INTEGRATING PROLOG INTO THE POPLOG ENVIRONMENT

Chris Mellish and Steve Hardy*
Cognitive Studies Programme,
University of Sussex,
Falmer,
BRIGHTON, UK.

ABSTRACT

Although Prolog undoubtedly has its good points, there are some tasks (such as writing a screen editor or network interface controller) for which it is not the language of choice. The most natural computational concepts [2] for these tasks are hard to reconcile with Prolog's declarative nature. Just as there is a need for even the most committed Prolog programmer to use "conventional" languages for some tasks, so too is there a need for "logic" oriented components in conventional applications programs, such as CAD systems [73 and relational databases [5]. At Sussex, the problems of integrating logic with procedural programming are being addressed by two projects. One of these [43 involves a distributed ring of processors communicating by message passing. The other project is the POPLOG system, a mixed language AI programming environment which runs on conventional hardware. This paper describes the way in which we have integrated Prolog into POPLOG.

I THE POPLOG ENVIRONMENT

The POPLOG system is an AI programming environment developed at Sussex University [3D. It supports Prolog, POP-11, a dialect of POP-2 [13, and a basic LISP. POPLOG currently runs on the DEC VAX series of computers under the VMS operating system, but other implementations are in progress.

In POPLOG, the link between the programming languages and the underlying machine is the POPLOG virtual machine. The compilers produce POPLOG virtual machine instructions, which are then further translated into the machine code for the host machine. At the level of host machine code, it is possible to link in programs written in languages such as FORTRAN. Procedures for "planting" instructions for the virtual machine are fully accessible to the user. Thus the Prolog compiler is just one of the many possible POPLOG programs that create new pieces of machine code. In particular, it is easy to create procedure closures. For the purposes of this paper, a closure is a structure which contains a pointer to a procedure plus a set of arguments for that procedure. The closure can then be applied as if it were a normal procedure with no arguments. Some "syntactic sugar" has been provided in POP-11 to make it easy to create closures; an expression such as:

```
doubled 3 %)
```

evaluates to a closure which when later invoked calls the procedure DOUBLE with argument 3.

II BACKTRACKING AND CONTINUATION PASSING

In this section, we illustrate, using examples written in POP-11, how backtracking programs are implemented in POPLOG using a technique called continuation passing. Although examples are shown in POP-11 for clarity, in practice Prolog programs *are* compiled directly to POPLOG virtual machine code.

Continuation passing is a technique in which procedures are given an additional argument, called a continuation. This continuation (which is a procedure closure) describes whatever computation remains to be performed once the called procedure has finished its computation. In conventional programming, the continuation is represented implicitly by the "return address" and code in the calling procedure. Suppose, for example that we have a procedure, called PROG, that has just two steps: calling the subprocedure F00 and then the subprocedure BAZ, thus:

```
define prog();
    foo();
    baz();
enddefine;
```

We can rewrite this procedure with an explicit continuation, thus:

```
define prog(continuation);
    foo();
    baz();
    continuation()
enddefine;
```

This definition presupposes that F00 and BAZ do not themselves take continuations. If they do, then we must arrange for BAZ to be passed CONTINUATION and for F00 to be passed an appropriate closure of BAZ, thus:

*Steve Hardy is now at Teknowledge Inc, 525 University Ave, Palo Alto, CA 94301, USA.

```
define prog(continuation);
    foo(baz(%continuationX)>
enddefine;
```

Thus F00 gets as argument a closure. This closure, when applied, will cause BAZ to be invoked with CONTINUATION as its argument.

Continuations have proved of great significance in studies on the semantics of programming languages C63. This apparently round about way of programming also has an enormous practical advantage - since procedures have explicit continuations there is no need for them to "return" to their invoker. Conventionally, sub-procedures returning to their invokers means "I have finished - continue with the computation", with explicit continuations we can assign a different meaning to a sub-procedure returning to its invoker, "Sorry - I wasn't able to do what you wanted me to do".

This can be illustrated if we define a new procedure NEWPROG, whose definition is try doing F00 and if that doesn't work then try doing BAZ, thus:

```
define newprog(continuation);
    foo(continuation);
    baz(continuation);
enddefine;
```

If we now invoke NEWPROG (with a continuation) then it first calls F00 (giving it the same continuation as itself). If F00 is succesful then it will invoke the continuation. If not then the call of F00 will return to NEWPROG which then tries BAZ. If BAZ too fails (by returning) then NEWPROG itself fails by returning to its invoker.

III INTRODUCING UNIFICATION

Consider the following Prolog program:

```
happy(X) :- healthy(X), wise(X).

healthy(X) :- jogs(X).
healthy(X) :- eats(X, cabbage).

jogs(chris).
jogs(jon).
```

This says that X is HAPPY if X is HEALTHY and WISE. Someone is HEALTHY if they JOG or EAT CABBAGE; CHRIS and JON both JOG. If these are complete definitions for these predicates, then we can translate them into POP-11 as follows:

```
define happy(x, continuation);
    healthy(x, wise(%x, continuation%))
enddefine;

define healthy(x, continuation);
    jogs(x, continuation);
    eats(x, "cabbage", continuation);
enddefine;

define jogs(x, continuation);
    unify(x,"chris",continuation);
    unify(x,"jon",continuation)
enddefine;
```

UNIFY is a procedure that takes two data structures and a continuation. It attempts to unify (that is, "make equal") the two structures. If it is unsuccessful, UNIFY immediately returns to its invoker. If, however, it is successful, then it applies the continuation and when that returns, UNIFY undoes any changes it made to the two structures and then itself returns to its invoker.

Before we can present a definition of UNIFY, we must consider the representation of Prolog variables. In Prolog, variables start off "uninstantiated" and can be given a value only once (without backtracking); moreover two "uninstantiated" variables when unified are said to "share", so that as soon as one of them obtains a value, the other one automatically obtains the same value.

In POPLOG, a Prolog variable is represented by a single element data structures called a REF. REFs are created by the procedure CONSREF and their components are accessed by the procedure CONT. An uninstantiated Prolog variable is represented by a REF containing the unique word "undef". If a variable is assigned some value, this value is placed into the CONT. If two variables come to "share", we make one point to the other. To find the "real" value of a variable, especially one that is sharing, it is necessary to "dereference" it (look for the contents of the "innermost" REF).

Here now is a simple definition of UNIFY written in POP-11:

```
define unify(x,y,continuation);
  if x == y then
    continuationO
  elseif isref(x) and cont(x) = "undef" then
    y -> cont(x);
    continuationO;
    "undef" -> cont(x)
  elseif isref(x) and cont(x) /= "undef" then
    unify(cont(x),y,continuation)
  elseif isref(y) then
    unify(y,x,continuation)
  elseif ispair(x) and ispair(y) then
    unify(front(x),front(y),
       unify(Xback(x),back(y),continuation%))
  endif
enddefine;
```

The procedure first sees if the two given data structures, X and Y, are identical. If so, it immediately applies the CONTINUATION. If the structures aren't identical then UNIFY looks to see whether X is a REF and if so whether it is uninstantiated (ie. whether its CONT is the word "undef"). If so, UNIFY sets its value to Y (by assigning to the CONT; assignment works from left to right in POP-11), does the CONTINUATION and if this returns (ie fails) unbinds the REF by setting

the CONT back to "undef". The final case of UNIFY deals with the possibility that X and Y may be list pairs. A complete definition of UNIFY must have a case here for each type of datastructure recognised as a Prolog complex term. Note that there is no ELSE part to the IF statement. The default action is simply to return (ie indicate failure).

As a more complex example, here is a translation of the Prolog MEMBER predicate into POP-11. The Prolog definition is:

```
member(X,[X|Y3).
member(X,[Y|Z3) :- member(X,Z).
```

When translated into POP-11, it will be necessary to make explicit the unifications which are implicitly done when a Prolog predicate is invoked. It may therefore be easier to understand the POP-11 translation if we rewrite the Prolog definition to make the various unifications explicit:

```
member(X,Y) :- Y = [X|M3.
member(X,Y) :- Y = [L|M3, member(X,M).
```

This translates into the following POP-11 procedure:

```
define member(x, y, continuation);
  vars l; consref("undef") -> l;
  vars m; consref("undef") -> m;
  unify(y, conspair(x,m), continuation);
  unify(y, conspair(l,m),
            member(%x,m,continuation%))
enddefine;
```

The first two lines of this definition create new Prolog variables (REFS with contents "undef") L and M. The next line checks if the value of Y can be unified with a newly created pair whose FRONT is the value of X and whose BACK is the new variable M; if so, UNIFY will perform the continuation. The last line of the definition tries unifying Y with a pair whose components are the new variables L and M; if successful, UNIFY will invoke its continuation which, in this case, is a closure of MEMBER itself.

IV CONCLUSIONS

We have presented a simplified version of how Prolog is implemented in the POPLOG environment. We believe that this system provides a basis for true mixed-language AI programming because:

(1) The POP-11 and Prolog compilers are just two of potentially many procedures which generate code for the POPLOG virtual machine. This means that the two languages are compatible at a low level, without there being the traditional asymmetry between a language and its implementation.

(2) The continuation passing model provides a semantics for communication between these two Languages which allows for far more than simple "subroutine calling".

(3) The control facilities available within POPLOG (not shown here) make it possible to implement a system which is faithful to the theoretical modeL, but which is nevertheless efficient.

ACKNOWLEDGEMENTS

We would like to thank John Gibson, the main implementer of the POPLOG virtual machine and the POP-11 compiler, for providing us with a powerful programming environment, without which this work would not have been possible. We would also like to thank Aaron Sloman and Jon Cunningham for many useful discussions.

REFERENCES

[13 Burstall, R.M., Collins, J.S. and Popplestone, R.J., Programming in POP-2, Department of Artificial Intelligence, University of Edinburgh, 1977.

[23 Hardy, S., "Towards More Natural Programming Languages" Cognitive Studies Memo 82-06, University of Sussex, 1982.

[3] Hardy, S., "The POPLOG Programming Environment", Cognitive Studies Memo 82-05, University of Sussex, 1982.

[43 Hunter J.R.W., Mellish, C.S. and Owen, D., "A Heterogeneous Interactive Distributed Computing Environment for the Implementation of AI Programs", SERC grant application, School of Engineering and Applied Sciences, University of Sussex, 1982.

[5] Kowalski, R., "Logic as a Database Language", Department of Computing, Imperial College, London, 1981.

[63 Strachey, C. and Wadsworth, C.P., "Continuations: A Mathematical Semantics for Handling Full Jumps", Technical Monograph PRG-11, Programming Research Group, Oxford University, 1974.

[7] Swinson, P.S.G, "Prescriptive to Descriptive Programming: A way ahead for CAAD", in Taernlund, S.-A., Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980.