AN ENVIRONMENT MODEL FOR THE INTEGRATION OF LOGIC AND FUNCTIONAL PROGRAMMING

Pierre E. Bonzon

University of Lausanne
HEC, 1015 Lausanne, Switzerland

Abstract

We propose a environment model for the evaluation of both function and relation applications; as an illustration, simple extensions of Scheme are introduced, together with their interpreter.

1. Introduction

Early attempts ([l],[2]) toward the goal of integrating logic and functional programming typically led to the introduction into Lisp of a collection of primitives allowing to return, as a list data object, all, or any number of, the tuples satisfying a given predicate. These proposals actually provide an interface between two different computational models, and therefore have been characterized as embedding one programming language into another. Further works [3] were more theoretic in nature, and dealt with the fundamental issue of providing a unified semantics for languages of both types.

Our own approach is of the embedding kind outlined above. It bears strong similarities with the recent works of both Srivastava and alii [4] and Haynes [5], in the sense that it leads to an extension of Scheme (a lexically scoped dialect of Lisp), allowing logic and functional expressions, represented by first class data objects, to be freely mixed, passed as arguments, returned as results, and so on. We thus achieve a functional embedding of logic programming within Scheme. Following Haynes's taxonomy, this embedding can be further described as an environment embedding, in the sense that the embedded and embedding languages share a common environment. It fails however to be a complete environment embedding, in the sense that it does not allow to access and/or modify control information.

With regard to these previous works, our contribution can be described as follows:

- instead of taking care of our extensions by macro expansions, we redefine the semantics of Scheme by modifying the meta-interpreter given in [6]; we are thus led to a new computational model based on two types of environments

This work was done while the author was visiting at the University of California at Santa Cruz.

- building upon this, we incorporate into Scheme the query language introduced in [6].

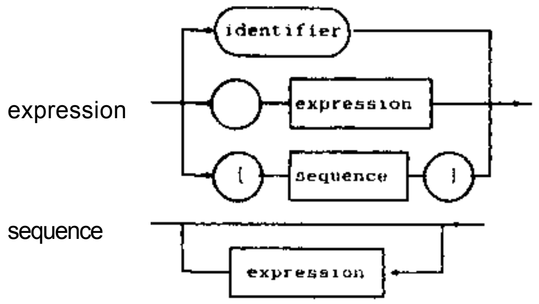
In the resulting model, the concatenation of two lists can be the result of either a function or a relation application:

```
(DEFINE APPEND (LAMBDA (X Y)
                 (COND ((NULL X) Y)
                       (T (CONS (CAR X]

                                (APPEND (CDR X) Y)

 -> APPEND

(APPEND  (A) '(B C))

  -> (A B C)

(DEFINE APPENDR (CLAUSE (NIL ?X ?X)))

 -> APPENDR
(DEFINE APPENDR (CLAUSE
                 ((CONS ?A ?X) ?Y (CONS ?A ?Z))
                 (APPENDR ?X ?Y ?Z)))
 -> APPENDR

(APPENDR '(A) '(B C) ?X)

 -> ((APPENDR (A) (B C) (ABC)))

(APPENDR ?X ?Y  (ABC))

 -> ((APPENDR () (ABC) (ABC))
     (APPENDR (A) (B C) (ABC))
     (APPENDR (A B) (C) (ABC))
     (APPENDR (ABC) () (ABC)))
```

In the following sections, we first consider a simple subset of Scheme, called Core Scheme. Next, we introduce a variant of Core Senear which can be used to define and apply relations (cr predicates) rather than functions. By combining these two models, we then obtain a logical extension of Core Scheme, allowing the application of both functions and relations. We conclude by showing how a query language can be fitted into this extension.

2. A Model for a functional subset of SCHEME

Let us consider a simple subset of Scheme, thereafter called Core Scheme, whose expressions have the following syntax:



Following the Lisp terminology, we .shall call an identifier an atom, the expression (), the empty list, and all other expressions, non-atomic lists. The semantics of Core Scheme will be defined by a function Eval of two arguments, i.e. an expression and an environment. This interpreter, similar to the metacircular interpreter given in [6], will be given in PASCAL, and rely on the implementation of an abstract data type called S-expression. This implementation will be viewed through a type List, operators NewList, NowAtom, Cons and Append, selectors Car and Cdr, predicates Atom, Eq, Null, and mutators RpiaCa and RplaCd. All of these operators correspond to the usual Lisp functions, except for the constructors NewList and NewAtom which return a empty list and a symbolic atom.

Taking into account the conventions given below for representing expressions, function Eval is:

```
function Eval(Expr, Env:List): List;
b»gin case TyppOfExpr<Expr) of
        S«lfExpr:Eval:= Expr,
        VarExpr:Eval:=EvalVar(Expr,Env);
        QuoteExpr:Eval . =Car(Cdr(Expr)),
        CondExpr:Eva l :=EvalCond(Cdr(Expr),Env) .
        DefExpr:Eval:= EvalDef(Car(Cdr(Expr)),
                                 Eval(Car(Cdr(Cdr(Expr))).Env).
                                 Env ; ;
        Lambda Expr :Eval : =Eval Lambda (Expr , Env ) ,
        CallExpr:Eval:=Apply(Eval(Car(Expr),Env).
                                EvalLIBt(Cdr(Expr).Env))
       end
end ;
```

The expressions recognized by the interpreter are:

2.1 Self-evaluating expressions

Self-evaluating expressions are expressions which evaluate to themselves. In Core Scheme, they are

- the self-evaluating atom NIL and the expression () standing both for the empty list
- the self-evaluating atom T , standing for the boolean value true (the boolean value false being represented by the empty list)
- the self evaluating atoms CAR, CDR, CONS, ATOM, EQ and NULL, standing for the corresponding primitive operators (or functions).

2.2 Variables

All atomic expressions which are not self evaluating are treated like variables: the value returned by the interpreter is looked up in the environment supplied as second argument. The pair formed by an atom and its associated value being called a binding, a list of bindings defines a partial environment (called frame in [6]). Environments *are* lists of partial environments. It should be noted that environments are not Scheme objects, and are introduced for describing the computational model.

2.3 Quoted expressions

Quoted expressions, prefixed with a quote, are encoded as lists with the atom QUOTE as first element: the value returned is their second element.

2.4 Laabda expressions

They represent functions, and have the general form

(LAMBDA ("arguments") "body")

where "arguments' is a possibly empty sequence of atoms representing the function formal argument list, and "body" is a sequence of call expressions (see 2.7 below) representing its body.

The value returned by the interpreter is a function closure (called a procedure *in* [o]), i.e. the association of the expression and its current environment. A closure will be viewed as an instance of an extended abstract data type, called S-Closure, implemented as a non atomic List variant, with additional constructor NewClosure, predicate Closure and selector Environment. It will be represented as

(LAMBDA (arguments") "body")["environment"]

stressing the fact that the associated environment, accessible through the selector Environment, is not "consed" with the lambda expression.

2.5 Conditional expressions

The general form of conditional expressions is

(COND "easel" ... "casen")

They have their usual Lisp interpretation.

2.6 Define expressions

Define expressions are used to assign a permanent value to an atom They have the form

(DEFINE "atom" "expression")

with the value of "atom" to be that of expression".

The value returned by the interpreter is the atom. As a side effect, a new binding is introduced in the head of the current environment.

2.7 Call expressions

They are all the remaining expressions, and are interpreted as function applications handled by function apply. To be legal, their head must be an atom or a lambda expression, and must evaluate to a primitive operator or a function closure. Actual arguments come from a sequential evaluation of their queue. Whereas primitive operators can be applied directly, the expressions contained in the body of function closures are evaluated in sequence, the application receiving the value of the last expression. The environment is taken from the closure and augmented with a partial environment build by function BindList, which takes the list of formal arguments from the closure and bind them with the list of actual arguments.

Function Apply can be defined as follows:

```
function Apply(Op,Ary:List):Llat;
bag in caaa TypaOfOp(Op) of
        CarOp: Apply:«Car(Car(Arg));
        CdrOp: Apply:=Cdr<Car lArg));
        ConaOp: Apply:"ConstCar(Arg).Car(Cdr(Arg>));
        AtoaiOp: If Atom (Car (Arg) )
                than Apply;'NawAtomlT ')
                alae Apply:sNawLlat;
        EqOp:  If Eq(Car(Arg),Car(Cdr(Arg)))
              than Apply:*NewAton("T ')
              alae Apply:sNawLiat;
        NullOp:  if NulKCar(Arg) )
                than Apply:=NawAton( T ')
                alaa Apply::NawLiat;
        ProoOp: Apply::EvalSaquance(Cdr(Cdr(Op)),
                          Cona(BindLiat(Car(Cdr(Op)).
                                        Arg),
                              Environment(Op)))
      and
and;
```

3. A logical variant of Core Scheme: a simple model of logical programming

Let us now introduce a variant of Core Scheme, allowing to define and evaluate relations (or predicates) rather than functions. Its interpreter, which uses two types of environments, is as follows:

```
function Eval(Expr.FunEnv,LogEnv:List):Li at;
bag in caaa TypaOfExpr(Expr) of
        LogVarExpr:Eval:=EvalLogVar(Expr,LogEnv);
        VarExpr:Eval:=EvalVar(Expr,FunEnv,LogEnv);
        QuoteExpr:Eval:*Car(Cdr(Expr));
        DafExpr:Eval:«EvalDaf(Car(Cdr(Expr)).
                              Eval(Car(Cdr(Cdr(Expr))),
                                   FunEnv,LogEnv),
                              FunEnv);
        ClauaaExpr.Eval:«EvalClauae(Expr,FunEnv,LogEnv);
        CallExpr:Eval:«Saarch(EvalQuary(Expr,FunEnv,LogEnv),
                              Eva1Ca11a(Expr,FunEnv,LogEnv))
      and
and;
```

The expressions recognized by the interpreter are:

3.1 Logical Variables

These are all identifiers prefixed with a question mark. As they can be bound by predicate application, the value returned by the interpreter is looked up in the current logical environment (since clause definitions cannot be block structured, each logical environment is simply a list of bindings, i.e contains just one partial environment). If no associated value is found, a variable closure is returned, associating the logical variable and the current logical environment. Whenever the associated value is » logical variable, possibly in closure form, it gets evaluated again. Logical variables in closure form are evaluated in their own environment, which overrides the current environment. They are implemented as atom variants, with selector LogEnv accessing the associated logical environment. Note that there is no need, in this particular model, to rename variables having the same name in different clauses, since they are distinguished by the environment they are associated with.

3.2 Functional variables

All other atoms are treated as functional variables, and their values looked up in the current functional environment. If no value is found, then the value of the logical variable of the same name (i.e. prefixed with a question mark) is returned.

3.3 Quoted expressions

Quoted expressions are defined and treated in the same way as in Core Scheme. They can be used to represent the equivalent of Prolog terms build with functors (e.g. '(F A) represents F(A)).

3.4 Clauae expressions

They represent logical clauses and have the form

(CLAUSE ("arguments") "body")

where "arguments" is a possibly empty sequence of terms representing the clause formal argument list, and "body" is a possibly empty sequence of call expressions (see 3.6) representing its body. The value returned is a predicate, defined as a list of clause closures, and containing in this case just one closure, associating the given expression and the two current environments. As before, closures are implemented as non atomic List variants, with selectors FunEnv and LogEnv accessing the associated functional and logical environments.

Examples

(CLAUSE (?X ?Z) (FATHER ?X ?Y) (FATHER ?Y ?Z))

will be returned as

((CLAUSE (?X ?Z) (FATHER ?X ?Y)(FATHER ?Y ?Z))
["environments"])

3.5 Define expreaalons

Define expressions are represented and treated much in the same way as in Core Scheme, except for the case when the value is a predicate: if the atom already evaluates to a predicate, then these two predicates are appended, a fucility for the definition of multiple clause predicates.

3.6 Call expressions

Call expressions are all other expressions. They are interpreted here as predicate applications. To be legal, their head must be an atom or a clause expression, and must evaluate to a predicate or a variable closure. In the first case, the value returned is the list of all instances of the call expression that can be deduced from the predicates defined in the current functional environment. In the second, an empty list is returned. Predicate applications are handled by function Search, which, following most Prolog interpreters, performs a depth-first, left-to-right search for the list of appropriate instances, and uses the unification algorithm without the occur check. The arguments of function Search are:

- a query closure, associating a call expression whose arguments have been evaluated, and two environments

  Example

  If variables X and ?X are unbound in their current environment, the call expression

  (GDFATHER X JIM)

  will produce the query closure

  (GDFATHER ?X JIM)["environments"]

- a list of calls, each of them formed by the association of a predicate and an argument closure, this closure associating itself an unevaluated list of arguments and a pair of environments.

  Example

  The call expression

  (GDFATHER X JIM)

  could produce a list of just one call, defined as

  ((((CLAUSE (?X ?Z) (FATHER ?X ?Y) (FATHER ?Y ?Z))
  ["environments")
  (CLAUSE (?X ?Z) (FATHER ?X ?Y) (MOTHER ?Y ?Z))
  ["environments"]).(X JIM)["environments"]))

Function Search is defined as follows:

```
function  Search(Query,Calla:Liat) :LIst;
begin if Null(Call*)
      then Search:*Cona(Cona(Car(Query),
                              Instance(Cdr(Query),
                                       LogEnvlQuery))),
                    NewLlat)
      elae oaae TypoOfOp(Car(Car(Calle))) of
              PredOp:Search:"TryEach(Query,
                                     Car(Car(Calle>),
                                     Cdr(Car(Calla)),
                                     Cdr(Call.));
              UndefOp:Search:>NewLlat
           end
end;
```

Function Search works as follows:

- if there is no calls (i.e the query has been reduced to a fact), then function Search returns a list containing the query with its arguments instantiated in their associated logical environment

- otherwise, if it is not undefined, the first call is passed to function TryEach, together with a continuation containing the remaining calls.

Function Instance returns a copy of its first argument with all its logical variables evaluated and their associated values instantiated in turn. Function TryEach constructs the list of all deducible instances by trying in turn each of the clause closures contained in the predicate:

```
function TryEach(Query,Pred,CallArg,Cont:List) .List;
begin if Null(Pred)
      then TryEach:=NewList
      elss TryEach:"Append(Try(Query,CartPred),CallArg,Cont),
                           TryEach(Query,Cdr(Pred),Cal1Arg,
                                   Cont))
```

Function Try attempts to unify the current call arguments in their associated logical environment, with the candidate clause argument.** in a new environment. In case of success, function Search is entered recursively after adding to the continuation the calls from the candidate clause body. All arguments are evaluated before unification:

```
function Try(Query,Clause,CallArg,Cont:Liat):Liat;
var ClauaeLogEnv,TralJ:List;
begin ClauaeLogEnv:*NewEnv;
      Trail:«NewList;
      if Unify(EvalLiEt(CallArg,FunEnv(CallArg) ,
                        LogEnvfCallArg)),
               LogEnv(Ca11Arg),
               EvalLIst(Car(Cdr(Clause)),FunEnv(Clause),
                        ClauseLogEnv),
               ClauaeLogEnv,
               Trail)
      then Try:«3earch(Query,
                       Append (EvalBody( Cdr (Cdr (Clauae) ).,
                                        FunEnv(Clauae) ,*
                                        ClauseLogEnv),
                               Cont))
      elae Try:=NewList;
      ReetaurefTrall)
end;
```

Function EvalBody returns the list of calls from the candidate clause body.

In the unification process, when a free variable is unified with a term, the environment associated with this variable gets a new binding associating the variable and the term instance in its current logical environment. In order to allow backtracking (i.e. to be able to restaure logical environments in the state they were before trying a particular candidate clause), a trail is used to keep track of the chronological order of bindings.

4. Logical Scheaie: the Integration of Core Scheme and its logical variant

Logical Scheme, the integration of Core Scheme and its logical variant, allows to define and apply functions and predicates. Furthermore, expressions of both kinds can be freely mixed.

Exaaple

For illustration purposes, we shall rely on an extended Core Scheme, where numeric atoms, as well as arithmetic operators, are self-evaluating expressions. The following expressions are then meaningful expressions of Logical Scheme:

```
(DEFINE AGE (CLAUSE I'JIM 20)])

(DEFINE YOUNG (CLAUSE (?X) (AGE ?X ?Y) I< ?Y 25)))

(DEFINE AND (LAMBDA IP Q) (CLAUSE (?X) (P ?X ?Y)
                                       (Q ?X))))

(DEFINE YOUNGFATHER (AND FATHER YOUNG))
```

While the clause expression assigned to atom YOUNG contains a function application, the body of the lambda expression assigned to atom AND is a clause expression. Finally, atom YOUNGFATHER is defined as a clause expression resulting from a function application whose arguments evaluate to predicates.

The interpreter for Logical Scheme is the union of the two previous interpreters, with its last case element modified as follows:

```
CallExpr:
case TypeOfCall (Eval (Car (Expr ), FunEnv, LogEnv ) ) of
  FunCall:Eval:=Apply(Eval(Car(Expr),FunEnv,LogEnv),
                      EvalLiat(Cdr(Expr),FunEnv,LogEnv),
                      LogEnv);
  PredCall:Eval:=Search(EvalQuery(Expr,FunEnv,LogEnv),
                        Eva 1Ca11s(Expr,FunEnv,LogEnv))
end
```

In order to allow predicate applications within function applications, function Apply has an additional argument. Conversely, in order to allow function applications within predicate applications, function Search has now the following form:

```
function Search(Query.Calls:Ltst):List;
begin
 If Null(Calls)
 then Search:=Con*(Cons(Car(Query),
                         Ins tance(Cdr(Query),LogEnv(Query))),
                   NewList)
 else case TypeOfCal1(Car(Car<Calls))) of
      FunCall:if Nul1(Apply(Car(Car(Calls)),
                      EvalLi»t(Cdr(Car(CallB)),
                               FunEnv(Cdr(Car(Calla))),
                               LogEnv(Cdr(Car(Calla)))),
                      LogEnv(Cdr(Car(Calla)))))
              then Search:=NewList
              else Search:»S©arch<Query,Cdr(Calls));
      PredCall^case TypeOfOp(Car(Car(Calls))) of
               PredOp:Search:«TryEach(Query,
                                      Car(Car(Calls)),
                                      CdrfCar(Calls)),
                                      Cdr(Calls));
               UndefOp:Search:*NewList
              end
      end
end;
```

This new function definition reflects the interpretation given, in a predicate body, to a function application: if this function evaluates to false, the predicate fails; otherwise the predicate evaluation goes on. Furthermore, since clause arguments are evaluated before unification, it follows that arguments of clauses can be function applications (as shown in the introductory example defining the RAPPEND relation), and vice-versa.

5. Query Scheie: a query language within Scheae

In the previous example, the predicate application

```
(YOUNGFATHER ?X)
```

follows the definition of a function AND returning the conjunction of two predicates of respectively two and one arguments. A query language is a facility for applying the disjunction and/or the conjunction, as well as the negation, of any number of predicates of any number of arguments.

Example

In the query language introduced in [6], predicate YOUNGFATHER could be defined by

```
(RULE (YOUNGFATHER ?X) (AND (FATHER ?X ?Y)
                            (AGE ?X ?Z)
                            (< ?Z 25)))
```

while, in Query Scheme, it would be is defined by

```
(DEFINE YOUNGFATHER (CLAUSE (?XJ (AND
                                  (FATHER ?X ?Y)
                                  (AGE ?X ?Z)
                                  {< ?Z 25))))
```

In both, this application would also be legal:

```
(AND (FATHER ?X ?Y) (AGE ?X ?Z) (< ?Z 25))
```

It must be noted however that, in [6], the query language is not a part of Scheme. The syntax of Query Scheme being the same as that of Logical Scheme, the new types of expressions are:

5.1 Self-evaluating expressions

In addition to the self-evaluating atoms introduced earlier, the self-evaluating atoms AND, OR and NOT stand for the usual operators defined on predicates.

5.2 Negative clause expressions

Negative clause expressions have the form:

```
(NEGATION ("arguments") "body")
```

where body is a single call expression. The value returned by the interpreter is a predicate containing a negative clause closure. Negative clauses are used to represent negative call expressions, i.e. expressions prefixed with the NOT operator.

Exaaple

The negative call expression

```
(NOT (YOUNGFATHER 'BILL))
```

will become the negative clause application

```
((NEGATION (?X) (YOUNGFATHER ?XJJ BILL]
```

5.3 Predicate expressions

Predicate expressions have the following form:

```
(PREDICATE clause expr." ... clause expr.")
```

The value returned by the interpreter is a predicate. Predicate expressions are thr non-closure representation of multiple clause predicates. They will be used to represent disjunctive calls expressions (see 5.4).

5.4 Call expressions

Call expressions now include conjunctive, disjunctive as well as negative calJ expressions, i.e. expressions whose head is equal to the atom AND, OR or NOT, respectively. These particular call expressions are treated as predicate applications whith a list of calls containing:

- for each conjunction, a clause expression
- for each disjunction, a predicate expression
- for each negation, a negative clause expression.

Example

```
(AND (FATHER ?X ?Yj
     (NOT (YOUNG ?X))
     (OR (HEALTHY ?X) (YOUNG ?Y)))J
```

will be regarded as €;quivalent to

```
((CLAUSE (?X) (FATHER ?X ?Y]
              ((NEGATION (?X) (YOUNG ?X)j ?X)
              ((PREDICATE
                 (CLAUSE (?X ?Y) (HEALTHY ?X))
                 (CLAUSE (?X ?Y]   (YOUNG ?Y)))
               ?X ?Y)]
 ?XJ
```

While a predicate call reduces to a predicate application, negative clause call.'., are handled as in most PROLOG implementations, using the closed world assumption: a negative clause succeeds if its calls cannot be satisfied, and fails otherwise. This is reflected in an extension of function TryEach:

```
function TryEach(Query,Pred,CallArg,Cont:Ptr> :Ptr ;
begin
 if  Null(Pred)
 then TryEach:»NewList
 else
 cat* TypeOfExpr(Car(Pred)) of
  ClauseExpr:TryEach:"Append(Try(Query,Car(Pred ) ,
                                  CailArg,Cont) ,
                              TryEach(Query,Cdr<Pr«d) ,
                                      CallArg,Cont)>;
  NegExprtif  Null(TrytQuery,Car(Prad>,CallArg,NewLi»t))
          than TryEach:*Append(Search(Query,Cont) .
                               TryEach(Query.Cdr(Pred),
                                       CallArg.Cont))
          else  TryEach:*TryEach<Query,Cdr(Pred),
                                 CallArg.Cont)
 •nd
end;
```

6. Conclusions

The interpretation of expressions involving function and predicate applications, as given above, mirrors the traditional use of both function applications within logic programming languages such an Prolog, and predicate applications within query languages. It is by no means the only possible way to evaluate such expressions, but not until a clear denotational semantics is agreed upon will an operational semantics possibly be called correct.

7. References

[I] Robinson, J.A.. and Sibert, E.E., LOGLISP: Motivation, Design and Implementation, in: K.L. Clark and S.-A. Tarnlund feds), Logic Programming, Academic Press, New York, pp. 299-314 (1982).

[2] Komorowski, H.J., QLOG - The Programming Environment for PROLOG in LISP, ibidem, pp. 315-324 (1982).

[3] DeGroot, D. and Lindstrom. G. (eds), Logic Programming / Functions, Relations, and Equations, Prentice-Hall, Englewood Cliffs,1986

[4j Srivastava, A., Oxley, D., and Srivastava, D., An(other) Integration of Logic and Functional Programming, in: Proceedings of the IEEE Symposium on Logic Programming, Boston, pp. 254-260 (1985)

[5] Haynes , Ch.T. , Logic Continuations, in: Proceedings of the Third International Conference on Logic Programming, London (1986)

[6j Abelson, A., and Sussman, G.J., with Sussman, J., Structure and Interpretation of Computer Programs, MIT Press, Cambridge (1985)