TECHNICAL REPORT NO. 182

TRANSLITERATING PROLOG INTO SCHEME

by

Matthias Felleisen

October, 1985

This material is based on work supported by the National Science Foundation under grant number DCR 85-01277.

Transliterating Prolog into Scheme

SUMMARY

Matthias Felleisen

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, Indiana 47405

1. Why yet another implementation of Prolog

Prolog implementations come in a great variety. There is the "shortest", the "most elegant", the "most efficient", and the "most-what-have-you" implementation (see for example [2]). Why propose yet another implementation of Prolog?

All of the published Prolog implementations are either interpreters or compilers. Our implementation *transliterates* Prolog entities into corresponding Scheme ([14]) constructs on a one-to-one basis: relations to functions, backtrack computations to first-class continuations, variables to references, lists to lists, atoms to atoms, etc.The transliteration is not quite a compilation because it does not code the unification algorithm in line. It differs from an interpretation in two respects. First, the symbolic manipulation of data structures representing Prolog procedures is completely eliminated. Instead of searching a (symbolic) data base the function corresponding to a relation is called. Second, Prolog and Scheme programs can now interact in a natural way since they are living on the same level. For example, functional Prolog procedures, i.e., relations which return one result per application, can be directly written as Scheme functions. Scheme's capability to package functions into modules facilitates the splitting of Prolog's global database into several local ones.

Most importantly our presentation illustrates a way of embedding one programming language in another. Transliteration as presented here is based on semantic algebras as discussed by Mosses ([4]) and Clinger, Friedman, and Wand ([11]). Given a powerful target language a programmer specifies the semantics of a linguistic facility of some source language by an expression in that target language. Assuming a syntax preprocessor he can then syntactically abstract the new construct making it conceptually available in the target language. We will show how to apply the principles to a complete and practical programming language.

The paper is organized as follows. Section 2 briefly introduces the Scheme-dialect of Lisp, especially first-class continuations and Scheme's syntax transformer. In Section 3 we present the implementation of the transliteration process. In the fourth section examples of mixed Prolog/Scheme programs are discussed. The last section is devoted to related work.

This material is based on work supported by the National Science Foundation under grant DCR 85-01277.

2. Scheme

Scheme is a lexically scoped dialect of Lisp. It distinguishes itself from other languages by its treatment of functions and continuations as first-class objects.

2.1 Standard Scheme

Figure 1 shows the syntax of standard Scheme. The semantics is roughly the same as for similar expressions in Common Lisp except for the missing `function` and `funcall` forms. Scheme also contains standard list processing functions like `cons`, `car`, `cdr`, `list`, `append`, and `map`, as well as predicates like `equal?`, `eq?`, and `list?`. Reference cells which are roughly equivalent to variables in other languages are first-class objects. They are created and initialized by the function `ref`. The function `ref?` tests the ref-hood of an object and `deref` returns its value. The function `set-ref!` changes a ref-cell's value.

```
⟨expression⟩ ::=
     ⟨constant⟩
   | ⟨identifier⟩
   | ⟨syntactic extension⟩
   | (quote ⟨object⟩)
   | (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
   | (cond [⟨expression⟩ ⟨expression⟩+]+)
   | (begin ⟨expression⟩+)
   | (lambda (⟨identifier⟩*) ⟨expression⟩+)
   | (lambda ⟨identifier⟩ ⟨expression⟩+)
   | (define ⟨identifier⟩ ⟨expression⟩)
   | (let ([⟨identifier⟩ ⟨value⟩]*) ⟨expression⟩+)
   | (let* ([⟨identifier⟩ ⟨value⟩]*) ⟨expression⟩+)
   | (letrec ([⟨identifier⟩ ⟨value⟩]*)⟨expression⟩+)
   | (iterate ⟨identifier⟩([⟨identifier⟩ ⟨value⟩]*) ⟨expression⟩+)
   | ⟨application⟩
⟨value⟩, ⟨tag⟩, ⟨function⟩ ::= ⟨expression⟩
⟨syntactic extension⟩ ::= (⟨keyword⟩ ⟨object⟩*)
⟨application⟩ ::= (⟨function⟩ ⟨expression⟩*)
```

Figure 1 : Syntax of Scheme

Continuations are obtained by the function `call-with-current-continuation` , abbreviated `call/cc`. The function `call/cc` evaluates its argument and applies it to the current continuation which is represented as a function of one argument. It stands for the remainder of the computation from the application of `call/cc`. This continuation may be invoked with any value at any time with the effect that this value is the value of the `call/cc`-application.

A typical example of a continuation is a return or `exit` continuation. Suppose a function wants to return prematurely. When the function is called, it simply catches the current continuation.

The continuation is then bound to the identifier `exit`, and it may be invoked anywhere within the function for immediately returning a value. If `<f-body>` is the function body containing expressions of the form `(exit <return-val>)`, then the following program schema implements the desired control strategy:

```
(lambda (<arg1> ...  <argn>) (call/cc (lambda (exit) <f-body>)))
```

Another application of continuations involves backtracking. Imagine a situation where two expressions are evaluated in sequence with the first one having the option of backtracking by invoking a function `backtrack`. Disregarding side-effects we can simply write this as

```
(begin (call/cc (lambda (backtrack) <exp1>)) <exp2>)
```

In the case where the second expression should not be evaluated unless expression `<exp1>` explicitly backtracks, we can use an exit continuation:

```
(call/cc (lambda (exit)
     (call/cc (lambda (backtrack) (exit <exp1>)))
     <exp2>))
```

Backtracking with side-effects is only slightly more complicated. Instead of passing a value to `backtrack` which is discarded anyway, we pass it some undo-information. The `backtrack` continuation has to include a call to the undo-function and must then invoke the second expression. The program schema becomes:

```
(call/cc (lambda (exit)
     (undo-function (call/cc (lambda (backtrack) (exit <exp1>))))
     <exp2>))
```

Backtracking is now accomplished by the application `(backtrack undo-information)`.

2.2 Scheme's syntax transformer

Scheme 84 provides a powerful syntax transformer ([7]). It allows the programmer to program syntax transformations by specifying (syntactic) input patterns and (semantic) output patterns. As soon as the preprocessor recognizes the keyword in the first position of an application and with it the syntactic pattern, it expands the program text according to the semantic pattern. The `let`-statement, for example, could be defined as:

```
(extend-syntax (let)
  [(let ([<id> <val>] ...) <exp> ...)
   ((lambda (<id> ...) <exp> ...) <val> ...)])
```

The syntactic pattern says that a `let`-expression is a list with `let` in its first position, zero or more pairs "`([<id> <val>] ...)`" in the second followed by expressions "`<exp> ...`". The second line, or semantic pattern, specifies that a `let`-expression is equivalent to an application where "`<exp> ...`" are the function bodies, "`<id> ...`" are the formal parameters, and "`<val>...`" are the actual arguments. Patterns like "`<x> ...`" stand for 0 or more objects of the type `<x>`.

The list "`(let)`" may contain more identifiers in order to specify syntactic and semantic keywords. Syntactic keywords may be used to make the syntactic pattern more readable. Semantic keywords introduce new bindings within the scope of the semantic expansion. An example would be a syntactic extension for the above backtracking facility where `backtrack` becomes a new binding:

```
(extend-syntax (seq backtrack)
  [(choose <exp1> <exp2>)
   (call/cc (lambda (exit)
        (undo-function
           (call/cc (lambda (backtrack)
              (exit <exp1>))))
       <exp2>)))])
```

Note that the new binding for `exit` remains hidden from the user since it is not a keyword, i.e., if the identifier `exit` appears as a free identifier in either `<exp1>` or `<exp2>`, it is alpha-substituted.

Semantic patterns may contain values and functions from the lexical scope of a syntax declaration. The import is accomplished via a `with`-statement which is wrapped around the semantic pattern mimicking a `let`-statement. The effect is comparable to the coding of calls to routines from the run-time library by a compiler. The examples in Section 3 will clarify its usage.

3. The transliteration of Prolog

Prolog's distinguishing features include computation by backtracking and call-by-unification. The latter is based on the unification procedure and logical variables (see [9]).

Logical variables are transliterated to ref-cells. For readability we introduce the operations `free?`, `newLV`, and `unset!*`. The function `derefLV` makes the logical contents of a variable available to other Scheme functions. The implementation of the unifier is straightforward. Since logical variables are changed destructively, they are collected in an undo-list in case the unification fails.

A Prolog procedure is mapped into a `relation`. The clause heads become the formal parameter patterns; the clause bodies correspond to the function bodies which are invoked if the respective formal pattern unifies with the argument pattern. Before arguments are passed to a `relation` they are evaluated just as for any other Scheme function.

The actual unification process is implemented as the syntactic extension `unify`. The process can be seen as an instance of the backtracking process described in Section 2. If the first parameter pattern unifies with the argument pattern, the computation proceeds with the clause bodies. Otherwise the unification backtracks, undoes the variable bindings, and tries the next parameter pattern.

The backtrack continuation `fail` is also used to implement Prolog's search behavior. When a `relation` succeeds, the current backtrack continuation `fail` is returned so that the `relation` can be resumed for more results. The returned `fail` continuation becomes the current backtrack continuation of the caller. The current backtrack continuation `fail` is passed along to relations in the clause body as an extra argument in case none of the parameter patterns unifies with the argument pattern. The management of the backtrack continuations is done in `clauses`.

Figure 2 shows the *entire* implementation of our transliteration process. Visible to the user are `derefLV` and the syntax declarations. Note that `cut` is implemented as a relation of no arguments local to each relation. Its effect is to take the current `fail` continuation and return the one that was passed to the relation, i.e., `bt`, thus making it the current backtrack continuation. In other words, if the clause body ever backtracks to this point the search process resumes the caller's continuation. The backtrack points of the current relation are discarded.

```
 (let ([newLV   (lambda () (ref 'unbound))]
       [free?   (lambda (var) (eq? (deref var) 'unbound))]
       [unset!* (lambda (l) (map (lambda (var) (set-ref! var 'unbound)) l))])
    (define derefLV (lambda (v)
                     (cond [(atom? v) v]
                           [(ref? v) (if (free? v) v (derefLV (deref v)))]
                           [t (cons (derefLV (car v)) (derefLV (cdr v)))])))
    (extend-syntax (letLV)
      [(letLV (<lv1> ...) <exp1> ...)
       (with ([newLV newLV])
       (let ([<lv1> (newLV)] ...) <exp1> ...))])
; -------------------------------------------------------------------------
  (letrec ([unifier! (lambda (pat1 pat2 fk)
                       (iterate U ([pat1 pat1][pat2 pat2][undo ()])
                             (cond [(eq? pat1 pat2)  undo]
                                   [(ref? pat1) (unify_variable! pat1 pat2 undo U)]
                                   [(ref? pat2) (unify_variable! pat2 pat1 undo U)]
                                   [(and (list? pat1) (list? pat2))
                                    (U (cdr pat1)(cdr pat2)
                                       (U (car pat1) (car pat2) undo))]
                                   [t (fk undo)])))]
          [unify_variable! (lambda (var pat undo U)
                            (cond [(free? var) (set-ref! var pat)
                                               (cons var undo)]
                                  [t (U (deref var) pat undo)]))])
; -------------------------------------------------------------------------
    (extend-syntax (unify fail)
       [(unify arg-pat ([<param-pat1> <exp1> ...] ...) <post-action>)
        (with ([unifier! unifier!][unset!* unset!*])
        (call/cc (lambda (exit)
            (unset!* (call/cc (lambda (fail)
                (let* ([undo (unifier! arg-pat <param-pat1> fk)]
                       [fail (lambda () (fk undo))])
                    (exit (begin <exp1> ...))))))
            ... ;; for all param-pat
            <post-action>))))])))
; -------------------------------------------------------------------------
(extend-syntax (clauses)
   [(clauses <rel1> ...) (let* ([fail (<rel1> fail)] ...) fail)])
(extend-syntax (relation cut fail)
   [(relation <lvs> [<para-pat-pt1> ...  ...] ...)
    (lambda arg-pat
      (lambda (bt)
        (letLV <lvs>
           (let ([cut  (lambda () (lambda (fail) bt))])
             (unify arg-pat
               ([(list <para-pat-pt1> ...) (clauses <exp1> ...)] ...)(bt))))])
```

Figure 2: A Prolog transliteration into Scheme

4. Examples

The following program is a naive sort program adapted from [5]. Note that the relation `sort` is the result returned from the `letrec` -expression and that `perm`, `append`, and `sorted` are hidden by Scheme's lexical scope. Relations cannot only be the result of expressions but they can also be arguments to functions and relations and can then be called without using a meta-predicate like call. In other words relations-as-functions are first-class objects.

```
(define sort
   (letrec ([sort (relation (L1 L2)
                     [L1 L2 <- (perm L1 L2) (sorted L2)])]
            [perm (relation (L H T v w vw)
                     [() () <-]
                     [L (cons H T) <- (append v (cons H w) L)
                                      (append v w vw)(perm vw T)])]
            [append (relation (H T Y Z)
                      [() Y Y  <-]
                      [(cons H T) Y (cons H Z)  <- (append T Y Z)])]
            [sorted (relation (H1 H2 R)
                      [(cons H1 (cons H2 R)) <- (order H1 H2)
                                                (sorted (cons H2 R))]
                      [(cons H1 ())  <- ]
      sort))
```

The above example can be rewritten using Scheme functions for `append` and `sorted` at the appropriate places.

```
(define sort
  (let ([F-append append]
        [F-sorted (rec loop (lambda (L)
                    (if (null? (cdr L)) t
                       (and (< (car L) (cadr L)) (loop (cdr L))))))])
   (letrec ([perm (relation (L H T V U)
                    [() () <-]
                    [L (cons H T)  <- (append U (cons H V) L)
                                      (perm (F-append (derefLV U)
                                                      (derefLV V)) T)])
           [append (relation (H T Y Z)
                     [() Y Y  <-]
                     [(cons H T) Y (cons H Z) <- (append T Y Z)])])
     (lambda (L1)
       (letLV (L2)
         (let ([next (RelApply (perm L1 L2))])
           (let ([res  (derefLV res)])
              (cond [(eq? next 'no)   nil]
                    [(F-sorted res)   res]
                    [t                (next)])))))))
```

For readability we have introduced the function `RelApply`. It is defined by:

```
(define RelApply
  (lambda (rel)
    (call/cc (lambda (backtrack) (rel (lambda () (backtrack 'no)))))))
```

5. Related Work

In the preceding sections we have shown how to implement a schema for the transliteration of Prolog programs into Scheme functions. The extended language is a union of Prolog and Lisp. Our work can be compared both to other implementations of Prolog and Prolog embeddings into Lisp.

Pure interpreters as in [3], [12], or [6] are almost incomparable to our transliteration. Although advanced programming techniques like continuation-passing-style are used they follow the standard strategies for interpreters. Prolog procedures are represented by conventional data structures. One of the major consequences is that interpreters have to search for a relation in a data base. Interpreters also often have their own read-eval-print loop on top of the one of their host language.

Compilers like those in [1] or [15] translate Prolog into some tailor-made assembly language and optimize the resulting code in many ways. One of the major optimizations is the in-line coding of the unification algorithm. We can achieve the same effect with the addition of one more syntactic extension. The details are exhibited in the appendix since they would not have contributed to our presentation of the principle. However, we would like to point out that with this enhancement the program becomes the shortest published (primitive) compiler.

Both pure interpreters and compilers like those in the two preceding paragraphs cannot provide a natural and simple interface to Lisp programs. QLOG ([8]) and POPLOG ([10]) explicitly address the implementation of a union of Lisp and Prolog. Programs written in either dialect can use facilities of the other.

QLOG is an interpreter-based language for a Prolog/Lisp mix. The implementation strategy is related to the ones in standard Prolog interpreters. It consists of major additions to a Lisp interpreter. Ten pages are needed for the interpreter extension; twenty more for the adaptation of the user-interface to INTERLISP.

Mellish and Hardy's compiler-based POPLOG comes closest to our Scheme extension. The resulting language mixture is similar; the implementation strategy is almost the same. It differs from ours in the control management which in the extended POP-11 is based on functions because of the lack of first-class continuations. The actual implementation of POPLOG is quite different since the Prolog part of POPLOG is compiled into asssembly code. POPLOG has achieved a similar result but building or changing compilers and interpreters every time a new linguistic facility is to be added is inconceivable. It is too costly for experimenting with languages and language mixtures.

Transliteration is a valuable alternative for the extension of languages with major features. QLOG, POPLOG, or LOGLISP ([13]) are certainly good languages but they do not offer the same potential for further extensions like the extended Scheme. Since it is no problem to add object-oriented language facilities to Scheme (by transliteration, of course), one can imagine an object/logic-oriented style. Similarly Scheme modules, continuations, and first-class functions can enhance pure Prolog programming. It is obvious that the coexistence of Scheme and Prolog programs deserves more research.

Acknowledgement. We wish to thank H.Ait-Kaci, S.Danforth, D.Friedman, R.Nasr, and M. Wand for comments on an earlier draft of the paper.

6. References

[1] Bowen, D.L, et.al., A portable Prolog compiler, *Proc. Logic Programming Workshop*, pp. 74–83, 1983.

[2] Campbell, J.A., *Implementations of Prolog*, Ellis Horwood, 1984.

[3] Carlsson, M., On implementing Prolog in functional programming, *New Generation Computing* **2**,347-359, 1984.

[4] Clinger, W., D. Friedman, M. Wand, A scheme for a higher-level semantic algebra, in *Algebraic Methods in Semantics*, J.Reynolds, M.Nivat (eds.), 1985.

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

[6] Kohlbecker, E., eu-Prolog, *Tech. Rep. No. 155*, Indiana University, Computer Science Dept. 1983.

[7] Kohlbecker, E., *Syntactic Preprocessing in a Lexically Scoped Language*, Ph.D. dissertation in progress, Indiana University, Computer Science Dept., 1985.

[8] Komorowski, H.J., QLOG - The programming language for Prolog in Lisp, in *Logic Programming*, K.L. Clark, S.-A. Tarnlund, 1982.

[9] Lindstrom, G., Functional programming and the logical variable, *Conf. Rec. 12th Symp. Principles of Programming Languages*, 1985.

[10] Mellish, C., S.Hardy, Integrating Prolog in the POPLOG environment, in [2].

[11] Mosses, P., Abstract semantic algebras!, *Proc. Formal Description of Programming Concepts*, 1982.

[12] Nilsson, M., The world's shortest Prolog interpreter?, in [2].

[13] Robinson, J.A., E.E.Sibert, LOGLISP: An alternative to Prolog, in *Machine Intelligence* **10**, 1982.

[14] Sussman G.J., G. Steele, Scheme: An interpreter for the extended lambda-calculus, *Memo 349*, MIT AI-Lab, 1975.

[15] Warren D., An abstract Prolog instruction set, *Tech. Note No. 309*, SRI, 1983.