TYPES, MODULARISATION AND ABSTRACTION IN LOGIC PROGRAMMING

George Dayantis

June 1987

Research papers

Cognitive Studies Programme

Serial No. CSRP. 089

The University of Sussex,
School of Cognitive Sciences.

TYPES, MODULARISATION AND ABSTRACTION IN LOGIC PROGRAMMING

George Dayantis

ABSTRACT

Although the concepts of typing, modularisation and data abstraction have already proved their usefulness in modern programming languages, bgic programming does not seem to have paid enough attention to them. A simple and efficient way that these ideas can be realised in a bgic programming framework, which also remains within the spirit of bgic programming, is proposed here. A polymorphic type system with subtypes forms the basis for modular structure, which also supports object-oriented programming. The ideas are presented as an extension to PROLOG, which is taken as a practical representative of logic programming. Additionally, they have been implemented in a skeleton language on top of standard Probg.

June 1987

1. INTRODUCTION

Logic programming has been an attempt to offer a solution to the software crisis. By making a clear separation between logic and control and by offering both simple declarative and operational semantics, it greatly simplifies software production. Logic programs are easier to understand, modify and reason about, due to their declarative reading based on some familiar logic. Thus far, logic programming has given rise to a practical tool, the programming language PROLOG, which is based on the Horn clause subset of first-order logic. Despite its apparent restricted expressive power and its controversial 'impure' features, PROLOG has become very popular and has already been used for large applications - especially in A.I. for the implementation of expert systems. In such applications a serious deficiency of PROLOG becomes evident, namely, its lack of a) any typing facilities, which is a source of errors that are difficult to detect, and b) any structure-imposing mechanisms, which renders large programs unmanageable. Thus, it seems that the logic programming school has virtually ignored all the main advances in conventional software development tools.

Drawing experience from the lessons learned from conventional software tools one realises that if PROLOG or any of its successors is to be successful as a tool for serious software development, it has to somehow support the concepts of typing, modularisation and data abstraction without, of course, seriously upsetting the principles of logic programming. This has only lately been realised by various researchers, who have considered the possibilities and advantages of incorporating such features into a logic programming language. Apart from MPROLOG [Domolki & Szeredi 1983], whose 'modules' simply offer syntactic structuring, it is worth mentioning the influential work of Goguen et al with EQLOG [Goguen & Meseguer 1984] - a language combining all the features of OBJ with functional and Horn-logic programming -, the Japanese response with Himiko [Furukawa, Nakajima & Yonezawa 1983], Zaniolo's object-oriented extension to PROLOG [Zaniolo 1984] and the most recent proposal to 'dress' PROLOG with ML's module system [Sanella & Wallen 1987].

A rather different, simple and efficient way of incorporating the features of modularisation, parameterisation and data abstraction in a logic programming language like PROLOG is proposed here. The basis of the proposal is the notion of 'type', so a polymorphic type system with subtypes for PROLOG is studied separately in the following section, while the central

notion of a 'module' and its features are detailed in section 3. Implementation issues are discussed in section 4 and a discussion and comparison with relevant work follows in section 5.

2. A POLYMORPHIC TYPE SYSTEM FOR PROLOG

Based on an untyped logic, PROLOG is naturally an untyped language. We can see it as having a single type - the term. This allows for much flexibility and is useful for fast prototyping. However, this is also a deficiency when using PROLOG for building large systems, since type errors can be detected only at run-time. An indication of how undesirable this can be is given by the traditional definition of the *append* relation, which is intended to have meaning only with lists for its arguments but from which *append([],1,1)* can be deduced. Additionally, and as far as reasoning about logic programs is concerned, an axiomatisation and explicit use of data types in logic - as firstly proposed in [Clark & Tarnlund 1977] - has been proved invaluable (see also [Dayantis 1987]). On the other hand, the incorporation into logic programs of explicit (run-time) type information in the form of predicates generally leads to longer and more complicated proofs and deductions (see [Walther 1983]).

The need for a type system for Prolog has been realised by various researchers leading to diverse proposals [Mycroft & O'Keefe 1983, Mishra 1984]. The question is: can we supply PROLOG with a type system without seriously upsetting the principles of the language? And the answer is yes. From the theoretical point we simply need a shift to *many-sorted* logic (or even better *order-sorted* logic[1]). From the implementation point we only need an enhancement of the compiler with a typechecker. Finally from the user's point we require a type discipline - the user needs to decide and define the types for his predicates.

The type system proposed here is an extension of the one reported in [Mycroft & O'Keefe 1983]. It is presented informally in the following two subsections, while section 2.3 deals with typechecking. A grammar for the type expression sublanguage and formal syntax and semi-formal semantics for the type definitions are given in the Appendix.

2.1. Types as sets.

The notion of a type or sort should be familiar to programmers. Intuitively, a type can be understood as a set of values; in our case as a subset of the Herbrand Universe. The phrase

---

(1) In [Smolka 1986] it is shown how the semantic and deductive methods developed for untyped logic generalise to order-sorted Horn logic.

having a type or *being of a type* is then interpreted as membership in the appropriate set and is denoted by *V:T (V is of type T)*. Thus, in order to define a type we can use the same techniques as for a set, i.e. by enumeration of its elements and/or by the use of predicates. For example, we could think of defining the familiar type of lists as: " *type list(T) iff [] | [T\|list(T)]*". Without insisting on the syntax, we interpret this type definition as: "a term is a list of T's iff it is the empty list ([]) or of the form [XR], where X is of type T and R is again a list of T's".
This is an inductive definition of the enumeration kind. The alternative term structures ([],[_|_]) used in such a type definition will be referred to as *subpatterns*, while the set of all subpatterns will make up a *pattern*. Furthermore, in this example, the notions of a *generic type* and *polymorphism* are introduced. The variable *T* is a *type variable*, that can stand for any type. So with a single (generic) definition we define a family of types - lists of anything. By instantiating *T* with a specific type, say *integer*, we get *lists of integers* and so on. Notice that the term '[]' has more than one type - *type polymorphism*.

In order to see how predicates can be used for defining types, first notice that any (untyped) predicate implicitly specifies types for its respective arguments - the sets of values that satisfy it. A simple example would then be the definition of the familiar type of integers, which is built-in in Prolog : " *type integer iff (X:any with integer(X)) | (X1:integer + X2:integer) | (X1:integer * X2:integer) | ....*". This reads: "anything that satisfies the unary predicate *integer* or is a sum of integers or a product of integers or ... is of type integer".
The type *any* is the union of all types, the Herbrand Universe, and its role here will be clarified later.
Consider also the generic type of all lists with a specific *length*:
" *type list(T,N:integer) iff L:list(T) with length(L,N).*".
Notice that, apart from *type variables*, *value variables* - variables that stand for terms, such as *N* and *L* in the above example - are also allowed in a type definition. Such variables can also be annotated with their types.
Any defined predicate or conjunction of these can be used after a *with* keyword with the obvious restriction that no new variables are introduced.

The reason, of course, for structuring our universe in this way is that we may restrict the arguments of all defined relations to range over specific types. Thus, we may add declarations of the form :
" *rels append(list(T),list(T),list(T))*.", " *rels length(list(T),integer)*.",
before the actual definition of the relations involved. Such declarations will be referred to as

relation declarations. What this effectively means is that we accept partially defined relations. For example, a PROLOG variable as an argument of a relation is understood to range over the whole domain (can match anything at all), while if the relation is declared to be defined only for arguments of a specific type then any attempt to match with terms that are not of this type will fail. The essential point is that such a match will not even be attempted (at run-time), since the burden of typechecking can be placed at compile-time.

2.2. Subtypes.

We can now impose some more structure in our typed Universe by considering the relationships between different types. Two types can either be disjoint or have a common subset. The most interesting relation between types is the *subtype* relation; that is, when a type is a subset of another type. Under this set inclusion relation (denoted by $<$) the set of all types forms a lattice, whose top element is the type *any* and the bottom one is the empty set. Thus, if $V{:}T1$ and $T1 < T2$ then $V{:}T2$. From this follows the property that makes this relation interesting: any relation defined over a type $T2$ works for members of any subtype $T1$.

It is obvious now that any type is a subtype of *any* and also that *list(T,N)* $<$ *list(T)* for all types T and integers N.

Another way of inducing a subtype relation is introduced by allowing a kind of equality between subpatterns of different types. For example, if we wished to give a different representation to lists of a specific length, say as a pair of a list and an integer (its length), we could define:

" *type list(T,N:integer) iff ([],0) as [] | (([(X:T)(L:list(T))],N) with length([XL],N)) as [XL]*".

This definition still implies that *list(T,L)* $<$ *list(T)*. That means that despite the different external representation we still want these objects (of the new type) to be regarded as lists and have all the operations (relations) on lists available for them. Of course, this 'equality' between subpatterns should encompass the whole pattern of the type definition in order to induce a subtype relation. Furthermore, this can be extended to accommodate 'equality' of a subpattern with more than one subpatterns (of different types), thus allowing a type to be a subtype of more than one disjoint type (*multiple inheritance*).

The subtype relation can also be used to restrict a type variable-parameter in a type definition. For example, if we wished to define just lists of anything whose type is a subtype of integers, we could simply use :

" *type int_list(T<integer) iff [] | [T|int_list(T)]*.".

Under this definition the expression *int_list(real)* would not be a legal type.

Although subtypes introduce some extra complication in typechecking. as v>e shall see in the following subsection, they also provide more expressive power and support for the basic concept of *inheritance* in *object-oriented* programming.

Z3. Typechecking.

Typechecking a logic program, with respect to the relation declarations given for it, consists of ensuring that all relations are used with the correct type and any answer-substitution uill return the correct type. The notion of a *well-typing* for a program is formalised in a manner similar to [Mysroft & O'Keefe 1983]. Hov^ver. in our case some run-time concepts need to be incorporated in order to cope with subtypes. Additionally, this formalisation ignores predicate constraints (introduced with *with),* since they cannot be taken into account at compile-time. The way these are handled is explained in section 4.

A simple example should be sufficient to illustrate the problem introduced with subtypes. Assume the declarations: " *rels PI(integer),*[1]* and " *rels P2(real)*,* where *integer* < *real,* and the single clause: **P1(X)* >• *P2(X)*\ Do we want this to be welt-typed ? Notice that if Pi is called v»ith X instantiated (to an integer) then computation can safely proceed, but if it is called with *X* uninstantiated then there is the danger that *P2(X)* may succeed and return a real (non-integer) value for X which violates the type-restrictions for *PL* Thus, we need to carefully restrict the notion of a well-typing, taking into account the instantiation states of shared variables, when a conflict between subtypes occurs. Although this may seem to result in the loss of the full power of the logical variable, this pov^er is not really required in the cases where such restrictions have to be imposed

Due to the fact that in a PROLOG program all variables are local to a clause \ke have : A logic program (set of clauses) is *well-typed* iff each of its clauses is *well-typed.*

A clause is *well-typed* iff it is *well-typed* under all different allowable *calling modes.* So, we now need to define the well-typing of a clause with respect to a particular calling mode.

Let *Q* be a clause or a goal *PT* an association of extended types to all the predicates in *Q* and of extended types to all the terms in *Q* and *VT* an association of type/literal pairs to all the variables appearing in *Q.* Each variable in *Q* is associated with a) an instance of its expected type in the literal where it gets most instantiated - after successful execution for the chosen calling mode - and b) the literal itself ; if it doesn't get instantiated at all we assume a dummy literal *lit.*

For each predicate *a* of arity *k* in *Q, PT* will contain an element of the form *aft i*~-tu)-*

For each functor/of arity *k inQ. FT* will contain an element of the fonn *jti^..,ty):L*
For each variable Xin *Q, VT* will contain an element of the form *X.tj/P.*

DEFINITION: *// VT, FT* and *Q* are as above, we say that *VTisa well-typing ofQ under FT.* and denote it by *VT/PT V Q.* iff:

Q - B_Q <- Bj. B^.-, B_m (where m* 0 if no body)
BQ « p(aj. .... a^) (where fc* 0 if no argiiments)
pCt$_1$ .... ^) E PT
VT/PT \-$_Q$ (a$_i$-: tj) (i« l....k) and
VT/PT K$_1$. B$_i$. (i~ 1.....m)

or

Q) - *Xi* D T5 ftw/i
i3^ , O/s..»«. D ano
VT/PT *-. B. (i- 1.....m)

where:

VT/PT k A (A is an atom in the body of a clause) iff
A « p(aj. .... a-^) (where k« 0 if no arguments)
Ktj. . . . .^) E PT
VT/PT l-j (aj: Sj) (i- l.....k) and
Ks?) •» tj. i« l.....k. for some substitution r.

VT/PT k (u : s) (u is a term) iff
u «• f(aj. .... ap (wfcere k« 0 if u is a constant)
f(tj. . . . .t ^ t E PT and
VT/PT l-j (a$_1$: s$_1$) (i- 1.....k) and
Ks$_1$) « tj. i« 1.....k. and r(s) -» t
for some substitution r.

VT/PT k (X : t) (X is a variable) iff
(X:t/lit)EVT or
(Xrt/B^EVT or
(X : s / B$_i$) E VT (i - / - j) and s < t //

A desirable property we would expect from a well-typing is that in a well- typed program no predicate can ever be called with the 'wrong' type or return a term of the 'wrong' type in one of its arguments (soundness). Since the only computation (inference) rule used to execute logic programs is SLD-resolution, we effectively need to prove that SLD-resolution preserves well-typing. More specifically we prove the following:

THEOREM: (Soundness of well-typing)

If $R = \cdot A_1, \ldots, A_n$ is a well-typed resolvent (goal) and

$Q = B_0 <- B_1, \ldots, B_m$ a well-typed clause,

such that there exists a substitution $r1 = MGU(B_0, A_1)$

then the resolvent $R1 = r1(B_1, \ldots, B_m, A_2, \ldots, A_n)$ is also well-typed.

PROOF: Here we ignore the polymorphic aspect of the typing, which has already been tackled in [Mycroft & O'Keefe 1983], and concentrate on the problem with subtypes, which is orthogonal to polymorphism.

We can assume that R and Q have no variables in common (if they have we can standardise them apart by renaming).

Let PT be the basis for all typings of R and Q (associates extended types to all their literals and functors used).

We can also assume that the elements of PT have no (type) variables in common (if they have we can standardise them apart by renaming).

By our assumptions there exist typings $VT1$, $VT2$, such that:

$VT1 / PT \vdash R$ and $VT2 / PT \vdash Q$.

Consider the typing:

$VT = \{ X : t / P$, *where* X *is a variable and* $(X : t / P) \in r1(VT1 \cup VT2) \}$

From this we shall construct a well-typing VT' for $R1$ $(VT' / PT \vdash R1)$.

For every variable X in $VT1$ do (assume $X \in a_j$ and $X:t/P \in VT1$):

a) if X unifies with a term F in c_j then

1) if $P = A1$ then leave as it is;

2) otherwise ($P = Ak$ for some k, $2 <= k <= n$) for every variable Y in F substitute $Y:s/P1$ by $Y:s1/Ak$ in VT, where $s1$ is the type inferred for Y in Ak.

In both cases the well-typing of $R1$ is preserved.

b) if Xunifies with a variable *Y* in c (assume *Z» rl(Xh rl(Y))* and *Y:s/Pl E VT2)* then

1) if P- *Aj* then delete *Zt/P* from VT (and leave *Z:stPl* in VT);

2) otherwise delete *Zis/Pl* (and leave *ZA/P)* from VT.

In both cases the well-typing of *Rl* is preserved.

c) if Xdoes not unify with anything (is part of a term that unifies with a variable *Y* in c/. such that *VT2/PT V Y:sJPl)* then

1) if *P** Aj* then delete *Xt/P* from VT and add *XtllPl,* where *tl* is the type inferred for X in *PI:*

2) otherwise leave as it is.

In both cases the well-typing of *Rl* is preserved

The cases are exhaustive and the resulting VT^1 will clearly be a welt-typing for *RL*

Q.E.D.

The implementation of the typechecking algorithm, which is an extension to the one described in [Mycroft & O'Keefe 1983], is discussed in section 4. It should be noted that most of the typechecking can still be performed at compile-time and so we can have the confidence that 'well-typed programs do not go wrong*. Obviously, a type system like the one described here can be useful even in its own right Let us now see how this type scheme blends with our ideas for modularity and abstraction.

3. PARAMETERISED MODULES AND DATA ABSTRACTION

A PROLOG program is just a series of clauses. This 'flatness' of PROLOG renders lai^e programs unmanageable. Of course, the disciplined programmer can impose some structure via clause clustering and commenting. But even simple grouping facilities, such as those offered by MPROLOG [Domolki & Szeredi 1983]. do not offer much in the way of data abstraction. Here modularity is introduced via the meta-logical notion of a *module.*

In a similar way that the notion of a type structures a universe of values the notion of a *module* structures a universe of clauses. Semantically a module is something like an abstract data type. It generally consists of a *signature* part and a set of standard Prolog clauses. A module definition in its complete form centres around a single type definition, which is the main component of the signature. The set of clauses defines all relevant predicates for manipulating this type. Ideally the defined type and its associated relations constitute a conceptual unit

The syntax should be easily conveyed from the examples presented in the figures bebw, where syntax keywords are underlined and. following the Prolog convention, variable names begin with a capital letter. Formal syntax and semantics for modules is given in the appendix. The signature part in its complete form consists of:

(a) The module's name including its parameters, which are enclosed in brackets and separated by \\ It follow the *module* keyword.

(b) Names of previously defined modules that this module uses (parent modules). They follow the *using* keyword and are separated by the *and* keyword.

(c) At most one *type* definition - where the name of the defined, type always coincides with the module-name. It folbws the *pattern* keyword.

(d) The names and arities of the predicate subpattern constructors - one for each subpattern. They follow the *oonstr* keyword and are separated by bars (0.

(e) For each relation defined in the module a *relation declaration* - declaring the intended types for the arguments of the relation. They follow the *rels* keyword and are separated by commas (.).

In figure 1 bebw a simple module is presented defining the familiar generic data type *queue* with only two relations on it, followed by an enrichment of it with a further relation (dr-*adar queue).*

In figure 2, two modules representing related geometrical objects are presented, an example often used in object-oriented languages.

```
module queue(Elm)
pattern empty I q(Elm,queue(Elm))
oonstr emptjq/11 addq/3
mis read(ctElm), dequeue(ctct).
clauses
iead(q(E,empty),E).
iead(q(El,q(E2,Q)).E) :- read(q(E2.Q),E).
dequeue(q(E,empty) .empty).
dequeue(q(El.q(E2.Q)),q(El.Ql)) :- dequeue(q(E2.Q).Ql).
endmodule.
```

```
module cqueue(Elm) using queue(Elm)
rels circulate(queue(Elm),queue(Elm)).
clauses
circulate(Eq,Eq) :- emptyq(Eq).
circulate(Q1,Q2) :- read(Q1,E), dequeue(Q1,Q), addq(E,Q,Q2).
endmodule.
```

Figure 1

Notes:

(1) The module-name identifier together with the number of its parameters uniquely identifies a module.

(2) The signature part is all one Prolog 'line' (or term); thus, a '.' should be placed only at the end of it, and one after 'endmodule'. This also means that all variables in the signature are shared.

(3) In the relation declaration the special constant 'ct' is a shorthand for the current type - in the above example 'ct = queue(Elm)'.

(4) The semantics of the type definition: " *module* <*modname*> *pattern* <*pattern*>" is as given in the appendix for the syntax: " *type* <*modname*> *iff* <*pattern*>".

```
module parallelogram
pattern p(S1:real,S2:real,H:real) with S1 = < S2, H = < S1
constr makeparal/4
rels perimeter(ct,real), area(ct,real).
clauses
perimeter(p(S1,S2,H),N) :- N is 2*(S1+ S2).
area(p(S1,S2,H),N) :- N is H*S2.
endmodule.
```

pattern sq(S:real) *as* makeparal(S,S,S)
constr makesq/2.
endmodule.

Figure 2

---

(5) Each *pattern* is known only inside the module in which it is defined. Access of this pattern from another module is possible only via the constructors, the names of which are given in the 'constr' declaration. In view of this, notice in *square's* pattern how a subpattern equality (subtype relation) has to be denoted now: instead of writing " *pattern sq(S:real) as p(S,S,S)*" we use the parallelogram constructor with its last argument missing (treating it as a function). In this way, since a pattern actually provides a data representation, we ensure representation independence and facilitate data abstraction.

(6) It is meaningful to have a module without a) a pattern declaration - when it simply enriches a previous module with more relations (fig.1), b) a 'constr' declaration - when we are not interested in accessing a pattern from outside its module, c) any clauses; therefore no 'rels'-part either (fig.2).

A program now consists of a series of interdependent modules, which can be thought of as comprising an acyclic graph. We assume the existence of a built-in *root* module at the root of this graph containing all the built-in types (boolean,integer,real,list) and Prolog predicates. We could also provide - without much trouble - some library managing facilities for modules.

As for query evaluation, since the same relation name can be present in different and independent modules - with different meaning - a query evaluation makes sense only with respect to the environment of a specific module. A module's environment consists of the module itself together with the environments of its parents.

In figure 3 below a richer example of program development using modules is presented. It is the famous N-queens problem, for which a naive solution is given. The algorithm used roughly works as follows: Start with an empty board. Assume an ordering of the board's squares from left to right and top to bottom. (1) Place the next queen at the next untried square of the next free column. (2) If this results in conflict advance it one square in the same column and continue with (2) until either a safe position is reached, in which case continue from (1) or the

bottom of the column is reached, in which case backtrack and proceed from (2) with the previously placed queen. The algorithm terminates either when all queens have been placed in a safe configuration or all possible configurations have been tried and failed.

Notice that when a queen is 'placed' outside the board (an ill-typed position) computation does not terminate, but backtracking occurs, while a warning is also issued in case this violation was not intended.

The N-Queens Problem

```
module absutil
rels absdiff(real,real,real).
clauses
absdiff(N1,N2,N) :- (N1 > N2, !, N is N1 - N2) ; N is N2 - N1 .
endmodule.

module queens(N:integer) using absutil
pattern queen(N1:integer,N2:integer) with 1 = < N1, N1 = < N, 1 = < N2, N2 = < N
constr makequeen/3
rels attacks(ct,ct).
clauses
attacks(queen(N1,N2),queen(M1,M2)) :- N1 = M1 ; N2 = M2 ;
                                      (absdiff(M1,N1,D), !, absdiff(M2,N2,D)).
endmodule.

module board(N:integer) using queens(N)
pattern empty | (queens(N) * board(N))
constr emptyb/1 | addqueen/3
rels addcolumn(ct,ct), moverow(ct,ct), badboard(ct).
clauses
addcolumn(empty,(Q*empty)) :- makequeen(1,1,Q).
addcolumn((Q1*X),(Q2*Q1*X)) :- makequeen(N,M,Q1), M1 is M+ 1, makequeen(1,M1,Q2).
```

```
moverow((Q1*X),(Q2*X)) :- makequeen(N,M,Q1), N1 is N+ 1, makequeen(N1,M,Q2).

badboard(Q1*(Q2*X)) :- attacks(Q1,Q2) ; badboard(Q1*X).
endmodule.

module solvequeens(N:integer) using board(N)
rels solve(board(N)), solve(board(N),board(N)), solve1(board(N),board(N)).
clauses
solve(X) :- emptyb(E), solve(E,X).

solve(X,Y) :- addcolumn(X,Y1), ((badboard(Y1), !, solve1(Y1,Y)) ; solve(Y1,Y)).
solve(X,X).

solve1(X,Y) :- moverow(X,Y1), ((badboard(Y1), !, solve1(Y1,Y)) ; solve(Y1,Y)).
solve1(X,Y) :- addqueen(Q,X1,X), solve1(X1,Y).
endmodule.
```

Query for the 8-queens problem : ?- solve(X) in solvequeens(8).
Answer: X = queen(4,8) * queen(2,7) * queen(7,6) * queen(3,5) *
queen(6,4) * queen(8,3) * queen(5,2) * queen(1,1) * empty

Figure 3

---

4. IMPLEMENTATION

4.1. Typechecking.

A typechecking algorithm for the polymorphic type system with subtypes, which was described in section 2, has been implemented in Prolog. The implementation is an extension to the one described in [Mycroft & O'Keefe 1983] and it is only these extensions, namely the handling of subtypes and 'with' constraints, that are discussed here.

In an abstract setting the typechecker's task is to determine a *VT* given *PT* (see 2.3) for every clause in the program. Naively, this can be achieved by inferring independently for each literal in the clause a set of 'Variable:Type' associations for all the variables in the literal and then reconciling the suggested types for all shared (common) variables between two or more literals.

It should be obvious from the well-typing theorem that when a subtype-conflict occurs this type-reconciliation cannot be done (at compile-time), since the notion of instantiation states for variables is involved. Of course, one could incorporate typechecking into unification (run-time), thus resulting in a much slower interpreter. On the other extreme, one could conceivably attempt to determine at compile-time all possible allowable instantiation states for any so conflicting variables in the program. This would either require the user to supply some mode information or the system to infer it all. In any case such information would have to be in many cases tediously detailed to be of practical use. A third alternative has been chosen here, which performs as much static typechecking as possible and traps unresolved subtype-conflicts during execution, but without touching unification. How is this done?
When two consecutive appearances of a variable occur in two literals with respective assigned types *t1* and *t2*, such that $t1 < t2$ or $t2 < t1$, then an extra (system-defined) literal is inserted between them, which will test whether the concerned variable is appropriately instantiated or uninstantiated (as required for the well-typing) when computation passes that point. If the test fails it will abort with a special type-failure message.

In this way, even though one cannot be sure that a successfully compiled program is correctly typed for different possible modes of execution, any type failure due to incorrect mode-usage will be captured and reported at run-time. Additionally, this trap does not require any modification to the interpreter.

As we stressed in section 2.2 the advantage of introducing subtypes is that relations defined over supertypes work for subtypes as well. How can this be made possible?
For example, the *square* 'sq(1)' passed as an argument to *perimeter* (see section 3, figure 2) simply won't work. There are a number of messy and inefficient solutions to the problem, but fortunately there is at least one, which is elegant, efficient and also provides a solution for the 'with' constraints. This is presented below.

With each subpattern of a type definition we associate a *constructor* predicate. These are the same as the *constructor* predicates that are mostly supplied by the user in the module system described in section 3. For each such predicate name, the system defines a type-accessing

relation, e.g. for the example in fig.l it asserts:

[m]emptyq(empty).[m]. [m]addq(E[t]Q4(E,Q))"; and for the one in fig.2 :

[m]makepamL(Sl[t]S2JS[t]p(Sl[t]S2[i]H)) :- (SI « < 52, H- < Sl[t] I) ;

*(write(*Waming:Type violation*), fail)![1]*

and **makesq(Sjq(S))*> *makesq(S[f]X) ;- makepamHSJS£[t]X).*

In older to make use of these predicates a pre-processor substitutes any occurrence of a non-variable term (other than the built-in constants and constructors) in a clause with a variable and adds a Call to the appropriate type-accessing relation.

For example, the clause for 'perimeter* in figure 2 is transformed to the logically equivalent clause: ** perimeter! PjM) ymakepaml(Sl[f]S2[t]HJ^)[t] Nis 2*(S1+ S2).[m].*

Notice how the 'with' constraints are incorporated in *makeparal[f] and that violation of those causes a warning to the user rather than abortion; the advantages of this choice are illustrated by the example in figure 3.

Similarly, a goal "?- *perimeteri'sq(l)).** is transformed to: "?- *makesq(l[t]S), perimeter(S)*.* Notice that in this way all the relations defined on parallebgrams become available for squares via the second clause for 'makesq'.

4.2. The module system.

The module system described in section 3 has also been implemented in Probg. The implementation takes the form of a pre-processor that transforms modular Probg programs to semantically equivalent ordinary Probg programs, wiiile typechecking them at the same time. (Act&ally the semantics of the module system can be given in terms of this transformation - see appendix).

In order to achieve modularity each relation is augmented with an extra argument, which is the module name in which it is defined. So, identical relation names defined in different modules are naturally distinguishable. Furthermore, in this way polymorphism is also achieved (free of charge) through Probg's unification algorithm.

As for the extra-Logical predicates *assert* and *retract,* it can be easily arranged so that they have a bcal effect by augmenting their argument with the module name in which they are called This also provides a clean separation of bgic databases.

Apart from that the module and subtype hierarchies are maintained in the Probg database as well as *FT* (used for typechecking purposes - see 2.2).

As for query evaluation, every query passes through a pre-processor. which typechecks it and augments it with the appropriate module name before submitting it to the standard Probg interpreter. In order to decide on the appropriate module name for a query we have to empby the concept of the *current module.* Every single query is evaluated in the environment of the *current module.* By default the *current module* is the module that was last compiled. This can be overwritten by the user by issuing a *module-navigation* command, using the syntax: "?- -> *Module?,* which makes *Module* the current module. Additionally, one can issue a *module-specific* query using the syntax: "?- *Query* in *Module**. Such a query makes *Module* the current module temporarily (so that evaluation of *Query* takes place in its environment), but after the query is evaluated, whichever module was current before the query becomes again the current module.

Notice that since most of the expensive computation takes place at compile-time and the resulting programs are just ordinary Probg programs, there is no significant overhead in the execution speed of modular Probg programs.

5. DISCUSSION

It was taken as a starting point that even though PROLOG does not fulfill the bgic programming ideals, it can still be competitively used for lai^ge scale software production provided it is enhanced with mechanisms that reinforce modularity and data abstraction. A proposal was then outlined for such an enhancement which has also been implemented in PROLOG and whose advantages, it is believed, outweigh the overhead in computational efficiency.

The language EQLOG has already offered a similar and. in a sense, richer modular environment based on many-sorted bgic with equality, which supports Horn bgic and functional programming. However, the use of equality and term-rewriting is a major source of inefficiency, wiiile the mixture of both functional and relational styles, although it offers more expressive power, can be very confusing for the user.

Himiko. being confined to Horn bgic only, seems to be cbser to this proposal It does not have a rigid type system and so lacks all the features and advantages summed up bebw. It also albws more than one type to be defined in one module, a possibility excluded in this system so as to impose [illegible] modularity.

There have been some attempts to introduce a form of modularity with abstraction over predicates by resorting to & higher-order bgic - [Miller 1986. Nait 1986]. Although such approaches are of theoretical interest they are far from providing a practical language.

The recent proposal to embed PROLOG in the module system developed for ML [Sanella & Wallen 1987] is yet another promising approach to modularity and abstraction. In this signatures are separate from code, which allows more freedom but requires a considerable amount of extra syntax and consequently more expensive book-keeping. Although it ignores the issue of types, which could be easily added to it, it offers abstraction over both predicates and functions, whereas this system seems to offer abstraction only over functions. However, this is easily compensated in practice by the ability for separate compilation. Thus, when we want to pass a different program for a predicate *P* defined in module *M1* and used in module *M2*, we can simply recompile module *M1* with the new program for *P*, which will overwrite the previous one. Thus, we can simulate abstraction over predicates at a meta-level. Of course, if we also change the type of *P* we would also recompile *M2* to make sure that the new type is still compatible.

Finally, it should be said that, although the terminology was borrowed from the algebraic specifications school, OOP terminology could have equally well been used. Simply rename *modules* to *classes*, *instances of modules* to *objects*, the *subtype* relationship to *inheritance*, and *query evaluation* to *message passing*. Bearing this correspondence in mind, Zaniolo's proposal, a rather partial solution, is fully superseded by this system, which also supports *multiple inheritance*. Certainly, a more careful comparison is needed between this approach, or more generally the algebraic approach to abstract data types, and object-oriented programming.

However, although the principal concepts introduced are not new, they have been realised by putting together semantic features in a novel and simple way - using a minimal syntax. Furthermore, it is believed that this way is in accordance with the logic programming spirit.
In particular, the following constitute novelties:
a) the rich polymorphic type system whose main distinguishing characteristics are i) the use of explicit patterns, which together with the predicate constraints, achieve great expressive power without resorting to the use of equationally defined functions - which, apart from confusion and inefficiency, introduce the usual confluence problems. ii) the possibility of a user-definable subtype relation between different types, which extends to support multiple inheritance.
b) the 'privacy' of type patterns to their defining modules and the use of predicate type constructors - which can be defined automatically - for accessing them from other modules, which ensures data representation independence.

Even though this proposal was specifically tuned to PROLOG, it is hoped that the ideas presented here will find their way into an enhanced logic programming language that will be of practical use for serious software production.

5. ACKNOWLEDGEMENTS

I am grateful to Dr. Matthew Hennessy, Rudi Lutz and Tom Khabaza for many useful suggestions and to the State Scholarships Foundation of Greece for their financial support.

6. REFERENCES

[1] Berztiss, A. & S. Thatte. Specification and Implementation of Abstract Data Types. *Advances in Computers, Vol.22*, pp.295-353, 1983.

[2] Clark, K. & S. Tarnlund. A first order theory of data and programs. *Information Processing (IFIP) '77*, North-Holland, pp.939-944, 1977.

[3] Clocksin, W.F. & C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[4] Cardelli L. & P. Wegner. On understanding Types, Data Abstraction and Polymorphism. *Computing Surveys, Vol.17, No.4*, December 1985, ACM, pp.471-522, 1986.

[5] Dayantis George. Logic program derivation for a class of first-order logic relations. Research paper CSRP No.61, Univ. of Sussex, G.Britain, October 1986. Also (to appear) in: *Proc. 10th IJCAI*, 1987.

[6] Domolki, B. & P. Szeredi. Prolog in practice. *Information Processing (IFIP) '83*, North-Holland, pp. 627-636, 1983.

[7] Furukawa, K., Nakajima, R. & A. Yonezawa. Modularization and Abstraction in Logic Programming. ICOT TR-022, 1983.

[8] Goguen, J. & J. Meseguer. Equalities, types and generic modules for logic programming. *Second International Logic Programming Conference*, Uppsala University, Sweden, July 1984. Also in: de Groot, D. & E. Lindstrom (eds.). *Logic Programming: Functions, Relations and Equations*. Prentice-Hall, 1986.

[9] Miller, D.A. A theory of modules for logic programming. *Proc. 3d IEEE Symposium on Logic Programming*, pp.448-462, 1986.

[10] Mishra, P. Towards a theory of types in Prolog. *Proc. 1st IEEE Symposium on Logic Programming*, pp.289-298, 1984.

[11] Mycroft, A. & R. O'Keefe. A polymorphic type system for Prolog. D.A.I. Research paper, No.211, Univ. of Edinburgh, 1983. Also in: *Artificial Intelligence, Vol.23, No.3*, pp.295-307, 1984.

[12] Nait, Abdallah M. Procedures in Horn-clause programming, in: Shapiro. E. (ed.) *Proceedings of Third International Conference on Logic Programming,* London, July 1986. Lecture Notes in Computer Science. VoL 225, Springer-Verlag. pp.433-447. 1986.

[13] O'Keefe, R. Tov*ards an algebra for constructing bgic programs. *Prvc. 2nd IEEE Symposium on Logic Programming,* pp. 152-160. 1985.

[14] Sannella. D.T. & L.A. Wallen. A calculus for the construction of modular Prolog programs. (To appear:) *Prvc. 4th TREE Symposium on Logic Programming,* 1987.

[15] Smolka. G. Order-sorted Horn logic, semantics and deduction. Technical paper. FB Informatik. Kaiserslautem Univ.. W.Germany. 1986.

[16] Walther. C. A many-sorted calculus based on resolution and paramodulation. *Prvc. 8th LICAI.* pp.882-891, 1983,

[17] Zaniolo. C. Object-oriented programming in Prolog. *Prvc. 1st IEEE Symposium on Logic Pmgximming.* pp.265-270, 1984.

APPENDIX

A) Formal syntax for the type expression sublanguage.

Type > Basicjype I Constructedjype

Basic Jype :» integer I tool I.....

ConstructedJEype > Tjfunctor ITJunctoKTju^gseq)

TJfunctor > any identifier allowed as a Pro tog predicate name

T_argseq:« T^arg 1Tjrg, T^rgseq

Tjurg > T_yar I Vjyar: Type I Subpattern I T_yar < Type I Type

Tjyar, V_yar > any identifier beginning with a capital letter (Prolog variable)

Subpattern :« ...see definitions below...

B) Formal syntax for type definitions*

Type_definition :- type Type iff Patternjsonstruct

Pattem_£onstruct > Subpattern^onstruct I Subpattern_£onstruct I Pattern^onstruct

Subpattern^^onstruct :« Subpattem.jdef I Subpattem_def as Subpattern^def^onj

Subpatternjgiefjgonj:- Subpattern^cief I Subpattern^lef and Subpattem_£ef_conj

Subpattern^^ief :- Subpattern I Subpattern "with Literal^eq

Subpattern :« V_yar I V_yar: Type I Vjyar: Tjyar I Vjiinctor I V^functoKVjrgseq)

VL_argseq > Subpattern I Subpattern, Vjtxgseq

Literal_§eq > Literal I Literal, Literal_§eq

Literal > any Pro tog literal

Additionally, all free (without type annotations) value variables *(V_yar)* in *Subpattern* should appear in *Type* as *Vjxtrs.* all type variables *(Tjxtr)* in *Subpattern* should appear in *Type* as *Tjtars* and all variables (V_v*zr) in *UtervL^xq* should appear in either *Type* or the corresponding *Subpattern* as V_m*yars.*

C) Semantics far type definitions.

Below v>« denote with *Gtype(V)* a type expression that depends on the tuple of variables *V;* and similarly for *Subpattern* and *Lttend.jeq* expressions.

In general a type definition of the form:

***type* Gtype(V) *iff* (PattKV.Z1) *with* Seql(V.Zl)) I**

(Pattn(V.Zn) *with* Seqn(V.Zn)).

is given the interpretation (in first-order bgic):

X E Gtype(V) <->]Zj (X - PattjCV.Zj) & Seq^V.Zj) exor

}Z_n(X- $Patt_h$(V.Z_n)&Seq_n(V.Z_n)

Additionally, for each *as* declaration of the form: # *as* Spatt(V.Z) *with* Sseq(V.Z)' add another disjunct *}Z (X - SpattCV.Z) & Sseq(V.Z))' in the definiens. And if for all such declarations *Spatt(V,Z) E Sfype(V,Z)* then: *Gtype(V) < Stype(V,Z).*

Additionally, for each type annotation of the form *V:T* or *V<T* in *Gtype(V),* add a conjunct *isojtype(VX)* or *issubtype(V,T)* correspondingly in the definiens. And similarly for each type annotation *V:T* in a *Subpattern* add a conjunct *isojtype(V,T)* inside the appropriate disjunct (the one corresponding to the *Subpattern).*

Where:

isoftype(V.T) <-> V E T or }T1 (V E Tl & issubtype(Tl.T))

issubtype(Tl.T2) <-> Tl < T2 or }T (Tl < T & issubtype(T.T2)) or
}F,V1.V2 (Tl - .. [FtVl] & T2 - .. [FIV2] & aresubtypes(Vl.V2))

aresubtypes([],[]).
aiesubtypes([TlRl].[T2R2]) <-issubtype(Tl.T2) & aresubtypes(Rl.R2)

For convenience in expression \MB have vised the Prolog operator *univ* (« ..) to decompose the type expressions into their principal functor and a list of arguments.

D) Formal syntax for a module definition.

Module_def := **module** Type [**using** Type_andseq] Pattern_code **endmodule.**
Pattern_code := Pattern | Code | Pattern Code
Pattern := **pattern** Pattern_construct [**constr** Constr_seq]
Pattern_construct := ...(see B)...
Constr_seq := Constr | Constr | Constr_seq
Constr := Functor/Natural
Code := **rels** Rels_decl . **clauses** Clauses
Rels_decl := Rel_decl | Rel_decl, Rels_decl
Rel_decl := Functor | Functor(Type_seq)
Type_seq := Type | Type, Type_seq
Type_andseq := Type | Type **and** Type_andseq
Clauses := a set of Prolog clauses.
Functor := any identifier allowed as a Prolog predicate.

E) Semantics of a module definition.

We could choose structures like (order-sorted) algebras - with relations instead of arbitrary functions - for our semantic domain and thus give algebraic semantics to our modules as is given for standard-ML modules. Here, however, simpler translation semantics seems to be more convenient and practical, since we already possess formal semantics for standard Prolog and we claim to have means for translating modular Prolog programs to ordinary Prolog ones. Thus, a more formal exposition of this translation serves as a semantics for modular Prolog and at the same time suggests an implementation.

A *structure* is a tuple *<mname,mparents,mtype,mpreds>*, where *mname*, *mparents* and *mpreds* are identical to the corresponding syntactic objects of the language (they denote themselves) and *mtype* is a set of values.

A *module_environment* is a set of *structures.*

Our semantic functions then are:

*Mod : module_def -> module_environment -> (structure * code)*
*Mod1 : module_def -> module_environment -> (structure * code) -> (structure * code)*
Q : goal -> structure -> module_environment -> goal

In order to compile a module definition *module_def* in an initial environment *e*:

1) compute *Mod*⟦*module_def*⟧*e* obtaining a structure *s* and some Prolog code,

2) compile the code in Prolog.

Notice that the new environment is: $e' = \{s\} \cup e$.

Accordingly, in order to evaluate a goal *g* in a specific module structure *s* and an e
ment *e*:

1) compute *Q*⟦*g*⟧*s e* obtaining a new goal *g'*,

2) evaluate the new goal *g'* in Prolog.

Semantic equations:

Mod⟦module M [using Pm] Rest⟧e =
error *if* exists_module(M,e) *else*
error *if* not(allexist_modules(Pm,e)) *else*
Mod1⟦Rest⟧e (<M,Pm,{},{}>,{})

Mod1⟦pattern Pn [constr Cr] Rest⟧e (<M,Pm,{},{}>,{}) =
Mod1⟦Rest⟧e (<M,Pm,T,P>,C)
where T is the type(set) defined by Pn as in part B
and P is the set of constructor predicate declarations corresponding to Cr
and C is the set of type-accessing clauses corresponding to Cr

Mod1⟦rels Rs . clauses Cls endmodule.⟧e (<M,Pm,T,P>,C) =
error *if* some_known_rel(Rs,P,M,Pm,e) *else*
error *if* not(typecheck(Cls,P1,T,M,Pm,e)) *else*
Mod1⟦endmodule⟧e (<M,Pm,T,P1>,C1)
where P1 = {Rs} U P
and C1 = {modularise(Cls,M,Pm,e)} U C

Mod1⟦endmodule⟧e (<P,Pm,T,P>,C) = (<P,Pm,T,P>,C)

Q⟦Goal⟧<M,Pm,T,P> e = error *if* not(typecheck(Goal,P,T,M,Pm,e)) *else*
modularise(Goal,M,Pm,e)

where:

exists_module(M,e) is true iff there exists a module named *M* in *e*.

allexist_modules(Pm,e) is true iff all the modules, whose names are given in *Pm* exist in *e*.

some_known_rel(Rs,P,M,Pm,e) is true iff at least one of the relations type-declared in *Rs* has been previously defined in any of the modules that belong to the transitive closure of the parental relationship.

typecheck is obviously the typechecking predicate.

modularise(Cls,M,Pm,e) is a function returning the clauses given as its first argument with all their predicates uniquely renamed in a way that identifies them according to which module they are defined in.