelligence **23** (1984) 295–307
0004-3702/84/$3.00

indication as to why this is so can be seen from the fact that the traditional definition of *append* has append(nil, 3, 3) deducible from its definition.

Prolog programs can contain many kinds of errors. Tools exist to detect several error classes. Two of the commoner error classes for which no tool previously existed are transposed arguments and omitted cases. Type checking often catches the first error and (providing we are willing to adopt a certain programming style) can give a method for determining whether all the cases in a Prolog predicate have been considered. For example, a predicate defined by

type neglist(list(int), list(int))
neglist(cons(A, L), cons(B, M)) ← negate(A, B), neglist(L, M)

will never succeed, since we have probably omitted the clause

neglist(nil, nil) ←

A type system would enable us to detect this by checking for exhaustive specification of argument patterns for a given data type. Of course, if we really did want a certain case to fail, then adding a clause such as

neglist(nil, nil) ← fail

would be an explicit way of requesting such an event without leaving first-order logic (and would facilitate later reading of the program).

Moreover, our type system can be used as the basis of an encapsulation providing an abstract data type facility. The ability to hide the internal details of a given object greatly aids the reliability of a large system built from a library of modules.

Finally, we note that static type checking cannot of itself provide a great increase in speed of Prolog programs, due to the fact that term unification must still be performed, as in the dynamic case. However, a new compiler for typed Prolog could improve the speed of compiled clauses of a given predicate by using a mapping of data constructors onto small adjacent integers to enable faster selection of the clause(s) to be invoked. By far the greatest gain is that of programmer time provided by early detection of errors.

As far as we know this work is the first application of a polymorphic type scheme to Prolog, but related work includes Milner's work [4] on typing a simple applicative language which is used in the ML [3] type checker and the HOPE language which uses a version of Milner's algorithm extended to permit overloading. However, this work differs from these in several respects. Firstly, the formulation of Prolog as clauses means that the problems of generic and nongeneric variables are much reduced. All predicate and functor definitions naturally receive generic polymorphic types which can be used at different type

instances within the program whilst all variables receive nongeneric types. Moreover, our formulation for Prolog removes a restriction in Milner's scheme in which all mutually recursive definitions can only be used nongenerically within their bodies. Thus in ML the (rather contrived) program

let rec $I\,x = x$
and $f\,x = I(x+1)$
and $g\,x =$ **if** $I(x)$ **then** 1 **else** 2

would be ill-typed. Since all Prolog clauses are defined mutually recursively, this restriction would have the effect of making the polymorphism useless.

Our approach is to consider type specifications as *restrictions* on arguments to predicates (and functors). Some built-in predicates already have such restrictions, for example "Z is $X + Y$" or plus(X, Y, Z) are not semantically meaningful when X, Y or Z is instantiated to a non-integer and in typical implementations give a run-time error rather than simple failure to indicate a presumed programming error. Our type scheme enables the user to place restrictions on defined predicates in an analogous manner. Moreover, we do not test for violation of such restrictions at run time, but by statically forbidding all programs which may lead to a type error. We make a slogan out of this and say that "well-typed programs do not go wrong". The notion of 'wrong' here is independent of success, failure or looping. For example "6 is 4 + 3" is well-typed and fails whereas eqint(foo, foo) (defined by eqint(X, X) and type restricted to integers) is ill-typed for us but would succeed if evaluated. Due to the type-checker being a separate program, type checking does not change the semantics of Prolog, but merely discourages the execution of ill-typed programs.

In contrast, Mishra [5] considers types as sets of terms generated by a regular tree grammar with the semantic basis that predicate p has type T if $p(t)$ fails for all t outside T. Such types include our *monomorphic* types (based on many-sorted algebras) as a special case, but it is unclear whether this idea can be naturally extended to *polymorphic* types (for example the polytype $\alpha \times \alpha$ can only be badly approximated by using the same regular trees as for $\alpha \times \beta$). One advantage is that the implicit data type declarations of an untyped Prolog program can often be determined.

2. Mathematics

We assume the notion of substitution, a map from variables (and terms by extension) to terms, ranged over by θ and ϕ. An invertible substitution is called a renaming. If a term, u, is obtained from another, v, by substitution then we say that u is an instance of v, and write $u \preceq v$. We write $u \cong v$ if $u \preceq v$ and $v \preceq u$. This means that u and v only differ in the names of their variables and that the substitutions involved are renamings. Also assumed is the notion of most general unifier (MGU) of two terms.

For any class of objects S, the notation S^* will be used to indicate the class of objects consisting of finite sequences of elements of S.

3. Prolog

The simple variant of Prolog we consider will be defined by the following syntax (we assume the existence of disjoint sets of symbols called Var, Pred and Functor, representing variables, predicates and functors symbols respectively):

Term ::= Var|Functor(Term*),
Atom ::= Pred(Term*).
Clause ::= Atom ← Atom*,
Sentence ::= Clause*,
Program ::= Sentence; Atom,
Resolvent ::= Atom*.

By definition of clause form each, implicitly universally quantified, variable appears in at most one clause. To make the formal description of typing simpler, we assume that the textual names of variables also follow this rule. A program then is given by a finite list of (Horn) clause declarations, followed by an Atom (short for atomic formula), called the query, to evaluate in their context. It specifies an initial resolvent by taking the query and treating it as a one-element list.

The evaluation mechanism for Prolog is very simple, and based on the notion of SLD-resolution as the computation step.

SLD-resolution is the one-step evaluation which transforms a Resolvent. Given a resolvent

$$R = A_1, \ldots, A_n$$

we select an Atom, the *selected atom*, say A_k, and perform resolution with it and a matching clause. So, choose a clause of the program, the *selected clause*, say Q, given by

$$C \leftarrow B_1, \ldots, B_m$$

and suppose that R has no variables in common with it (otherwise we must rename its (Q's) free variables since they are implicitly universally quantified for the clause).

Now let θ be MGU(A_k, C) if this exists. If it does, then we can rewrite R into R' given by

$$\theta(A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n).$$

The most common form of Prolog interpreter uses $k = 1$ when this expression simplifies somewhat.

An answer is produced when the resolvent is rewritten into a sequence of

zero atoms. The associated answer to such a rewriting sequence is the composition of most general unifiers encountered during the rewriting process, or rather its restriction to the variables in the query.

Observe that the above specification only told us how we could produce an answer (if one exists) from a Prolog program. For computation the choices above (the selected atom and clause) must be incorporated into a deterministic tree searching algorithm, which we take time to explain below for the reader's benefit. However, we would like to stress now that the results on type checking given in Section 5 work for *any* order to evaluation (choices of atoms and clauses) of Prolog programs (depth-first/breadth-first/coroutining/parallel).

3.1. Digression: SLD-trees

The idea of SLD-resolution above, leads to the idea of an SLD-tree: whenever we are forced to select a *clause* then, instead of irreversibly choosing a given matching clause, we construct a tree of resolvents (an SLD-tree) where a resolvent has a son resolvent for each clause which matches with the selected atom. A sensible computation (the standard implementation of Prolog) is then to search this tree in depth-first left–right manner.

Some branches die out, in that no clause matches the selected atom, whereas others have more than one subtree contribute to the answer. This is often referred to as the nondeterminacy of Prolog.

Finally, we remark that there is never any need to seek alternatives to the selected *atom*—in fact doing so would merely lead to duplication of answers exhibited elsewhere in the SLD-tree. (For more details on this aspect see [1].)

4. Types

The scheme of types (Type) we allow are given by the following grammar and are essentially the same as those which occur in ML [3]. We assume disjoint sets of type constructors (Tcons, ranged over by roman words) and type variables (Tvar, ranged over by greek letters like α, β, γ). These are also assumed to be disjoint from Var, Pred and Functor.

$$\text{Type} ::= \text{Tvar} \mid \text{Tcons}(\text{Type}^*)$$

Type will be ranged over by $\rho, \sigma, \tau, \ldots$ A type is called a monotype if it has no type variables. Otherwise it is a polytype.

For example, we suppose that Tcons includes the nullary constructor *int* and the unary *list*. Example types are then

$$\text{list}(\alpha), \quad \text{int}, \quad \text{list}(\text{list}(\text{int})), \quad \text{etc.}$$

Note that the third type is an instance of the first.

4.1. Digression: the Unary Predicate Calculus

The type systems used *in* many AI programs are variants or restrictions of the Unary Predicate Calculus. However, UPC is not adequate as the single type system *for* an AI programming language. Rules such as

$$(\forall N, L)\ \text{integer}(N) \wedge \text{int\_list}(L) \Leftrightarrow \text{int\_list}(\text{cons}(N, L)),$$
$$(\forall L)\ \text{int\_list}(L) \Leftrightarrow L = \text{nil} \vee (\text{integer}(\text{car}(L)) \wedge \text{int\_list}(\text{cdr}(L)))$$

cannot be expressed in it.

5. Well-typing of Prolog

This section contains the central definition of a Prolog program being well-typed, together with precursor and auxiliary definitions. Many of the ideas appear in [4] where a polymorphic applicative language is typed, but our formulation for Prolog poses new problems and simplifies old ones as we discussed in the introduction.

Let Q be the clause $C \leftarrow B_1, \ldots, B_m$ and P be a finite subset of Var $\cup$ Pred $\cup$ Functor containing all the symbols of Q. We define a typing $\bar{P}$ of P to be an association of an extended type of each symbol occurring in Q. The types are members of a given algebra as defined in Section 4. Predicates and functors are associated with extended types as given below. Types and extended types will be written as a superscript on the object they are associated with. σ_i and τ will represent (non-extended) types. For each variable X occurring in Q, $\bar{P}$ will contain an element of the form X^τ, for each predicate a of arity k in Q, $\bar{P}$ will contain an element of the form $a^{\sigma_1, \ldots, \sigma_k}$. For each functor f of arity k in Q, $\bar{P}$ will contain an element of the form $f^{(\sigma_1, \ldots, \sigma_k) \to \tau}$.

Similarly, the clause Q will be written as a typed clause $\bar{Q}$ by the writing of a type on each term (this includes variables).

As an example of a clause and its typing consider the clause Q, given by

$$\text{app}(\text{cons}(A, L), M, \text{cons}(A, N) \leftarrow \text{app}(L, M, N).$$

The set $P = \{A, L, M, N, \text{app}, \text{cons}\}$ gives its set of symbols, and a typing (which will turn out to be a well-typing considered later) can be given by $\bar{P}$:

$$\{A^\alpha, L^\tau, M^\tau, N^\tau, \text{app}^{\tau,\tau,\tau}, \text{cons}^{(\alpha,\tau)\to\tau}\}$$

where τ is used as a shorthand for list(α) and the associated typed clause $\bar{Q}$ given by:

$$\text{app}(\text{cons}(A^\alpha, L^\tau)^\tau, M^\tau, \text{cons}(A^\alpha, N^\tau)^\tau) \leftarrow \text{app}(L^\tau, M^\tau, N^\tau).$$

$\bar{P}$ will be called the typed premise of $\bar{Q}$ due to the relation to theorem proving.

Fortunately, it will turn out that most of the mess of types written above are interdependent and the above expression can be well-typed much more succinctly—see later.

We will now define $\bar{Q}$ to be a well-typing of Q under $\bar{P}$, written $\bar{P} \vdash \bar{Q}$ if the following conditions hold:

(1) $\bar{P} \vdash (A \leftarrow B_1, \ldots, B_m)$ if
$A = a(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ and $a^{\rho} \in \bar{P}$
with $(\tau_1, \ldots, \tau_k) \equiv \rho$
and $\bar{P} \vdash t_i^{\tau_i}$ $(1 \leqslant i \leqslant k)$
and $\bar{P} \vdash B_i$ $(1 \leqslant i \leqslant m)$.

(2) $\bar{P} \vdash A$ if A is an Atom and
$A = a(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ and $a^{\rho} \in \bar{P}$
with $(\tau_1, \ldots, \tau_k) \leqslant \rho$
and $\bar{P} \vdash t_i^{\tau_i}$ $(1 \leqslant i \leqslant k)$.

(3) $\bar{P} \vdash u^{\sigma}$ if u is a Term and
$u = f(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ and $f^{\rho} \in \bar{P}$
with $((\tau_1, \ldots, \tau_k) \rightarrow \sigma) \leqslant \rho$
and $\bar{P} \vdash t_i^{\tau_i}$ $(1 \leqslant i \leqslant k)$.

(4) $\bar{P} \vdash X^{\sigma}$ if $X^{\sigma} \in \bar{P}$.

Now, we will define a program to be well-typed under a typed premise $\bar{P}$ if each of its clauses is well-typed under $\bar{P}$ and if its query atom is. Similarly a resolvent is well-typed if each of its atoms is.

Well-typing as a mathematical concept is of little use, unless we relate it to computation. This we will now do, under the motto 'Well-typed programs do not go wrong'.

6. Well-typed Programs Do Not Go Wrong

What we desire to show, is the semantic soundness condition that if a program can be well-typed, then one step of SLD-resolution will take a well-typed resolvent into a new well-typed resolvent. Thus any SLD-evaluation of a well-typed program will remain well-typed. It is trivially the case that the initial resolvent is well-typed if the program is. Moreover, this means that the variables in the query can only be instantiated to terms specified by their types given by the well-typing. As discussed in the introduction "do not go wrong" means only that resolvents satisfy the type restrictions, and nothing about whether the evaluation succeeds, fails or loops.

The first condition is simply proved: Let R be the resolvent $A_1, \ldots, A_n$ and let Q be a clause $C \leftarrow B_1, \ldots, B_m$ which has no variables in common with R (the case where Q and R have variables in common will be discussed later). Without loss of generality (symmetry) let A_1 be the selected atom and suppose $\theta = \mathrm{MGU}(A_1, C)$ exists. The resolvent produced by one-step evaluation is R' given by

$$\theta(B_1, \ldots, B_m, A_2, \ldots, A_n).$$

We will now show how to well-type this from the well-typing of R.

Let us suppose that there is a typing $\bar{P}$ of the symbols of Q and R and associated well-typings $\bar{R}$ and $\bar{Q}$ such that $\bar{P} \vdash \bar{R}$ and $\bar{P} \vdash \bar{Q}$ (note this provides well-typings $\bar{A}_i$, $\bar{C}$, $\bar{B}_i$). Moreover, let us suppose that $\bar{R}$ and $\bar{Q}$ have no type variables in common (again, we will discuss this later, but note that the typing rules never rely on the 'absolute' names of the type variables).

Let the type of the predicate symbol of C in $\bar{P}$ be $c^{\rho_1,\ldots,\rho_k}$. Now the well-typing determines that $\bar{C}$ can be written $c(s_1^{\sigma_1}, \ldots, s_k^{\sigma_k})$ and $\bar{A}_1$ as $c(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$ where

$$(\sigma_1, \ldots, \sigma_k) \cong (\rho_1, \ldots, \rho_k),$$
$$(\tau_1, \ldots, \tau_k) \leqslant (\rho_1, \ldots, \rho_k).$$

This means that there is a substitution ϕ on type variables (actually $\phi \cong \mathrm{MGU}((\sigma_1, \ldots, \sigma_k), (\tau_1, \ldots, \tau_k))$) such that $(\tau_1, \ldots, \tau_k) = \phi((\sigma_1, \ldots, \sigma_k))$.

The claim is that

$$\bar{P} \vdash \theta(\phi(\bar{B}_1), \ldots, \phi(\bar{B}_m), \bar{A}_2, \ldots, \bar{A}_n)$$

gives a well-typing of R', where applying ϕ (a type substitution) to a typed atom means that it is to be applied to the type variables in types associated with terms occurring within that atom.

We now address the problem of there being variables, or type variables, in common between R and Q. These are really the same problem (the perennial one of renaming in Prolog). A simple solution is the following: Whenever we come to perform resolution between a clause Q and a resolvent R we rename Q such that all its variables (using a renaming ψ) and all its type variables (using a renaming η) are distinct from the variables (and type variables) in R and the other clauses. This can always be done since R can only contain a finite number of different variables. Moreover, this does not change the meaning of Q. This strictly breaks the type scheme, since the new variables appearing in Q do not appear in $\bar{P}$. However, a simple addition to $\bar{P}$ of $\psi(X)^{\eta(\tau)}$ for each variable X in the original Q which appeared as X^τ in $\bar{P}$ serves to correct this and preserve the typing. We are now back in the case where Q and R have no variables or type variables in common.

We now return to the problem of showing that a well-typed program can only instantiate the variables of its query to values having types as dictated by the typed premise. To see that this is the case, it is merely necessary to observe that each resolution step (as above) is performed between an Atom A and a (type) instance of a clause $C \leftarrow B_1, \ldots, B_m$, such that the types of A and this instance of C are identical except for the names of type variables. Thus variables in A can only be instantiated to Terms (possibly other variables) having identical types. The whole result is proved by induction on the length of computation leading to a refutation.

7. Specification of the Type Information to Prolog

We suggest that the type specification be performed by annotations to the Prolog system. The well-typing required three sets of information to be supplied:

- the types of the predicates;
- the types of the functors;
- the types of the variables.

We suggest that declarations be supplied which give the type of the first two but the type of variables can easily be determined from them. This can be seen by observing that a well-typed Atom or Term labels the type of each argument Term, and so each variable is labelled with a type. The most general unifier of all the types associated with a single variable (if it exists) gives a type for that variable. (This is also convenient since the scope of variables in Prolog is a single clause, whereas the other objects have a global scope.)

It is convenient to specify the names of types along with the functors which create them from other types. This has been demonstrated by HOPE [2] and we do not expect to better this idea.

So one of the declarations, or metacommands is one of the form

Declaration ::= **type** Tcons(Tvar*) $\Rightarrow$ Functor(Type*)* .

Examples would be (the second somewhat improper)

type list(α) $\Rightarrow$ nil, cons(α, list(α)) .
type int $\Rightarrow$ 0, 1, −1, 2, −2, 3, −3, . . .

The second declaration specifies the type of predicates. Suggested syntax is

Declaration ::= **pred** Pred(Type*) .

and an example for the 'equal' function defined by

equal(X, X) $\leftarrow$

would be

pred equal(α, α) .

We note here, that, given the types of the functors, it would seem possible to determine the types of the predicates involved without any great amount of work (as in ML [4]). However, this seems to depend on an analysis of the whole program at once, rather than any form of interaction.[1] We would also claim that the documentation provided by the written form of the types facilitates human understanding of programs in much the same way that

[1] Moreover there is a small technical problem concerning recursive definitions which makes checking of type specifications of such definitions much easier than their derivation.

explicit specification of mode information (input/output use of parameters) for predicates does.

7.1. Abstract data types

We observe that the above declarations furnish a form of abstract data typing. Providing a 'module' construct and exporting from it a given type name, and predicates which operate on that type, but *not* the constructor functors for that type, enables us to use a type, but not to determine anything about its representation. HOPE has such a construct, and we think it would greatly benefit Prolog.

8. Overloading

The above discussion has centred on a formalism for well-typing Prolog. However, it does not allow for one feature which we have found to be useful, and which is very easy to build into the type system. This feature is overloading and appears in a similar form in HOPE [2].

The observation, is, that quite often, we may wish a given function, predicate or functor name to stand for more than one distinct operation. This is common in mathematics and computer science, where an operator (e.g. '+') may be used to denote a different function at different types. In Prolog this can be useful too. For example, we may wish to have types specified by

$$\text{list}(\alpha) \Rightarrow \text{nil}, \text{cons}(\alpha, \text{list}(\alpha)),$$
$$\text{tree}(\alpha) \Rightarrow \text{nil}, \text{leaf}(\alpha), \text{cons}(\text{tree}(\alpha), \text{tree}(\alpha)),$$

where the constructors nil and cons(_ , _) have different meanings according to whether they act on lists or trees. (Of course we *could* give them different names, but this is not always helpful to the programmer.)

Similarly, we may want certain predicate symbols to refer to different predicates according to the type of their arguments. A typical example would be some sort of 'size' predicate.

We formalize this by permitting the typed premises used above to contain more than one type associated with any given functor or predicate symbol.

9. Implementation

We have built such a system in Prolog which implements the overloaded type checker by backtracking. Note that this is not particularly difficult since our well-typing rules given in Section 5 are essentially Horn clauses. There are merely two points to observe. Firstly, the 'occur-check' of unification (which is often omitted by Prolog implementations) is essential for this type checking scheme. Secondly, the use of $\leqslant$ can be simulated by instantiation of a copy of the functor or predicate type and the use of $\cong$ by a common metalinguistic

predicate (*numbervars*) which instantiates variables in a term to ground terms to avoid their further instantiation. Copies of the code can be found in [6].

That the well-typing rules (which define when a given program has a given type) can be used to determine the type of a given program is a simple consequence of the Horn clause input/output duality. Moreover, when the well-typing rules are used in this fashion on a given program, T say, then the standard SLD-resolution will produce a *terminating* evaluation giving the most general types associated with T. The basic idea is that if the well-typing problem has no solution, then the program is ill-typed. If it has exactly one, then the program is well-typed, and if it has more than one then some overloaded operator is ambiguous.

10. Higher-order Objects

This section is much more tentative and more in the manner of suggestion than the rest of the paper and we would be grateful for any comments on its inclusion or its contents. It is included because we want to discuss the well-typing of objects which do not form part of first-order Prolog, in particular the *call* and *univ* operators.

The definition of *call* is based on the fact that most Prolog implementations use the same set of symbols for predicates and functors (this causes no syntactic ambiguity) and thus a Term has a naturally corresponding Atom. Hence *call* is defined to be that predicate such that call(X) is equivalent to Y where Y is the Atom corresponding to the Term X. Thus *call* provides a method of evaluating a Term which has been constructed in a program and is accordingly related to EVAL in Lisp. We would like to argue that such a predicate is more powerful than is required and indeed encourages both bad programming style and inefficient code. It is certainly the case that most uses of *call* are used in the restricted case of applying a certain functor passed as a parameter to arguments determined locally (as in mapping predicates). Functions or predicates like EVAL or *call* do not appear to have sensible types and are thus generally omitted from strongly typed languages in favour of some form of APPLY construct.

We would like then to change our definition of Prolog and its typing to introduce this construct. To do this we introduce a family of abstract data types, called

$$\text{pred}(\alpha), \quad \text{pred}(\alpha, \beta), \quad \text{pred}(\alpha, \beta, \gamma), \ldots$$

and a family of predicates with types given by

$$\textbf{pred}\ \text{apply}(\text{pred}(\alpha), \alpha), \quad \text{apply}(\text{pred}(\alpha, \beta), \alpha, \beta), \ldots$$

The only way to introduce objects of type pred is by a special piece of syntax given by

$$\text{Term} :: = \lambda\ \text{Var}^* \cdot \text{Atom}\,.$$

The intended meaning of a value such as $\lambda(X, Y) \cdot \text{append}(\_, X, Y)$ is "that dyadic predicate which is true if and only if its first parameter is a terminal sublist of its second". The term $\lambda(X_1, \ldots, X_n) \cdot p(t_1, \ldots, t_k, X_1, \ldots, X_n)$ receives type $\text{pred}(\tau_{k+1}, \ldots, \tau_{k+n})$ if p has type $(\tau_1, \ldots, \tau_{k+n})$. Such values belong to an abstract data type and can (eventually) only be used in the predicate family *apply*. The above notation is borrowed from the λ-calculus and is at some variance with the commonly implemented form (see [8] for more discussion) which unfortunately requires symbol tables to be kept even in compiled Prolog and does not appear to fit into a notion of type. Our syntax makes it clear that the intended use (the *only* use permitted by the type rules) of append(_, X, Y) is as an Atom and not as a Term with a coincidental representation. This means that the value of such a Term can be represented as a code (or possibly closure) pointer rather than the Term structure thus avoiding the need for run time symbol tables. It can be shown that such a scheme is type secure. For example, the map predicate can be defined and used by:

$$\text{map}(F, \text{cons}(A, L), \text{cons}(B, M)) \leftarrow \text{apply}(F, A, B), \text{map}(L, M)$$
$$\text{map}(F, \text{nil}, \text{nil}) \leftarrow$$
$$\text{neglist}(X, Y) \leftarrow \text{map}(\lambda(A, B) \cdot \text{negate}(A, B), X, Y)$$

assuming that negate is defined as a dyadic predicate. The type of map so defined would be $(\text{pred}(\alpha, \beta), \text{list}(\alpha), \text{list}(\beta))$.

The other higher-order object frequently used is the *univ* predicate (often written '=..') which can be used to transform a Term into a list of Terms derived from the former's top-level substructure. (This is typically used for analysing terms read with input functions.) Thus

$$\text{univ}(f(g(X, a), Y), [f, g(X, a), Y])$$

is true. As it stands this clearly breaks the type-scheme we are proposing since the elements of the list represented by the second parameter need not be of the same type. We observe again, that such a predicate is not commonly used in its full generality, but rather to allow arbitrary terms to be input. As such, we suspect that introducing a new type 'input_term' which specifies the type of objects generated by input routines and giving *univ* the type (input_term, list(input_term)), together with a notation for treating a Term as an input_term would give much of the power of *univ* within a strong typing discipline.

The difficulty of typing *univ* arises from the conflation of object and metalevels in one language, which requires the same object to *simultaneously* possess at least two types, in a stronger sense than overloading. A satisfactory resolution of this problem waits on the introduction of an explicit metalevel or the construction of a genuinely reflective Prolog [7].

11. Conclusions

We have shown how to well-type that subset of Prolog described by first-order logic and indicated how this might be extended to allow higher-order objects. It is an interesting result that the well-typing problem for a Prolog program can itself be regarded as a Prolog metaprogram.

REFERENCES

1. Apt, K.R. and van Emden, M.H., Contributions to the theory of logic programming, *J. ACM* **29**(3) (1982) 841–862.
2. Burstall, R.M., MacQueen, D. and Sannella, D.T., HOPE: an experimental applicative language, in: *Conference Record of the 1980 LISP Conference* (1980); also: Internal Rept. CSR-62-80, Department of Computer Science, Edinburgh University, Edinburgh, 1980.
3. Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. and Wadsworth, C., A metalanguage for interactive proof in LCF, in: *Proc. 5th ACM Symp. Principles of Programming Languages*, Tucson, AZ, 1978.
4. Milner, R., A theory of type polymorphism in programming, *J. Comput. System Sci.* **17**(3) (1978) 348–375.
5. Mishra, P., Towards a theory of types in Prolog, in *Proc. IEEE Internat. Symp. Logic Programming*, Atlantic City, 1984.
6. Mycroft, A. and O'Keefe, R.A., A polymorphic type system for Prolog, DAI Research Paper 211, Dept. of Artificial Intelligence, Edinburgh University, 1983.
7. Smith, B.C., Reflection and semantics in a procedural language, Ph.D. Thesis, MIT LCS, Cambridge, MA, 1982.
8. Warren, D.H.D., Higher order extensions to Prolog—are they needed? DAI Research Paper 154, Dept. of Artificial Intelligence, University of Edinburgh, 1981.

Received August 1983