COMMENTS ON "THE ADA/ED SYSTEM: A LARGE-SCALE EXPERIMENT IN SOFTWARE PROTOTYPING USING SETL" BY P. KRUCHTEN AND E. SCHONBERG

EVERYTHING YOU CAN DO I CAN DO BETTER?

A tentative assessment of Prolog features
for language processor prototyping

Peter Schnupp
InterFace Computer GmbH

Oberföhringer Str. 24a
D - 8000 München 81
F.R. Germany

1. Theme

In a profound and thought-provoking paper [KRUCHTEN] Kruchten and Schonberg discuss their approach to prototype the NYUADA compiler system using SETL as a very high level specification language. Their experience is invaluable for the following reasons:

(1) They show the feasibility of building a full, functional prototype of a very substantial software product, viz. a major language processor.

(2) They demonstrate the advantages of gradually transforming the prototype into a production version, using it as a specification, a documentation, an arbiter between different semantical interpretations and - possibly - a source of already running software for program parts not demanding for high efficiency.

(3) They point out the value of a very high level, but executable specification in an environment of uncertain and rapidly changing environment as usually encountered during the definition phase of a new language as well as of most commercial application products (against every current software engineering lore).

(4) They advance the state of the art by presenting some ingenious and certainly powerful modelling techniques, e.g. the "on the fly" code generation as a clever way to make an interpreter "behave as a compiler" down to its exception handling characteristics.

Consequently, this author cannot but support whole-heartedly their general approach as well as their observations that "there is no question of formalizing the requirements being that there is in general nothing to formalize yet" and "a technique is needed to produce WORKING SKETCHES of the system ... that are flexible enough to respond to rapid changes in requirements by rapid changes in design".

However, the specification language used, SETL, is not widely known, nor is it available in most installations. Considering the undeniable advantages of the authors' methodology, it is not a moot question whether another language could have been an equal or possibly even better choice.

2. Variation

This author's practical experience with prototyping is confined to two languages, the unix shell and Prolog. Because the shell for various reasons certainly is not a plausible candidate for this kind of modelling task, we shall limit ourselves to explore if and how Prolog could be used for similar purposes. Prolog currently is not a very widely used language either, but there exist a steady growth if its user community as well as commercially available implementations (e.g. the Prolog system under unix marketed by Springer Verlag, Heidelberg). Hence, at least it cannot be considered an "exote" , which SETL for the majority of DP users even in academic or systems software development circles still is and probably will remain.

The first item discussed by Kruchten and Schonberg is the modelling of the "static semantics" of ADA, viz. the NAME RESOLUTION mechanism. Its task is a scope dependent mapping of source identifiers into one or more UNIQUE NAMES of object entities. Here the following complications are discussed and must be modelled:

(1) Generally, a stack of open scopes and a set of used packages must be held as part of the state of comilation/execution. Identifiers must be mapped into unique names by observing inheritance from ancestor scopes in the stack and import from used packages.

(2) Due to overloading an identifier may be mapped into more than one unique name, "overloaded" in the current scope, ancestor scopes, or a used package.

It seems to us that Prolog unification could model this name resolution in an even more transparent and natural way than possible in SETL according to the example shown in the appendix of [KRUCHTEN].

In Prolog, we would store each identifier-name-relation as a fact using the scopename (constructed in an arbitrary way guaranteeing its uniqueness as functor). E.g. the binding of the identifier "x" with the unique name "un112" in the scope "s17" would

be represented by the fact

```
s17(x,un112).
```

Attributes of unique names, like types and values, could be represented by appropriate relations, e.g.

```
type(un112,integer)
```

or

```
value(un112,4711),
```

variable attributes like VALUE being updated by ASSERT and RETRACT in the course of computation.

Now, the ENVIRONMENT STACK, as the representation of currently open scopes is called in [KRUCHTEN], in Prolog is readily maintained as a list of scopes

```
[TopScope|Ancestors],
```

allowing for a recursive formulation of the name resolution function

```
find_simple_name(Id,ScopeStack,NameList)
```

instead of the "search and set-union"-algorithm in SETL which seems to us less elegant. Here, the terms needed to look for the necessary facts would be constructed from the scope names in the environment using the Prolog UNIV operator as sketched in Exhibit 1.

```
find_simple_name(Id,[Scope|Ancestors],NameList) :-
  Declared =.. [Scope,Id,UName], Declared, !,
  can_overload(UName)
  -> all_overloads(UName,Id,Ancestors,NameList)
  ;  NameList = [UName].

find_simple_name(Id,[_|Ancestors],NameList) :-
  find_simple_name(Id,Ancestors,NameList).

find_simple_name(Id,[],[]) :-
  cause_exeption('undefined name',Id).
```

Exhibit 1 : Name resolution in Prolog

The CUT in the first clause of the name resolution procedure guarantees that only the highest declaration in the scope stack will be honored.

The second item discussed in the paper is the construction of the interpreter, employing the clever "on the fly" code generation method already mentioned.

This kind of interpreting algorithm may be written down in a straightforward manner using Prolog as shown in Exhibit 2. Every clause corresponds to one interpretation step. Its first (input) argument describes the current code pattern to be interpreted, the second (output) one the new code pattern after successful evaluation of the step. One should note the elegant way the Prolog unification mechanism lends itself to realize the generation of new code text by operations like GOTO or IF as described in [KRUCHTEN].

Exhibit 2 assumes that the current environment is held in the Prolog data base and updated using RETRACT and ASSERT by each operation wishing to change the environment.

```
interprete([],[]) :-
  display('End of Program').

interprete(Program,EndState) :-
  ip(Program,NextState), !,
  interprete(NextState,EndState).

ip([[goto(Label)|BlockRest]|Rest],[Text|Rest]) :-
  findlabel(Label,BlockRest,Text), !.
ip([[goto(Label)|_]|Rest],Text) :-
  findlabel(Label,Rest,Text), !.

ip([if(LogExpr,Then,Else)|Rest],[Then|Rest)] :-
  is_true(LogExpr), !.
ip([if(_,_,Else)|Rest],[Else|Rest]).

            ......................

ip([[Op]|Rest],NewRest) :-
  ip([Op|Rest],NewRest), !.

ip([[Op|BlockRest]|Rest],[NewBlock|NewRest]) :-
  ip([Op|BlockRest],NewBlock), !,
  ip(Rest,NewRest).

ip([PlainCommand|Rest],Rest) :-
  PlainCommand, !.
```

Exibit 2 : The interpreter in Prolog

This of course is not too good a Prolog style. It would be preferable to supply each interpretation clause with two more arguments, the (input) old environment and the (output) new one.

Then we would arrive at

```
interprete(OldText,NewText,OldEnv,NewEnv)
```

as the general form of the head of each interpreter clause.

Exhibit 2 uses end recursion to implement the interpretation cycle. This of course assumes that the Prolog system used supports end recursion resolution. To inform the system that no backtracking ever will be asked for, the interpreter code should be liberally sprinkled with CUTS as shown in Exhibit 2.

If the Prolog available does not provide for automatic end recursion resolution, the interpretation cycle must be implemented using a REPEAT-FAIL loop, keeping and updating the current code text and environment in the data base. This clearly is not only less elegant and transparent but also less efficient and, hence, should be considered as a preliminary solution at most.

The ease of modelling the exception handler depends entirely on the exception handling mechanism of the Prolog system available. If it provides a good exception handler allowing for user traps bound at will to any functor (e.g. as implemented in IF-Prolog [LEIBRANDT]) the exception handling of the prototype should be straightforward and easy to model, to change, and to experiment with. On the other hand, without it its implementation should be nontrivial and clutter the interpreter with a lot of extraneous and difficult code.

3. Coda

For the prototyping tasks discussed in [KRUCHTEN] Prolog provides very appropriate linguistic means and mechanisms. Hence, to this author it seems even preferable to SETL, judging from the examples given. However, whether its theoretical advantages can be realized in a practical, major prototyping task as accomplished by Kruchten and Schonberg remains to be shown. That certainly depends on the properties of the Prolog system employed; some possible shortcomings of actual implementations were already discussed above, and others undoubtedly will be discovered when undertaking a similar project using Prolog. However, a try at it seems exceedingly worthwhile. Hopefully, we soon will hear of similar experiences using Prolog.

Literature

LEIBRANDT, U., L. BERNHARD, P. FOLKJAER, AND W. GELDMACHER:
IF/Prolog User's Manual.
München: InterFace Computer 1983.

KRUCHTEN, PH., AND E. SCHONBERG:
The ADA/Ed System: A Large-Scale Experiment in Software Prototyping Using SETL.
(In this volume.)