New Generation Computing, 2 (1984) 347-359
OHMSHA,LTD. and Springer-Verlag

NEW GENERATION COMPUTING

On Implementing Prolog in Functional Programming †

Mats CARLSSON
UPMAIL, Uppsala Programming Methodology and Artificial Intelligence Laboratory, Department of Computing Science, Uppsala University, P. O. Box 2059, S-750 02 Uppsala, Sweden

Received 3 September 1984

Abstract This report surveys techniques for implementing the programming language Prolog. It focuses on explaining the procedural semantics of the language in terms of functional programming constructs. The techniques *success continuations* and *proof streams* are introduced, and it is shown how Horn clause interpreters can be built upon them. Continuations are well known from denotational semantics theory, in this paper it is shown that they are viable constructs in actual programs.

Other issues include implementation of logical variables, structure sharing vs. structure copying, determinacy, builtin predicates, and *cut*.

Keywords: Continuations, Functional Programming, Interpreters, Logic Programming, Prolog.

§ 1 Achieving Backtracking

Several authors [5,10,12,19,20] have proposed abstract machinery to implement the backtracking behavior of Prolog. Typically the abstract machine includes a set of registers and various stacks carrying the state of the machine. Backtracking amounts to restoring parts of this state as it was at some previous time.

In this section, we will exploit an alternative technique for backtrack programming using concepts from functional programming. Continuations [18] are the workhorse for achieving the desired control structure. Recursion is used instead of manipulation of explicit stacks, and parameter passing is used instead

† Adapted from the paper "On Implementing Prolog In Functional Programming" by MATS CARLSSON appearing in 1984 INTERNATIONAL SYMPOSIUM ON LOGIC PROGRAMMING, February 6-9, 1984, Atlantic City, NJ, pp. 154-159. Copyright © 1984 IEEE.

of assignments to machine registers.

There are at least two fundamentally different techniques for achieving backtracking and we will call them *proof streams* and *success continuations*. In this paper, the word "continuation" denotes any function that is either passed as an argument or returned as a value.

1.1 Success Continuation Scheme

The idea here is that the theorem prover receives an extra argument, the *success continuation*. If the theorem prover succeeds in proving its goal, it calls this continuation. If it fails, it simply returns. Backtracking is achieved by the possibility that the theorem prover finds several proofs of its goal, in which case it calls the continuation for each proof found.

1.2 Proof Stream Scheme

Here, the theorem prover returns a *proof stream*, which conceptually is a lazily evaluated list of environments, corresponding to the possible proofs of the given query. In a concrete implementation, a proof stream could be either

()

for failure i.e. no more proofs, or a pair

(*environment* . *continuation*)

for success i.e. a proof was found. *Continuation* is a function that returns a new proof stream. *Environment* could be the set of variable substitutions involved in a particular proof. Backtracking is achieved by calling the continuation of successive proof streams until failure eventually results. To our knowledge this idea was first conceived by Abelson.[1] It has been used later by Kornfeld [11] and Kohlbecker.[8]

1.3 Interpreters

We present here running implementations of the two techniques written in pure Lisp.

A continuation is implemented as a Lisp function consed to an incomplete argument list. The continuation is called with additional arguments that complete the argument list, which is then passed to the Lisp function.

Prolog datatypes used are atoms, pairs, and variables. Variables are implemented as symbols that begin with a '?'. We will use MacLisp's backquote abbreviation mechanism and write

```
`(x, y z)
```

instead of

```
(list 'x y 'z)
```

Variable bindings are kept in an association list consisting of pairs

((*variable* . *index1*) . (*term* . *index2*))

where the indexes are used to distinguish between synonymous variables belonging to different uses of the same assertion. The index is increased once per resolution step. This is essentially the *structure sharing* technique of Boyer and Moore.[2] It is discussed in more detail in chapter 3.

The database is only indexed on predicate symbols. The :assertions property of a symbol contains a list of assertions.

〔1〕 Success continuation interpreter

This is a MacLisp implementation of the above ideas. It defines a Horn clause interpreter and includes predicates about appended and reversed lists.

```
(defun prove (env j goals i cont)
  ;;Proves goals seen through index i and calls
  ;;cont. J is next available index.
  (cond((null env) nil)
       ;;An impossible environment is empty.
       ((null goals) (invoke cont env j))
       (t (resolve(car goals) i
                  (assertions (car goals)) j
                  `(prove ,(cdr goals) ,i ,cont) env))))

(defun resolve (goal i assertions j cont env)
  ;;For each proof of goal with respect to assertions,
  ;;cont is called.
  (cond((null assertions) nil)
       ((prove (unify goal i (caar assertions) j env ) (1 + j)
                              (cdar assertions) j cont))
       (t (resolve goal i
                  (cdr assertions) j
                  cont env))))

(defun unify (x i y j e)
  ;;Returns a non-empty environment upon success.
  (unify1 (ult (cons x i)e) (ult (cons y j) e) e))

(defun unify1 (xi yi e)
  (cond((null e) e)
       ((variable-symbol-p (car xi)) (cons (cons xi yj) e))
       ((variable-symbol-p (car yj)) (cons (cons yj xi ) e))
       ((and (eq (car xi) (car yj)) (= (cdr xi ) (cdr yj)))
       e)
       ((and (consp (car xi)) (consp (car yj)))
```

```
        (unify (cdar xi) (cdr xi) (cdar yj) (cdr yj)
               (unify (caar xi) (cdr xi) (caar yj) (cdr yj)
                      e)))
       ((equal (car xi ) (car yj )) e)))

(defun ult (x e)
  ;;Follows chain of linked variables.
  (let ((pair (assoc x e)))
   (cond ((null pair) x)
         ((eq x (cdr pair)) x)
         (t (ult (cdr pair) e)))))
```

A predicate to append two lists could for example be represented as

```
(defprop append
  (((append (?X . ?XS) ?Y(?X . ?ZS)) (append ?XS ?Y ?ZS))
   ((append () ?X ?X)))
  :assertions)
```

Auxiliary functions include invoke for invoking continuations, variable-symbol-p for recognizing variables, assertions for retrieving appropriate assertions from the database, and a toplevel function.

[2] Proof stream interpreter

To arrive at an interpreter based on proof streams, the functions prove and resolve above should take proof streams instead of continuations as arguments, and should return proof streams as values.

The following code implements streams as ordinary lists. Under lazy evaluation this would result in the desired backtracking. The environment in the above discussion of proof streams is actually implemented as a pair of (i) the variable binding alist and (ii) next available index.

```
(defun prove (stream goals i)
  ;;Each goal "filters" the proof stream.
  (cond((null stream) ())
       ((null goals ) stream)
       (t (prove (resolve stream (car goals) i (assertions (car goals)))
                 (cdr goals)
                 i))))

(defun resolve (stream goal i assertions)
  ;;Each stream element is replaced by a new stream
  ;;incorporating the proofs of "goal".
  (cond((null stream ) ())
       (t (append (resolve1 goal i assertions (cdar stream) (caar stream))
                  (resolve (cdr stream) goal i assertions)))))
```

```
(defun resolve1 (goal i assertions j env)
  (cond ((null assertions) ())
        ((null env) ())
        (t (let ((env1 (unify goal i (caar assertions) i env)))
             (append (prove `((,env1 . ,(1+ j))) (cdar assertions) j)
                     (resolve1 goal i (cdr assertions) j env))))))
```

A concrete implementation along the lines in Section 2.2 is given in the Appendix.

1.4 Comparison

It is fairly obvious that using success continuations recurses deeper and so consumes more stack space, whereas using proof streams constructs more delayed objects and so consumes more cons space. However, by introducing in the code special cases for e.g. last conjunct or last disjunct, the behaviors of both schemes improve significantly.

Moreover both interpreters contain plenty of direct and indirect tail recursion, which of course is transformed to iterative form in "production" versions of the algorithms.

§ 2 Structure Sharing vs. Structure Copying

A major source of inefficiency in the above interpreters is the implementation of the binding environment. It is implemented as an association list without any indexing. To get the bound value of a variable one may have to search the whole list. Worse, one may have to do repeated searches in case there are variable-to-variable bindings, which leads to quadratic time complexity.

In *structure sharing* implementations such as DEC-10 Prolog [19] every use of an assertion has its own activation-record like binding environment. There is then no need to search the environment for a binding. Warren lets a base register point to the activation record and assigns offsets to variables. He is then able to get a variable binding in just one machine instruction. Structure sharing is typically implemented to let unify destructively update the binding environments.

Another method is *structure copying* in which one uses *copies* of assertions. A new copy is constructed in each resolution step. The copies contain value cells, and unify is allowed to destructively update these. Whether it is worth while to recycle the copies is an open question.

A more sophisticated variant is unification driven structure copying, where "pure code" is copied only when it is unified with a variable. This variant is used in the systems described by Mellish [13] by Warren,[20] and by Carlsson and Kahn.[3,4,7]

With destructive changes, the need to undo these arises. Structure sharing and structure copying implementations typically use a *reset list* or *trail* to record

all variable bindings, in order to know what to undo upon backtracking.

The cost of constructing copies of assertions should be weighted against the relative complexity of structure sharing implementations where terms always must be "seen" through an index (a pointer to a binding environment). This means that twice as many arguments have to be passed around in the inner loop of the interpreter.

On computers with indirect addressing support in Lisp one can even make pointers to value cells totally transparent. This is a very attractive feature since it drastically reduces the cost of interfacing Prolog to Lisp, which one typically wants in a Lisp-based Prolog system. An extensive comparison of structure sharing vs. structure copying has been done by Mellish.[13]

We will now further refine the success continuation technique to use structure copying. It is left as an exercise to the reader to implement indexed structure sharing.

2.1 Structure Copying Interpreter

Value cells are represented here as pairs

(var . *value*)

The *value* field of an unbound value cell points to the cell itself, so that the unifier can get hold of the value cell after dereferencing.

The following is the central parts of a success continuation interpreter that uses "naïve" structure copying. Note here that an assertion is represented as *code* to construct a copy of the assertion in question.

```
(defun prove (goals cont)
  ;;Proves goals and calls cont.
  (cond (goals (resolve (car goals `(prove ,(cdr goals) ,cont)))
        (t (invoke cont))))

(defun resolve (goal cont)
  ;;For all proofs of goal, cont is called.
  (try-assertions goal (assertions goal) *trail* cont))

(defun try-assertions (goal assertions mark cont)
  (cond (assertions
         (cond ((try-assertion goal (car assertions) cont))
               (t (reset mark)
                  ;;Unwind trail back to mark.
                  (try-assertions goal (cdr assertions) mark cont))))))

(defun try-assertion (goal assertion cont)
  (cond ((unify goal (funcall (car assertion)))
         (prove (funcall (cdr assertion)) cont))))
```

```
(defun unify (x y)
  (cond ((eq x y))
        ((variable-p x) (unify-variable x y))
        ((variable-p y) (unify-variable y x))
        ((and (consp x) (consp y))
         (and (unify (dereference (car x)) (dereference (car y)))
              (unify (dereference (cdr x)) (dereference (cdr y)))))
        ((equal x y))))

(defun unify-variable (x y)
  ;;Unifies a variable with a term.
  (progn (push x *trail*) (rplacd x y)))

(defun dereference (x)
  ;;Follows chain of linked variables.
  (cond ((variable-p x)
         (cond ((eq x (cdr x)) x)
               (t (dereference (cdr x)))))
        (t x)))
```

The global variable *trail* holds the reset stack. The function cell creates a value cell. The function reset restores value cells to the unbound state. Variable-p is a predicate for recognizing value cells.

Each assertion in the database is represented by a pair of functions, where the first element constructs the head of an assertion and the second element constructs the body. This is exemplified by the database entry for append:

```
(defprop append
  ((app-1-1 . app-1-2) (app-2-1 . app-2-2))
  :assertions)
(defun app-1-1()
  (setq ?X (cell)) `(append () ,?X ,?X))
(defun app-1-2() `())
(defun app-2-1()
  (setq ?X (cell) ?XS (cell) ?Y (cell) ?ZS (cell))
  `(append (,?X . ,?XS) ,?Y(,?X . ,?ZS)))
(defun app-2-2 () `((append ,?XS ,?Y ,?ZS)))
```

§ 3 A Note on Determinacy

It should be noted that the stack space consumption of the above success continuation interpreters can be much reduced if prove can test whether a goal is determinate. The modified prove procedure for the structure copying case is:

```
(defun prove (goals cont)
  (cond((null goals) (invoke cont))
```

```
((determinate (car goals))
 (and (resolve (car goals) '(true)) (prove (cdr goals) cont)))
(t (resolve (car goals) '(prove ,(cdr goals ) ,cont)))))
```

§ 4 Adding Builtin Predicates

In addition to the pure Horn clause theorem proving capabilities, any Prolog implementation needs builtin predicates. This can be done e.g. in the following way: Let the :*assertions* property of the predicate symbol be the pair

(:builtin . *function*)

where *function* accepts two arguments: a goal and a continuation. Resolve needs to take care of this case. As an example, we show here how to implement bagof with this technique.

```
(defun resolve (goal cont)
  (let ((assertions (assertions goal)))
    (cond((eq ':builtin (car assertions))
          (funcall (cdr assertions) goal cont))
         (t (try-assertions goal assertions *trail* cont)))))

;;(bagof ?t ?p ?b): ?b is the bag of all ?t such that ?p holds
(defprop bagof (:builtin . bagof-prover) :assertions)

(defun bagof-prover (goal cont)
  (let ((mark *trail*)
        (reslist (list ())))
    ;;Reslist collects the result.
    (resolve ((third goal) '(bagof-aux ,(second goal) ,reslist))
    (reset mark)
    (cond((unify (dereference (fourth goal))
                 (nreverse (car reslist)))
         (invoke cont)))))

(defun bagof-aux (term reslist)
  (push (instantiate (dereference term)) (car reslist))
  nil)
```

where instantiate is a function that copies its argument, removing bound value cells and replacing unbound value cells by fresh ones. This is necessary since different proofs of ?p may assign different values to value cells in ?t.

Note that bagof-aux always returns nil. This is to force the theorem prover to really find all proofs of ?p. In try-assertions, the value returned from the non-tail-recursive call to prove is tested, and if non-nil, no more assertions are tried. This is used by the toplevel function prove so that the user can stop the search at a particular proof. It also prepares for the issue coming up in the next section.

§ 5 Adding Cut

The cut control primitive needs extra machinery. The test in try-assertions mentioned above offers the control alternatives "find all proofs" vs. "find first proof" for a given goal. Cut, however, is more complex because it is lexically scoped, and so at run time one needs some device that can mimic lexical scoping. One such device is to keep track of the ancestor depth of the current and-tree node. Instead of returning nil for failure, the theorem prover and builtin functions can return an integer specifying an ancestor depth to return to. These ideas are implemented as follows.

```
(defun prove (goals cont d)
  ;;Proves goals at depth d and calls cont.
  (cond (goals (resolve (car goals) '(prove ,(cdr goals) ,cont ,d) d))
        (t (invoke cont))))

(defun resolve (goal cont d)
  (let ((assertions (assertions goal)))
    (cond ((eq ':builtin (car assertions))
           (funcall (cdr assertions ) goal cont d))
          (t (try-assertions goal assertions *trail* cont d)))))

(defun try-assertions (goal assertions mark cont d)
  (cond (assertions
         (let ((msg (try-assertion goal (car assertions) cont d)))
         ;;This code returns msg to the right level.
         (cond ((< msg d)
                (reset mark)
                (try-assertions goal (cdr assertions) mark cont d))
               ((= msg d)*failure*)
               (t msg))))
        (t *failure*)))

(defun try-assertion (goal assertion cont d)
  (cond ((unify goal (funcall (car assertion)))
  ;;The depth is increased in each resolution step.
         (prove (funcall (cdr assertion)) cont (1+ d)))
        (t *failure*)))

(defprop cut (:builtin . cut-prover) :assertions)
(defun cut-prover (ignore cont d)
  ;;Upon backtracking, cut fails the parent goal.
  (invoke cont) (1- d))
```

The constant *failure* contains a large integer.

An interesting alternative, exploited in eu-Prolog,[8] is to understand cut

as an invocation of a continuation.

§ 6 Prolog

To extend the toy interpreters of this paper into full-fledged Prolog systems is straight forward and has been done. The resulting system is comparable in speed with interpreted DEC-10 Prolog and is available from the author.

§ 7 Conclusions

We have surveyed techniques for implementing Prolog interpreters in Lisp. In particular, we have implemented logic programming in a functional setting, turning functions into relations by using continuations.

§ 8 Related Work

The Lisp-based Prolog implementations are too numerous to be treated in detail. We will however compare our work with some of the more well known.

- QLOG [9] was perhaps the first Lisp-based interpeter. It was based on structure sharing and defined Prolog predicates as Lisp *fexprs*. Komorowski was the first to point out the potential of inheriting a programming environment by embedding Prolog in Lisp.
- LogLisp [17] is an interpreter incorporating breadth-first and heuristic search mode and a notion of reducibility : Every term or predication that can be interpreted as a Lisp form is evaluated as such before resolution happens.
- Prolog/KR [15] is an implementation with novel control language constructs. Nakashima has argued for a list-based syntax since it provides equivalence between program and data and thus ease of using program manipulating programs such as structure oriented editors.
- Foolog [16] is a beautiful example of a very concise system consisting of an interpreter written in MacLisp together with a compiler written in Foolog.
- eu-Prolog [8] is an interpreter written in Scheme in which both backtracking and cut are seen as invocations of continuations.
- POPLOG [14] is a trilingual programming environment supporting Prolog, Lisp, and POP-2. The Prolog part is compiler-based using a variant of success continuations. In particular, their unifier takes a continuation.
- LM-Prolog [3,4,7] is a ZetaLisp implementation for MIT Lisp Machines [6] based on structure copying and success continuations. It consists of a compiler and an interpreter written in ZetaLisp and a microcoded unifier.

Acknowledgements

The author would like to thank the referees, Dr. Kenneth Kahn, and Dr. Ehud Shapiro for many valuable comments on this paper. We gratefully acknowledge the support of the National Swedish Board for Technical Development for making possible the research reported herein.

References

1) Abelson, H. and Sussman, G. J.: "Structure and interpretation of computer programs," Draft (Department of EE and CS, MIT) (July, 1983) Sections 4.3 and 4.4.
2) Boyer, R. S. and Moore, J. S.: "The sharing of structure in theorem proving programs," Machine Intelligence, *7* (ed. Meltzer and Mitchie) (Edinburgh UP) (1972).
3) Carlsson, M.: "LM-Prolog — The Language and Its Implementation," UPMAIL Technical Report (Computing Science Department, Uppsala University) (forthcoming 1984).
4) Carlsson, M. and Kahn, K. M.: "LM-Prolog User Manual," UPMAIL Technical Report, *No. 24* (Computing Science Department, Uppsala University) (1983).
5) van Emden, R. H.: "An Interpreting Algorithm for Logic Programs," Proc. 1st International Logic Programming Conference (Marseille) (1982).
6) Greenblatt, R.: "The Lisp Machine," MIT AI Lab Working Paper, *79* (Cambridge MA) (Novemember, 1974).
7) Kahn, K. M. and Carlsson, M.: "How to Implement Prolog on a Lisp Machine," in Implementations of Prolog, J. Campbell (ed.) (Ellis Horwood Ltd., Chichester) (1984).
8) Kohlbecker, E.: "eu-Prolog," Technical Report, *No. 155* (Indiana University, Computer Science Department, Bloomington IN) (April, 1984).
9) Komorowski, H. J.: "QLOG — The Software for Prolog and Logic Programming," in Logic Programming, K. Clark and S.-Å. Tärnlund (eds.) (Academic Press, London) (1982).
10) Komorowski, H. J.: "A Specification of an Abstract Prolog Machine and Its Application to Partial Evaluation," Ph. D. Thesis (Linköping University, Linköping) (1981).
11) Kornfeld, W. A.: "Equality for Prolog," Proc. 8th IJCAI (Karlsruhe) (1983).
12) McCabe, F.: "Abstract PROLOG machine — a specification" Department of Computing, Imperial College, London) (1983).
13) Mellish, C. S.: "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter," in Logic Programming, K. Clark and S.-Å. Tärnlund (eds.) (Academic Press, London) (1982).
14) Mellish, C. S. and Hardy, S.: "Integrating Prolog in the POPLOG environment," in Implementations of Prolog, J. Campbell (ed.) (Ellis Horwood Ltd., Chichester) (1984).
15) Nakashima, H.: "Prolog/KR Language Features," Proc. First Logic Programming Conference (Marseille) (September, 1982).
16) Nilsson, M.: "Foolog — A Small and Efficient Prolog Interpreter," Technical Report, *No. 20* (UPMAIL, Uppsala University) (June, 1983).
17) Robinson, J. A. and Sibert, E. E.: "Logic Programming in LISP" (School of Computer and Information Science, Syracuse University) (1980).
18) Strachey, C. and Wadsworth, C. P.: "Continuations —A Mathematical Semantics for Handling Full Jumps" (Programming Research Group, Oxford University) (1974).

19) Warren, D. H. D.: "Implementing Prolog — compiling predicate logic programs" (Department of Artificial Intelligence, University of Edinburgh) (1977).
20) Warren, D. H. D.: "An Abstract Prolog Instruction Set" Technical Note, *309* (Artificial Intelligence Center, SRI International) (October, 1983).

Appendix : Proof Streams with Continuations

The following is the central part of a proof stream interpreter as was discussed in section 2.2. All of these functions return proof streams which are either

()

for failure i.e. no more proofs, or a pair

(*environment* . *continuation*)

for success i.e. a proof was found. *Continuation* is a function that returns a new proof stream. *Environment* is a pair of (i) the variable substitutions involved in a particular proof and (ii) next available index. Backtracking is achieved by calling the continuation of successive proof streams until failure eventually results. All of these functions return proof streams.

```
(defun prove (env newi goals oldi)
  ;;Returns the proofs of goals in env.
  (cond ((null env) ())
        ((null goals) `((,env . ,newi) . (false)))
        (t (invoke-in-each
             (resolve (car goals) oldi (assertions (car goals)) newi env)
             `(prove ,(cdr goals) ,oldi)))))

(defun resolve (goal oldi assertions newi env)
  ;;Returns the proofs of goal in env.
  (cond ((null assertions) ())
        (t (invoke-after
             (prove (unity goal oldi (caar assertions) newi env)
                    (1 + newi)
                    (cdar assertions)
                    newi)
             `(resolve ,goal ,oldi ,(cdr assertions) ,newi ,env)))))


(defun invoke-after (stream1 cont)
  ;;Appends cont onto the stream stream1.
  (cond ((null stream1) (invoke cont))
        (t (cons (car stream1) `(invoke-after* ,(cdr stream1), cont)))))
```

```
(defun invoke-in-each (stream1 cont)
  ;;Invokes cont in each element of a proof stream
  (cond ((null stream1) ())
        (t (invoke-after (invoke cont (caar stream1) (cdar stream1))
                         `(invoke-in-each* ,(cdr stream1) ,cont)))))

(defun invoke-after*(delayed continuation)
  (invoke-after (invoke delayed) continuation))
(defun invoke-in-each*(delayed continuation)
  (invoke-in-each (invoke delayed) continuation))
```